GRAPHPILOT

GraphPilot handbook

The GraphPilot documentation in one document:
concepts, guides and reference.

BUILT
October 26, 2026

ONLINE
docs.graphpilot.io

Contents

Get started 3

Introduction 3
How GraphPilot works 5
Features and limitations 8
Quickstart: GraphQL 13
Quickstart: HTTP 19

Connect your origin 25

Connection to the origin 25
Apollo Server 31
GraphQL Yoga 35
PostGraphile 38
Federation routers 42
Other servers 44

Caching 47

How a request flows 47
When a response is cached 50
Cache statuses (gp-cache header) 57

How long

Overview 62
From your origin (Cache-Control) 65
From the schema (@cacheControl) 68
Stale windows (stale-while-revalidate) 74
To your clients (cache.policy.downstream) 77

Who shares an entry

Overview 79
In the schema (@cacheControl scope) 85
In gpilot.toml (cache.policy) 91
Authenticated requests 102
Revoking access 107

Header and cookie filters (gpilot.toml) 110
Request collapsing 116

Invalidation 119

Overview 119
Automatic, from mutations 121
Selection set expansion 127

Surrogate keys

Overview 133
Set keys in the schema (@surrogateKey) 138
Set keys from your origin (Surrogate-Key header) 153
How keys are built 156

Purge on demand (API, CLI, portal) 158

Security 169

Security and data protection 169
Privacy and personal data 170
Authentication (JWT, API keys) 175
Hardening (request limits) 182
Persisted queries 191
Request signing (HMAC) 196
Origin allowlist (edge IP ranges) 204

Deploy and operate 207

Deployments 207
Deploy from CI 213
Staging and rollback 218
Edge locations 221
Shielding 227
Quotas and lean mode 233
Go-live checklist 240

Guides 244

Overview 244
Coming from another CDN 244
Caching a REST API 247
Partitioning your cache 252
Lists and creates 259
Caching per language 262
A/B testing with a cookie 265
Caching per Auth0 organization 267
Purging from a CMS webhook 270
Telling your clients apart 273
Retiring old mobile app versions 278
Migrating to trusted documents 279
Subscriptions and WebSockets 282
Promoting from staging in CI 286
Triaging an alert 290
Keeping request telemetry minimal 292

Troubleshooting 296

By symptom 296
Why is nothing cached? 300
Verifying a purge 305
Debugging surrogate keys 309
Correlating with your logs 311

Portal 315

Overview 315
Organizations and members 317
API keys 321
Custom domains 324
Billing and usage 327
Services and settings 336
Insights 346
Alerts 353
Deployments page 356
Purges page 362
Queries page 366
Schema and config findings 368
Your account 372

CLI 376

Overview 376
Installation 381
Signing in 386
Profiles 391
Command reference 394

Reference 419

Configuration (gpilot.toml) 419
GraphQL directives 449
Policy expressions 455
HTTP headers 471
Error and log codes 483
Permissions 502
Limits 506
GraphPilot API 514
Performance and benchmarks 532
Glossary 538
Runtime versions 542
Versioning and deprecation 543

Get started

GET STARTED

GraphPilot documentation

GraphPilot is a managed edge cache and gateway that sits in front of your API, whether it speaks plain HTTP, GraphQL, or both. It caches responses at the edge on the rules your origin sends, lets you tag them and purge them by key, and protects your origin: it verifies credentials, limits what a request may carry, and signs every request it forwards to you. For a GraphQL API it goes further: it reads your schema, so a mutation clears the cached responses that contained the objects it returned, and it bounds what an operation may ask for. You describe it with one config file, plus your schema for GraphQL. There is no infrastructure to run.

Start here

Quickstart

From a new account to a cached response in front of your own API, and a way to clear it.

GraphQL quickstart · HTTP quickstart

How GraphPilot works

Where GraphPilot sits, what it does for every request and what it adds for GraphQL, and what a service, a config, a schema and a deployment are.

Read the overview

Features and limitations

The feature matrix for plain HTTP and GraphQL, what is deliberately left to your origin, and the current limitations.

Read the feature matrix

Reference

Every configuration setting, error code and response header, the policy expression grammar, and the CLI.

Configuration · Error codes · CLI

I want to…

- **Cache signed-in data per organization, not per user**: partitioning your cache
- **Put GraphPilot in front of a REST API**: caching a REST API
- **Show a new item in a cached list right away**: lists and creates
- **Clear cached responses when my data changes**: cache invalidation
- **Find out why nothing is cached**: why is nothing cached?
- **Name my clients, retire old versions, refuse unknown callers**: telling your clients apart
- **Deploy from GitHub Actions or GitLab CI**: deploy from CI
- **Follow a request into my own logs**: correlating with your logs
- **Allow only the operations my clients send**: migrating to trusted documents
- **Test changes on staging, and roll back**: staging and rollback
- **Serve subscriptions over WebSockets safely**: subscriptions and WebSockets

Every guide is listed on guides.

Get an account and the CLI

Sign up at cockpit.graphpilot.io. If you do not belong to an organization yet, create one with **New organization**; the dialog asks you to accept the Terms of Service. The portal at cockpit.graphpilot.io is where you manage it. Then install the CLI and log in:

```
brew install graphpilot/tap/gpilot
gpilot auth login
```

Linux, Windows and the other install methods are on installing the CLI. See signing in for device-code and API-key logins, and profiles for keeping several identities side by side.

Find your way around

If you are	Read
Evaluating GraphPilot	How it works, features and limitations, then limits
Integrating it for the first time	The GraphQL quickstart or the HTTP quickstart, then the connection to the origin and the go-live checklist
Moving from another CDN	Coming from another CDN
Using a specific server or router	Apollo Server, GraphQL Yoga, PostGraphile, federation routers, other servers
Caching data that differs per user or per organization	Who shares an entry

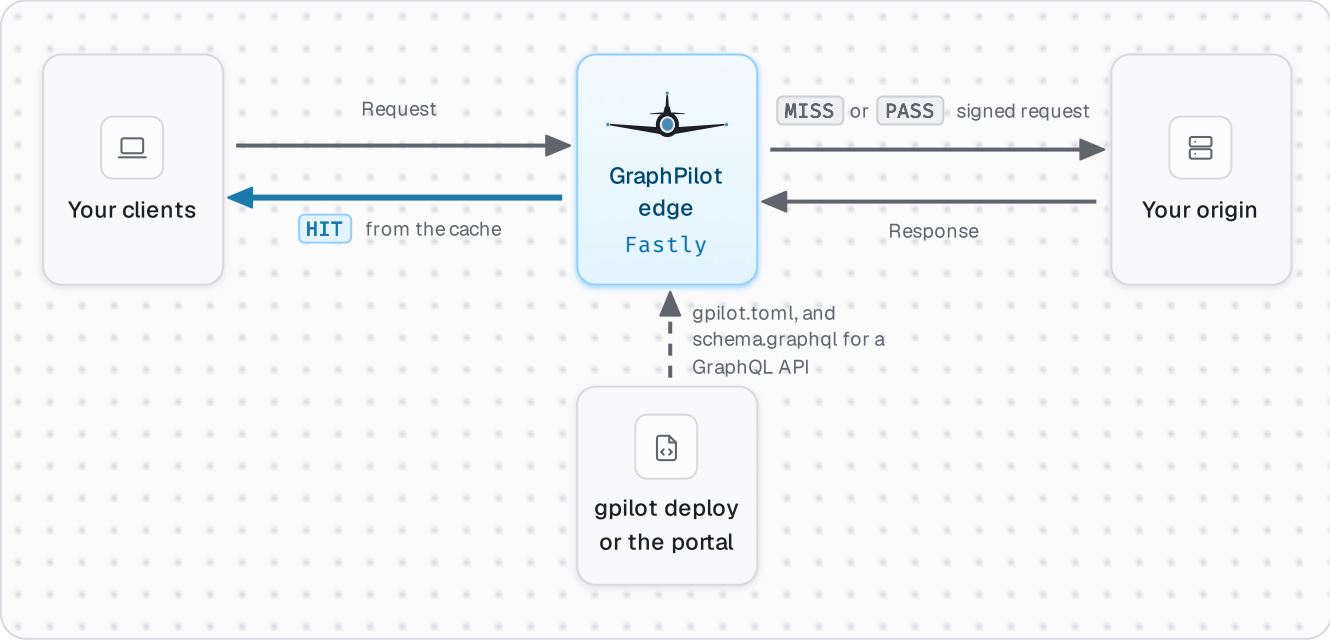
If you are	Read
Clearing the cache when data changes	Invalidation
Working in the portal	Portal overview
Debugging a miss or an error	Troubleshooting, why is nothing cached?, headers, codes
Running it in production	Deployments, deploy from CI, staging and rollback, quotas
Reviewing it for security	Security and data protection, then hardening, authentication and request signing
Looking a term up	Glossary
Getting help	Troubleshooting, contacting support, and the status page
Reading offline or on paper	Download the handbook (PDF): every page of these docs in one document

GET STARTED

How GraphPilot works

GraphPilot is a managed edge in front of one origin at a time: a plain HTTP API, a GraphQL API, or both on one host. Your clients send their requests to GraphPilot instead of to your server. GraphPilot answers what it can from its cache and forwards the rest to your server, called your **origin**.

GraphPilot runs on Fastly's edge network. Each request is handled at an edge location near the client that sent it; see edge locations.



A hit never reaches your origin. Your config and schema reach the edge with a deploy, not with a request.

Nothing changes in your client's code except the hostname it sends requests to. Your origin receives the client's request with a signature added and, for a GraphQL query, the key fields GraphPilot needs; see how a request flows.

One service, one origin

A **service** is one origin behind GraphPilot, with its own hostname, configuration and cache. What the origin speaks decides which features apply, not a setting on the service:

- **Plain HTTP.** A REST API, a JSON or HTML backend, anything that answers HTTP requests.
- **GraphQL.** A GraphQL server, a federation router or a gateway.
- **Both on one host.** GraphPilot treats a request as a GraphQL operation when its path is one of `request.graphql_paths` (`/graphql` by default) and its body, or for a `GET` its query string, holds an operation. Every other request on the same hostname is plain HTTP. A path below a GraphQL path counts as that path (`/graphql/health` is `/graphql`, `/graphql-admin` is not), so keep your REST routes outside it: a body there that is not a GraphQL operation is refused with `415` by default. The other way round, a GraphQL operation sent to any other path is refused with `404` rather than forwarded past the GraphQL checks (uploads and subscriptions excepted).

What GraphPilot does for every request

These apply to a request whatever it carries, GraphQL or not.

Cache what your origin allows

GraphPilot never decides on its own that a response may be reused. Your origin says so with a `Cache-Control` header (`s-maxage`, `max-age`, `stale-while-revalidate`), and a response that does not allow it is passed through and not stored. Data that differs per caller can still be cached, in entries partitioned per caller, per organization or on any value you derive from a verified token, a header or a cookie. Simultaneous identical misses are collapsed into one origin request.

Responses carry a `gp-cache` header that says what the cache did (`HIT`, `MISS`, `PASS` or `STALE`), and a response that was not stored says why in `gp-cache-reason`.

Read more: when a response is cached, cache statuses, identifying callers.

Tag and purge

Every cached response carries **surrogate keys**, labels that let you remove entries as a group. Your origin adds keys of its own with a `Surrogate-Key` response header, and you purge by key, by cache key, or everything, through the CLI, the portal or the GraphPilot API. A purge reaches every edge location.

Read more: invalidation, surrogate keys, the purge API.

Protect your origin

GraphPilot stands between the internet and your origin, so it can refuse requests before they cost your origin anything:

- **Authentication.** It verifies JWTs (and recognizes opaque API keys) at the edge, and can refuse requests whose credentials do not verify.
- **Request limits and filters.** It enforces a request body limit, and header and cookie filters drop what should not reach your origin, your clients or a cache entry.
- **Request signing.** It signs every request it forwards, so your origin can reject anything that did not come through GraphPilot.

It also applies your origin timeouts and retries, and records every request for Insights unless you choose to record fewer. A request GraphPilot refuses itself is answered with a stable reason code, in an `application/problem+json` body for a plain HTTP request.

Read more: authentication, request signing, origin timeouts, codes.

What GraphPilot adds for a GraphQL operation

> **GraphQL only**

When a request is a parsed GraphQL operation, GraphPilot knows what it asks for, and does more with it.

Cache by operation, with lifetimes from your schema

A query is keyed by its document, its operation name and its variables, so the same operation shares one entry whether it arrives as `POST` or `GET`. Mutations and subscriptions are never cached. Besides your origin's `Cache-Control`, your schema can set lifetimes and audiences with `@cacheControl`, including named scopes such as one entry per organization.

Read more: how long an entry lives, who shares an entry.

Invalidate from your schema and your mutations

GraphPilot reads your schema to find each type's key fields (by default, fields named `id`, `_id` or `key`, and fields of type `ID`), and tags every cached response with keys such as `Product:id:42` for the objects it contains. When a mutation returns an object, GraphPilot purges every cached response tagged with that object's key, whichever query produced it. You write no purge calls for this, and clients do not have to select the key fields: GraphPilot adds them to the request and removes them from the response.

Read more: automatic invalidation, selection set expansion.

Bound what a client may ask

GraphPilot limits query depth, document size and selection count, blocks introspection by default, and supports automatic persisted queries (APQ) and a registry of trusted documents, so you can refuse any operation your

clients did not ship.

Read more: hardening a public API, persisted queries.

The pieces you work with

Term	What it is
Organization	Your team's workspace. It holds services, members and API keys. You create one after signing up, and can belong to several.
Service	One origin behind GraphPilot, with its own hostname, configuration and cache. It has a name, your origin URL, a service hostname (`https://<subdomain>.<label>.graphpilot.cloud`), a signing key, and its deployments.
Config	`gpilot.toml` , the service's settings: caching, partitioning, authentication, limits, persisted queries and more. Every setting has a default, so an empty file is valid, and a deployment can also run without one.
Schema	Your GraphQL API's `schema.graphql` . GraphPilot reads it for types, key fields and caching directives. It does not validate operations against it: your origin stays the authority on what is a valid request.
Deployment	One version of your config and, for a GraphQL API, your schema, built and rolled out to the edge. `gpilot deploy` creates one; the portal lists them under **Deployments**. A failed build never goes live.

A plain HTTP service deploys with its config alone. Without a schema file, `gpilot deploy` deploys it as plain HTTP, with GraphQL features off.

You create a service once, in the portal or with `gpilot service create` . After that, the work is a loop: change the config or the schema, deploy, and watch the result in the response headers and the portal.

Where to go next

You want to	Read
Put GraphPilot in front of a GraphQL API	Quickstart: GraphQL
Put GraphPilot in front of a plain HTTP API	Quickstart: HTTP
See which features apply to plain HTTP and which to GraphQL	Features and limitations
Follow a request through every stage	How a request flows
Understand what a deployment changes for live traffic	Deployments
Look up a term	Glossary

GET STARTED

Features and limitations

Evaluating an edge is mostly a question of what it does **not** do, because that is what lands back on your team. This page is the inventory, with links to the detail. If a term is unfamiliar, the glossary defines the vocabulary these docs use.

The feature matrix

A request is **GraphQL** when GraphPilot parses a GraphQL operation from it: its path is one of `request.graphql_paths` (`/graphql` by default) and it carries an operation. Every other request is **plain HTTP**, and gets everything in the first column. See how GraphPilot works.

Caching

Feature	Plain HTTP	GraphQL
Caching from your origin's `Cache-Control`, including `s-maxage` and `stale-while-revalidate`	Yes	Yes
Lifetimes and audiences from the schema with `@cacheControl`	No	Yes
Cache key	Method, URI with query string, and request body	Query text, operation name and variables, whatever the method
Error responses (`4xx`, `5xx`) with a storable `Cache-Control`	Not stored unless you opt in	Not stored unless you opt in
Partitioning per caller or per group: `private`, vary rules, verified claims, headers, cookies, URL parameters	Yes	Yes
Named scopes, such as one entry per organization, declared on schema types	No, use a vary rule	Yes
Authenticated requests, following RFC 9111 §3.5	Yes	Yes
Your origin's `Vary`	Yes	Yes
Bypass for preview or editor traffic	Yes	Yes
Request collapsing	Yes	Yes
`gp-cache` status headers	Yes	Yes
Response body	Passed through byte for byte	Key fields GraphPilot added are removed

Invalidation

Feature	Plain HTTP	GraphQL
Tagging from your origin with a `Surrogate-Key` response header	Yes, on `GET` and `HEAD`	Yes, on queries
Purging from your origin with a `Surrogate-Key` header on a write's response	Yes, on `POST`, `PUT`, `PATCH` and `DELETE`	Yes, on mutations
Surrogate keys derived from the schema, with selection set expansion	No	Yes
Automatic purging of what a mutation returned	No	Yes
The purge API and `gpilot purge`, hard or soft: by key, by cache key, by request id, or everything	Yes	Yes, and by query hash
Purges that reach every edge location	Yes	Yes

Security

Feature	Plain HTTP	GraphQL
JWT verification and opaque API keys at the edge	Yes	Yes
Request signing and TLS to your origin	Yes	Yes
Published edge IP ranges for your firewall	Yes	Yes
Request body limit	Yes	Yes
Header and cookie filters	Yes	Yes
Client profiles	Identification, refusing unknown clients, minimum version, body limit, cache settings	Yes, plus depth, size, selection and persisted-query rules
Persisted queries and trusted documents	No	Yes
Limits on query depth, document size and selection count	No	Yes
Introspection blocked by default	No	Yes

Feature	Plain HTTP	GraphQL
GraphQL batches and multipart uploads	Not applicable	Refused unless you forward them unchecked with `request.forward_unchecked_bodies`

Operating it

Feature	Plain HTTP	GraphQL
What a deployment needs	`gpilot.toml` only	`gpilot.toml` and `schema.graphql`
Deploy findings	Config findings, without a schema	Schema and config findings
Origin timeouts	Yes	Yes
Retries, off until you enable `[origin.retry]`	`GET`, `HEAD` and `OPTIONS`; other methods only with `retry_unsafe_methods`	Queries; mutations only with `retry_mutations`
Compression toward the client	Yes	Yes
WebSocket upgrades	Yes	Yes
Per-request telemetry in Insights, on every plan	Yes, without the GraphQL fields	Yes
Codes on every refusal	Yes, as `application/problem+json`	Yes, as a GraphQL error
Custom domains with managed certificates, from Jet (Starter)	Yes	Yes
Redeploying an earlier deployment from the portal	Yes	Yes
Shielding, on paid plans, when you set `origin.shield`	`GET` and `HEAD` only	Queries, mutations and persisted-query hashes, not subscriptions
Lean mode when the Glider (Free) allowance is used up, a payment is overdue or GraphPilot pauses the account: no edge caching, no shielding, nothing recorded	Yes	Yes

What it deliberately leaves to you

These are not gaps. They are decisions, and each one is a decision about who is the authority.

Authorization. GraphPilot verifies *who* is calling. It does not decide *what* they may see, and on a cache hit your origin is not consulted at all. An entry keyed to one caller keeps being served to that caller until it expires or is

purged, so revoking access is two steps. Detail.

Whether a response may be cached. GraphPilot never decides on its own that something is reusable. Either your origin says so or, for a GraphQL API, your schema does. A service that says neither caches nothing, which is the safe default rather than an oversight.

Query validation. For a GraphQL operation, the edge parses and plans; it does not validate the operation against your schema. Your origin remains the authority on what is a valid request.

Verifying GraphPilot's signature. GraphPilot signs every forwarded request. Whether your origin checks it is yours to implement, and until it does, your origin is reachable by anyone who learns its address. This is the first item on the go-live checklist.

WebSocket traffic past the handshake. The WebSocket upgrade goes through the whole pipeline; the messages after it are a tunnel. Everything the edge guarantees stops at the upgrade, the query depth, document and introspection checks included. Detail.

Composing subgraphs. GraphPilot does not compose a federated graph. Put it in front of your federation router or gateway as one service, and it caches and protects the composed graph like any other GraphQL API. This works on every plan. Detail.

Current limitations

Not available	Consequence
`@cacheControl` cannot express a shared-cache lifetime	`maxAge` sets both how long GraphPilot keeps the entry and the `max-age` of any downstream header; a separate edge lifetime needs `s-maxage` from an origin header
No conditional revalidation	`no-cache` is refused rather than stored-and-revalidated, and a background refresh is a full request
Schema-declared invalidation on mutations	A mutation invalidates what it *returned*; to invalidate more, name a key on the response header
Field-level authorization from the schema	Authorization stays with your origin
Mutual TLS to the origin, or a per-service source address	An IP allowlist admits the whole edge network, not only your service
A staging-to-production promotion flow	A staging service is a second service you deploy to first, which needs a plan that includes more than one service (see pricing)
Rate limiting and per-caller quotas	No request cap and no per-second limit at the edge; see quotas for what plan limits do
Query cost analysis	Depth, document size and selection count are limited; per-field cost is not
Soft purging from the automatic path	Automatic invalidation is always a hard purge

Not available	Consequence
Single sign-on (SSO/SAML) and an audit log	Not available yet

> **Note**
> If something you need is on this list, contact support.

GET STARTED

Quickstart: GraphQL

GraphQL only For a REST or other plain HTTP origin, follow the HTTP quickstart instead.

This takes you from a new account to a verified cache `HIT` in front of your own GraphQL API, and then shows a mutation clearing that entry. Nothing here is throwaway: the service you create and the config you write are the ones you keep. If your origin is not a GraphQL API, follow the HTTP quickstart.

Before you start you need a GraphQL API that GraphPilot can connect to over the internet. It does not have to be public. The examples use a `products` query and an `updateProduct` mutation; substitute an operation from your own API.

1 Get an account and the CLI

Sign up at cockpit.graphpilot.io and create an organization with **New organization** if you do not belong to one yet; the dialog asks you to accept the Terms of Service. Then install the `gpilot` CLI and log in:

```
brew install graphpilot/tap/gpilot
gpilot auth login
```

Other platforms: installing the CLI. Several organizations or identities: signing in.

2 Create the service

A **service** is one origin behind GraphPilot, with its own hostname, configuration and cache.

> **Portal**
>
> In the portal, choose **New service** on your organization's page. The **Create a new service** dialog asks for a **Service name** (`Checkout API`), an **Origin URL** (`https://api.internal.example.com`), a **Subdomain** (`checkout-api`) and an **Environment**. See creating a service for what each field accepts. Keep **Deploy right away (no schema, default configuration)** ticked.

Select **Create service**. The next dialog shows the service's signing key, **once**. Copy it into your secret manager, then select **I have copied the key**.

The service's dashboard now shows **Setting up checkout-api**: GraphPilot deploys the service once as a plain pass-through, then looks for your GraphQL API behind it and shows what it found, such as the endpoint path and the number of types. **Apply suggested configuration and schema** turns GraphQL on with the schema it read. You can apply it, or select **Keep as plain HTTP**: step 5 deploys your own `schema.graphql` either way, because a schema read by introspection carries no caching directives. See set up a new service for every state of the setup.

Then select the service in the CLI:

```
gpilot service select checkout-api
```

CLI

```
gpilot service create \
  --name "Checkout API" \
  --url https://api.internal.example.com
gpilot service select checkout-api
```

The subdomain is derived from the name unless you pass `--subdomain`. The command prints the service's signing key **once**, in the **Signing key** row; copy it into your secret manager then. GraphPilot never shows it again. See the signing key.

The command asks `Deploy right away?`; answer yes (the default). It then follows the same setup as the portal and, when it finds your GraphQL API with a schema, asks whether to apply the suggested configuration and schema. Either answer works for this quickstart, since step 5 deploys your own `schema.graphql`. `--deploy` skips the first question, `--no-deploy` only creates the service; see `gpilot service create`.

Once the first deployment is live, your service is reachable at `https://checkout-api.k7m2qp.graphpilot.cloud` (`k7m2qp` stands for your organization's hostname label, which the portal shows). That takes one to two minutes, and for your organization's first service a few minutes more, up to an hour, while its certificate is issued; the setup shows both. Until you deploy a schema, it is a plain pass-through: every request reaches your origin unchanged, and only responses your origin marks with `Cache-Control` are cached.

GraphPilot forwards each request to your origin's host with the path the client sent, so if your API answers on `/graphql`, your clients call `https://checkout-api.k7m2qp.graphpilot.cloud/graphql`. How the origin URL is used (scheme, port, TLS) is on origin.

3 Put your schema next to a config

In the directory you will deploy from, you need two files.

`schema.graphql` is your API's schema, exactly as your server exposes it. GraphPilot reads it for your types, the key fields it derives surrogate keys from, and any caching directives. If you have no schema file, read one from your running API with `gpilot schema pull`, then add the directives from step 4 to it: introspection returns your types but no applied directives such as `@cacheControl`.

`gpilot.toml` is the service configuration. Every setting has a default, so an empty file is valid. GraphPilot treats requests to `/graphql` as GraphQL. If your API answers on a different path, say so: until you do, a GraphQL operation sent to that path is refused with `404` and `NOT_A_GRAPHQL_PATH`, because the GraphQL checks run only on the paths listed here:

gpilot.toml

```
[request]
graphql_paths = ["/api/graphql"]
```

4 Make something cacheable

GraphPilot does not decide on its own that a response may be reused. Your origin or your schema has to say so. Pick either; you can use both later.

From your origin

Send a `Cache-Control` header on the responses that may be cached:

```
Cache-Control: s-maxage=60
```

A header from your origin decides before your schema is consulted. How the two combine, and why `s-maxage` rather than `max-age`, is on when a response is cached.

> **⚠ Apollo Server sends no-store by default**
>
> Apollo Server's cache-control plugin is on by default and sends `Cache-Control: no-store` on every response it does not consider cacheable. GraphPilot never stores those. Annotate your schema so Apollo Server computes a lifetime, or switch its header off: see making Apollo Server responses cacheable.

From your schema

Annotate the schema, the way Apollo's cache control plugin does. Declare the directive and use it:

schema.graphql

```
directive @cacheControl(
```

```
    maxAge: Int
    scope: CacheControlScope
    inheritMaxAge: Boolean
) on OBJECT | INTERFACE | UNION | FIELD_DEFINITION

directive @vary(names: [String!]!) on ENUM_VALUE

enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION @vary(names: ["x-org"])
}

type Product {
    id: ID!
    name: String!
}

type Query {
    products: [Product!]! @cacheControl(maxAge: 60)
}
```

`products` is the same for everyone, so it keeps the default scope, `PUBLIC`. `ORGANIZATION` is a named scope, one entry per organization, for data that differs per customer. It needs a `[[cache.policy.vary]]` entry named `x-org` before you use it; see named scopes. `PRIVATE` is for what really differs per person.

This path decides when your origin sends no `Cache-Control` header, or one that states neither storage nor freshness. See lifetime from the schema.

5 Deploy

```
gpilot deploy
```

This picks up `./schema.graphql` and `./gpilot.toml`, shows what changed since the last deployment, and asks before uploading anything. It then waits for the deployment to go live, printing each step and any findings the build reported. A failed build never goes live. Flags for CI, other paths and not waiting are in the command reference.

6 Verify the cache

Send the same query twice to your service hostname:

```
export GRAPHPILOT_HOST=checkout-api.k7m2qp.graphpilot.cloud
```

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"{ products { id name } }"}' | grep -i '^gp-'
```

Use your API's GraphQL path in place of `/graphql` if you set `request.graphql_paths` in step 3.

The first response is a `MISS`: it came from your origin and was stored. The second is a `HIT`:

```
gp-cache: HIT
gp-cache-hits: 1
gp-cache-age: 4
gp-cache-max-age: 60
gp-cache-remaining: 56
```

If you see `PASS`, read the next section.

7 See a write clear the cache

Send a mutation through GraphPilot that changes one of the products in that list and returns it:

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"mutation { updateProduct(id: \"1\", name: \"Blue shirt\") { id name } }"}'
```

Repeat the query from step 6. The first response is a `MISS` again, with the new name from your origin, and the one after it is a `HIT`.

Nothing was configured for this. The cached `products` response was tagged with a surrogate key for each product it contained, such as `Product:id:1`. The mutation's response contained product 1, so GraphPilot purged every cached response tagged `Product:id:1`. A mutation purges only what it returns: see automatic invalidation for what is and is not purged.

If you see PASS

`gp-cache: PASS` means the response was not stored, and `gp-cache-reason` names the rule that refused. These three cause most first-deployment misses:

- **Your origin sets a cookie** (`CACHE_SKIPPED_SET_COOKIE`). A response that sets a session cookie is never stored. Frameworks that touch the session on every request do this by default. See the refusals.
- **Apollo Server sent** `no-store` (`CACHE_SKIPPED_NO_STORE`). It does so for every response its hints do not make cacheable. See Apollo Server.
- **Your hosting platform added** `Cache-Control: private`. Some platforms, such as Google Cloud Run with session affinity, add it to every response that has no `Cache-Control` of its own. GraphPilot then stores the

response per caller, or not at all. See when your hosting platform adds `Cache-Control: private`.

Every other reason is in why is nothing cached?, and a second `MISS` where you expected a `HIT` in troubleshooting.

Point your clients at it

Change your client's endpoint to the service hostname, keeping the path your API uses. Nothing else changes: GraphPilot speaks the same GraphQL over the same transports, and what reaches your origin is an ordinary GraphQL request.

Apollo Client

```
import { ApolloClient, HttpLink, InMemoryCache } from "@apollo/client";

const client = new ApolloClient({
  link: new HttpLink({ uri: "https://checkout-api.k7m2qp.graphpilot.cloud/graphql" }),
  cache: new InMemoryCache(),
});
```

urql

```
import { Client, cacheExchange, fetchExchange } from "urql";

const client = new Client({
  url: "https://checkout-api.k7m2qp.graphpilot.cloud/graphql",
  exchanges: [cacheExchange, fetchExchange],
});
```

Relay

```
import { Environment, Network, RecordSource, Store } from "relay-runtime";

const network = Network.create(async (operation, variables) => {
  const response = await fetch("https://checkout-api.k7m2qp.graphpilot.cloud/graphql", {
    method: "POST",
    headers: { "content-type": "application/json" },
    body: JSON.stringify({ query: operation.text, variables }),
  });
  return response.json();
});

const environment = new Environment({ network, store: new Store(new RecordSource()) });
```

fetch

```
const response = await fetch("https://checkout-api.k7m2qp.graphpilot.cloud/graphql",
{
  method: "POST",
  headers: { "content-type": "application/json" },
  body: JSON.stringify({ query: "{ products { id name } }" }),
});
const { data } = await response.json();
```

Before you consider this finished, **verify the request signature at your origin**, so it only trusts traffic that came through GraphPilot, and keep your origin's clock in sync, because the signature is time-bound. Both are on the go-live checklist.

Where to go next

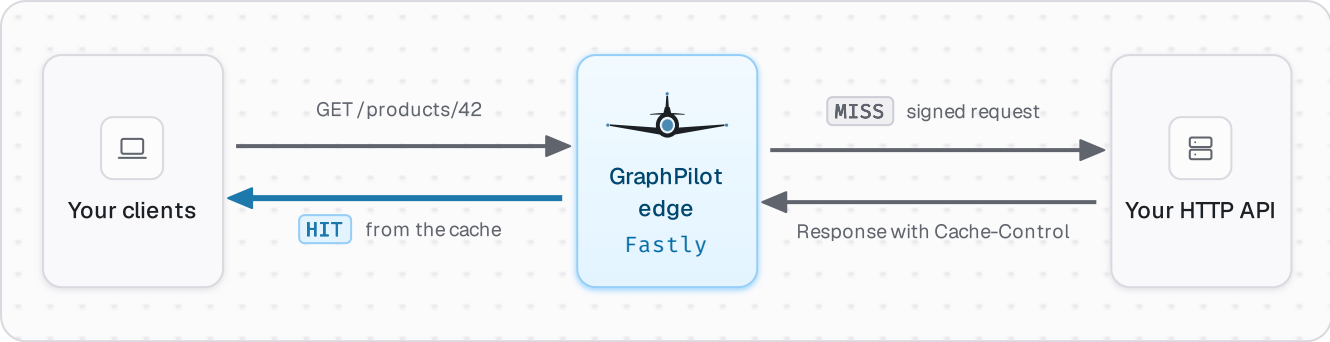
You want to	Read
Understand exactly what gets cached and what does not	When a response is cached
Decide who shares a cached answer	Who shares an entry
Know what a mutation purges, and what it does not	Automatic invalidation
Clear something by hand	The purge API
Stop clients sending arbitrary queries	Persisted queries
Deploy from a pipeline	Deploy from CI
See every setting	Configuration reference
Serve it on your own hostname	Custom domains

GET STARTED

Quickstart: HTTP

This takes you from a new account to a verified cache `HIT` in front of your own HTTP API, and then shows a purge clearing that entry. Nothing here is throwaway: the service you create and the config you write are the ones you keep. If your origin is a GraphQL API, follow the GraphQL quickstart instead; it adds caching and invalidation from your schema.

Before you start you need an HTTP API that GraphPilot can connect to over the internet, and the ability to change the response headers it sends. It does not have to be public. The examples use `GET /products/42` and `PUT /products/42`; substitute an endpoint from your own API.



Your origin's Cache-Control decides what is stored. A hit never reaches your origin.

1 Get an account and the CLI

Sign up at cockpit.graphpilot.io and create an organization with **New organization** if you do not belong to one yet; the dialog asks you to accept the Terms of Service. Then install the `gpilot` CLI and log in:

```
brew install graphpilot/tap/gpilot
gpilot auth login
```

Other platforms: installing the CLI. Several organizations or identities: signing in.

2 Create the service

A **service** is one origin behind GraphPilot, with its own hostname, configuration and cache.

> **Portal**
>
> In the portal, choose **New service** on your organization's page. The **Create a new service** dialog asks for a **Service name** (`Catalog API`), an **Origin URL** (`https://api.internal.example.com`, the URL of your HTTP API), a **Subdomain** (`catalog-api`) and an **Environment**. See creating a service for what each field accepts. Keep **Deploy right away (no schema, default configuration)** ticked.
>
> Select **Create service**. The next dialog shows the service's signing key, **once**. Copy it into your secret manager, then select **I have copied the key**.
>
> The service's dashboard now shows **Setting up catalog-api**: GraphPilot deploys the service once as a plain pass-through with every setting at its default, then checks for a GraphQL API behind it. For a REST or plain HTTP origin it finds none and says **No GraphQL endpoint detected, the service runs as plain HTTP**; select **Keep as plain HTTP** to finish. See set up a new service.
>
> Then select the service in the CLI:
>
> ```
> gpilot service select catalog-api
> ```

CLI

```
gpilot service create \
  --name "Catalog API" \
  --url https://api.internal.example.com
gpilot service select catalog-api
```

The subdomain is derived from the name unless you pass `--subdomain`. The command prints the service's signing key **once**, in the **Signing key** row; copy it into your secret manager then. GraphPilot never shows it again. See the signing key.

The command asks `Deploy right away?`; answer yes (the default). It then follows the same setup as the portal, finds no GraphQL behind a REST or plain HTTP origin, and asks `Keep the service as plain HTTP?`: answer yes. `--deploy --keep-plain-http` answers both up front; see `gpilot service create`.

Once the first deployment is live, your service is reachable at `https://catalog-api.k7m2qp.graphpilot.cloud` (`k7m2qp` stands for your organization's hostname label, which the portal shows). That takes one to two minutes, and for your organization's first service a few minutes more, up to an hour, while its certificate is issued; the setup shows both. GraphPilot forwards each request to your origin's host with the path and query string the client sent, so `https://catalog-api.k7m2qp.graphpilot.cloud/products/42` reaches `/products/42` on your origin. How the origin URL is used (scheme, port, TLS) is on origin.

3 Write a config

In the directory you will deploy from, create `gpilot.toml`. Every setting has a default, so an empty file is valid, and a plain HTTP origin needs nothing else to start:

gpilot.toml

```
# Nothing here is required. See /reference/configuration for every setting.
```

4 Send Cache-Control and a surrogate key from your origin

GraphPilot does not decide on its own that a response may be reused. Your origin says so. Add two headers to the response for `GET /products/42`:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: s-maxage=60
Surrogate-Key: product-42 products
```

`Cache-Control: s-maxage=60` lets GraphPilot keep the response for 60 seconds and share it between callers; see when a response is cached for every directive it reads. `Surrogate-Key` tags the entry with keys

you choose, so you can purge it by name later. GraphPilot removes that header before the response reaches the client. The format is on the Surrogate-Key header.

Each entry is keyed by the request method and the path, query string included, so `/products/42?lang=de` is a separate entry. The hostname is not part of the key.

5 Deploy

Deploy the config on its own. A plain HTTP service needs no schema. The setup already runs the service on the defaults, so nothing changes in behavior while the file is empty, but from here on the service runs on a configuration you keep in your repository, and the dashboard's checklist counts this deployment as your own:

```
gpilot deploy
```

This picks up `./gpilot.toml`, shows what changed since the last deployment, and asks before uploading anything. With no `schema.graphql` or `schema.gql` in the directory, it says so before it asks:

```
No schema found: deploying as a plain HTTP service, GraphQL features are off.
```

That is the expected line for this origin, not an error. It then waits for the deployment to go live, printing each step and any findings the build reported, including a notice that the service has no schema. A failed build never goes live. Flags for CI, other paths and not waiting are in the command reference.

If the same origin also serves a GraphQL API, deploy it with that schema instead, by following the GraphQL quickstart: requests to its other paths are plain HTTP and get everything on this page.

6 Verify the cache

Send the same request twice to your service hostname:

```
export GRAPHPILOT_HOST=catalog-api.k7m2qp.graphpilot.cloud

curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/products/42" | grep -i '^gp-'
```

The first response is a `MISS`: it came from your origin and was stored. The second is a `HIT`:

```
gp-cache: HIT
gp-cache-hits: 1
gp-cache-age: 4
gp-cache-max-age: 60
gp-cache-remaining: 56
```

If you see `PASS`, read the next section.

7 Purge by key

Clear every response tagged `product-42`:

```
gpilot purge product-42
```

The command shows what it is about to purge and asks you to confirm. Repeat the request from step 6: the first response is a `MISS` again, and the one after it is a `HIT`. A purge reaches every edge location. To clear one response without a key of your own, purge its cache key, `ck:GET:/products/42`, the method and the path without scheme or hostname. See the purge API for every way to purge.

Your origin can also purge without any call. When the response to a write carries `Surrogate-Key`, GraphPilot purges the keys it names instead of tagging:

```
HTTP/1.1 200 OK
Surrogate-Key: product-42
```

Send that on your origin's response to `PUT /products/42` (or any `POST`, `PUT`, `PATCH` or `DELETE`) through GraphPilot, and the next `GET /products/42` is a `MISS`. See the Surrogate-Key header.

If you see PASS

`gp-cache: PASS` means the response was not stored, and `gp-cache-reason` names the rule that refused. These three cause most first-deployment misses:

- **Your origin sets a cookie** (`CACHE_SKIPPED_SET_COOKIE`). A response that sets a session cookie is never stored. Frameworks that touch the session on every request do this by default. See the refusals.
- **The response has no lifetime** (`CACHE_SKIPPED_NO_CACHE_POLICY` or `CACHE_SKIPPED_NO_MAX_AGE`). Check that your origin's response carries `s-maxage` or `max-age`, and that nothing between your application and GraphPilot replaced the header.
- **The request carried a credential** (`CACHE_SKIPPED_CREDENTIALED_REQUEST`). A request with an `Authorization` header, a cookie that is not on your cookie allow list, or the header an auth provider reads is only shared when the response says so with `public` or `s-maxage`. For answers that differ per caller, see identifying callers and authenticated requests.

Every other reason is in why is nothing cached?, and a second `MISS` where you expected a `HIT` in troubleshooting.

Point your clients at it

Change your clients' base URL to the service hostname, keeping your paths. Nothing else changes: what reaches your origin is the same request, with a signature added and `Accept-Encoding: identity` (GraphPilot compresses the response for the client itself).

```
const response = await fetch("https://catalog-
api.k7m2qp.graphpilot.cloud/products/42");
const product = await response.json();
```

Before you consider this finished, **verify the request signature at your origin**, so it only trusts traffic that came through GraphPilot, and keep your origin's clock in sync, because the signature is time-bound. Both are on the go-live checklist.

Where to go next

You want to	Read
Understand exactly what gets cached and what does not	When a response is cached
Tag, partition and purge a whole REST API	Caching a REST API
Cache answers that differ per caller or per organization	Who shares an entry
See which features apply to plain HTTP	Features and limitations
Clear something by hand or from a job	The purge API
Deploy from a pipeline	Deploy from CI
See every setting	Configuration reference
Serve it on your own hostname	Custom domains

Connect your origin

CONNECT YOUR ORIGIN

Connection to the origin

Every request the cache cannot answer goes on to your origin. This page covers that connection: scheme and port, what GraphPilot asks your origin to send, how long it waits, and when it tries again. Whether your origin can trust what arrives is on request signing and the origin allowlist.

TLS and ports

GraphPilot speaks TLS to your origin when your origin URL is `https://`, and does not when it is `http://`. There is no separate switch: the scheme you register the service with is the scheme GraphPilot uses.

On an `https://` origin the connection is verified: the certificate is checked against your origin's hostname, and SNI is sent with that hostname. A certificate that does not match is a failed connection, not a warning.

An `http://` origin is not encrypted, and the signature does not compensate: it proves who sent the request, not that nobody read it. If your origin is reachable over the public internet, register it as `https://`.

> **Scheme, host and port only**
>
> GraphPilot uses only the scheme, host and port of your origin URL, and refuses a URL that carries a path, query or fragment (`ORIGIN_URL_PATH_NOT_ALLOWED`): the path forwarded is the one the client called. When the URL names no port, GraphPilot connects to port `443` for `https://` and port `80` for `http://`. Name any other port explicitly, for example `http://origin.example.com:8080`. A service saved with a path before this rule keeps deploying, with the path ignored.

The origin URL belongs to the service, not to `gpilot.toml`. Change it in the portal or with `gpilot service edit`; it takes effect with the next deployment. There is **no mutual TLS** to the origin today.

Your origin receives its own hostname in `Host`, the host of the origin URL, not the hostname the client called (a custom domain or the service's GraphPilot hostname). An origin that routes by `Host`, such as a virtual host or a platform that serves several apps from one address, has to answer on the hostname in the origin URL.

What else your origin receives

These options under `[origin]` change the request on its way to your origin. Each is off or neutral by default:

Option	What it does
`origin.method_rewrite`	Rewrites `QUERY` to `POST`, or `POST` to `QUERY` for query operations.
`[origin.headers]` and `[origin.cookies]`	Header and cookie names removed before the request reaches your origin.

Option	What it does
`origin.request_id_header`	Sends the GraphPilot request id under a header name you choose.
`origin.forward_client_ip`	Sends the client's IP address in `gp-client-ip`.

The full list of request headers your origin sees, and which ones GraphPilot adds or removes, is on request headers your origin receives.

Compression

GraphPilot sends your origin `Accept-Encoding: identity`, asking for an uncompressed body. It has to read the body to derive surrogate keys, to remove the fields it injected and to decide whether to cache it, and none of that works on compressed bytes. Compression toward the client is added on the way out, negotiated against the client's own `Accept-Encoding`, so your clients still receive compressed responses.

An origin that compresses anyway still works, at a cost: a body encoded with `gzip`, `deflate`, `br` or `zstd` is decoded first, logged as `ORIGIN_RESPONSE_DECODED` with both sizes. A body that cannot be decoded is answered with `502 Bad Gateway`:

Code	When
`ORIGIN_RESPONSE_ENCODING_UNSUPPORTED`	The `Content-Encoding` names a coding GraphPilot cannot decode.
`ORIGIN_RESPONSE_ENCODING_CORRUPT`	The body did not decode under the coding its `Content-Encoding` named.
`ORIGIN_RESPONSE_DECODED_TOO_LARGE`	The body would decode to more than GraphPilot processes.

The fix for all three is to make your origin honor `Accept-Encoding`.

A plain HTTP response body reaches your client byte-identical to what your origin sent, after any decoding above. GraphPilot does not rewrite it, so a signature, a digest or an ETag your origin computed over the body stays valid.

Timeouts

Three optional keys under `[origin]` in `gpilot.toml` bound how long the edge waits for your origin:

Key	What it bounds	Default	Ceiling
`connect_timeout_ms`	Accepting the TCP connection and completing the TLS handshake	1000	5000 on every plan
`first_byte_timeout_ms`	Waiting for the first byte of the response, after the request was sent	15000, or your plan's ceiling if that is lower	Per plan
`between_bytes_timeout_ms`	Waiting between two bytes of a response that has already started	10000	60000 on every plan

They take effect with your next deployment. An origin that does not connect or send its first byte in time fails the request with `502` and `BAD_GATEWAY`, unless a retry succeeds.

What your plan allows

Plan	`first_byte_timeout_ms` ceiling
Glider (Free)	5000
Jet (Starter)	15000
Airliner (Business)	30000
Jumbo (Enterprise)	30000

A ceiling is not a recommendation: failing in two seconds is often better than failing in thirty.

On Glider (Free), the default is lower. An unconfigured service gets its plan's ceiling where that is below 15000, so the edge waits **5 seconds** for the first byte. A container scaling up from zero can take longer, and the request fails with a `502` while it boots. Keep an instance warm, or move to a plan whose ceiling covers your cold start.

A value out of bounds is refused

Nothing is quietly adjusted. The deployment is refused, with a message naming the value, when:

- **a value exceeds its ceiling.** The message names the ceiling and the plan:

  ```
  origin.first_byte_timeout_ms = 30000 exceeds the maximum of 5000 ms on the FREE
  plan.
  Lower the value or upgrade the plan.
  ```

  After a downgrade, your next deployment is refused until the value comes down.

- **a value is below 100 ms.** The values are milliseconds, and a smaller one is almost certainly seconds written in the wrong unit:

  ```
  origin.first_byte_timeout_ms = 30 is below the minimum of 100 ms. The value is
  milliseconds;
  30 would fail every request to a healthy origin.
  ```

Timeouts and retries

A request may spend at most **120 seconds** at the edge: the wall clock Fastly's edge network, which GraphPilot runs on, allows per request. With retries, every attempt gets the full connect and first-byte timeouts, and the delays between attempts add up, so the worst case has to fit:

```
(max_retries + 1) × (connect_timeout_ms + first_byte_timeout_ms) + backoff delays ≤
120000 ms
```

A configuration that does not fit is refused by the deployment build, with a message naming the worst case and asking you to lower `origin.first_byte_timeout_ms` or `origin.retry.max_retries`. `between_bytes_timeout_ms` is not counted, because how long a body takes depends on its size. This is also why no plan allows a first-byte timeout above 30000: a retry could never run.

The edge also allows each request at most 32 backend requests, and every attempt is one of them. Keep `max_retries` at 30 or below: the deployment build checks the time budget only, so a higher value is accepted, and the attempts past the limit cannot be sent.

Fitting the budget is not fitting your client's patience: two attempts at 30 seconds are a minute of waiting. Pick the numbers together.

Retries

A request that fails before it reaches your application can be sent again without the client learning that it failed. GraphPilot does this only when you ask it to.

> **Note**
>
> Retries are **off by default**. Retrying changes what your origin sees, and a service that never asked for it must not start receiving a failed request twice because GraphPilot was upgraded.

The keys, under `[origin.retry]`:

Key	Type	Default	What it does
`enabled`	boolean	`false`	Whether failed origin requests are retried at all.
`max_retries`	integer	`1`	How many **additional** attempts a failed request gets. `1` means at most two attempts in total.
`status_codes`	list of integers	`[502, 503, 504]`	The origin response statuses that trigger a retry.
`on_network_error`	boolean	`true`	Whether a transport failure is retried.
`retry_mutations`	boolean	`false`	Whether GraphQL mutations are retried. Does not apply to plain HTTP requests.
`retry_unsafe_methods`	boolean	`false`	Whether plain HTTP requests with a method other than `GET`, `HEAD` or `OPTIONS` are retried. Does not apply to GraphQL requests.
`backoff`	enum	`"fixed"`	How the delay between attempts is computed.
`delay_ms`	integer	`1000`	The base delay in milliseconds.

Key	Type	Default	What it does
`max_delay_ms`	integer	`2000`	The ceiling on any single delay, for both `fixed` and `exponential`.

What counts as a failure

A retryable status. The default list is the three gateway statuses that mean the request did not reach your application. A `500` is deliberately not in it: a `500` is your application answering, and answering twice is rarely what a client wants. If your origin uses `5xx` semantically, add to the list rather than assuming.

A transport failure, covered by `on_network_error`. The origin was unreachable, the connection dropped, or a timeout expired. There is no status to match on, because nothing came back at all.

Which requests are repeated

GraphQL mutations are not retried by default. Enabling `[origin.retry]` repeats queries; a mutation is sent once unless you set `retry_mutations = true`. The retryable statuses usually mean the request never reached your application, so the mutation never ran, but that holds only as far as your origin's status codes do. A proxy in front of your application can time out with a `504` while the application is still committing, and the write would then run twice.

A timeout says nothing about whether your origin already acted. With `on_network_error` on (the default once retries are enabled), a transport failure is retried, a timeout included. With `retry_mutations = true`, that covers mutations too, so a mutation that timed out after your origin executed it is sent again. **Set `retry_mutations = true` only if your origin answers a gateway status before it executes anything, or your mutations are idempotent**, for example by an idempotency key.

A plain HTTP request is retried only when its method is `GET`, `HEAD` or `OPTIONS`. A `POST`, `PUT`, `PATCH`, `DELETE` or any other method is sent once, because repeating a write your origin may already have executed would execute it twice. Set `retry_unsafe_methods = true` only for an origin that makes its writes idempotent, for example by an idempotency key. The two switches do not overlap: `retry_mutations` never decides a plain HTTP request, and `retry_unsafe_methods` never decides a GraphQL one.

A WebSocket handshake is never retried, whatever the configuration says. The connection is handed to your origin on the first attempt, so there is no second attempt to make; a handshake that fails reaches the client as a `502`. See subscriptions and WebSockets.

Backoff

Strategy	Behavior	When to use it
`none`	Retry immediately.	Your origin is a pool and a failure means "this instance is unhealthy". The next attempt lands elsewhere, and waiting only adds latency.

Strategy	Behavior	When to use it
`fixed` (default)	Wait `delay_ms` before every retry, capped at `max_delay_ms`.	The general case.
`exponential`	Wait `delay_ms`, then double before each further attempt, capped at `max_delay_ms`.	The failure is more likely load than one bad instance, so attempts spread out instead of arriving together.

A delay is spent suspended, so it costs wall clock, not CPU, and counts against the 120-second budget.

Observing them

A response whose origin request was repeated carries the count, and the header is absent when there were none:

```
gp-origin-retries: 2
```

The same event is logged as `ORIGIN_RETRIED`, once per repeated attempt, which is what to alert on: retries succeeding quietly still mean your origin is failing.

A retry happens below the cache: the response that eventually succeeds goes through the normal store decision, and a cache hit never retries. A background revalidation is an ordinary origin request and retries like any other.

Shielding

With shielding, a miss at an edge location goes to the shield location first, and only the shield connects to your origin. The shield's request to your origin uses the same TLS, signing and timeouts as any other.

The extra hop counts against the 120-second budget. The edge waits for the shield's whole origin sequence, every retry attempt and the gaps between them, plus a fixed 5-second hop buffer, rather than a single attempt's timeout: the shield may still be legitimately retrying when a single attempt's timeout would have expired. An unset `first_byte_timeout_ms` counts as 15000 ms here. A configuration whose total would exceed the 120-second budget is refused by the deployment build, naming `origin.shield`.

Retries do not double at your origin. With a shield, your origin sees no more attempts per request than your `[origin.retry]` policy allows, the same as without one.

When the shield cannot be used, a read falls back to your origin and a mutation is refused; how each case is answered is on when the shield cannot be used, and how a shielded request counts for usage on request units.

Related

- Request signing: proving to your origin that a request came through GraphPilot.
- Origin allowlist: the addresses GraphPilot connects from.
- Shielding

- Quotas and lean mode
- Configuration reference
- Troubleshooting

CONNECT YOUR ORIGIN

Apollo Server

This page takes an Apollo Server 4 or 5 origin running on Express through four steps: export the schema GraphPilot deploys, make responses cacheable, check that requests really came through GraphPilot, and purge when data changes. Each step stands on its own, so you can stop after the first two and add the rest later.

The examples use the Express integration, `@as-integrations/express5` (or `@as-integrations/express4` if you are still on Express 4; the API is the same). On Apollo Server 4 you may still import `expressMiddleware` from `@apollo/server/express4`.

Mount Apollo Server on the path in `request.graphql_paths` (`/graphql` by default) and nowhere else. GraphPilot reads a request as GraphQL on that path, on paths below it and in any letter case, the way Express routes `app.use("/graphql", ...)`, and forwards every other path as plain HTTP. `startStandaloneServer` answers GraphQL on every path. GraphPilot refuses an operation sent to another path with `404` and `NOT_A_GRAPHQL_PATH`, but it does not recognize an upload or a subscription there, so use an integration that mounts Apollo Server on one path, such as the Express one below.

1. Export the schema

A deployment is built from a `schema.graphql` file next to your `gpilot.toml` (see deployments). It has to be the schema your server runs, **with its directives left in**: `@cacheControl` and `@surrogateKey` are read from that file, not from your server.

`printSchema` from the `graphql` package drops every directive you applied, so use `printSchemaWithDirectives` from `@graphql-tools/utils` instead:

scripts/export-schema.ts

```
import { writeFileSync } from "node:fs"
import { makeExecutableSchema } from "@graphql-tools/schema"
import { printSchemaWithDirectives } from "@graphql-tools/utils"
import { typeDefs } from "../src/schema"

// Resolvers are not needed to print the schema.
const schema = makeExecutableSchema({ typeDefs })
writeFileSync("schema.graphql", printSchemaWithDirectives(schema))
```

Run it before every deployment, for example as a step in your CI workflow. If your type definitions already live in `.graphql` files, concatenating them into `schema.graphql` works just as well.

Deploy the new schema to GraphPilot before you release the server that adds fields to it. A query selecting a field the deployed schema does not declare is answered but never cached (`CACHE_SKIPPED_UNRESOLVED_SELECTION`); see deploy the schema before the origin.

2. Make responses cacheable

Apollo Server installs its cache-control plugin in every server by default. The plugin computes a policy from your `@cacheControl` hints (and from hints your resolvers set at runtime) and sends it as a response header: `Cache-Control: max-age=N, public` or `max-age=N, private` when the response is cacheable, and `Cache-Control: no-store` when it is not.

GraphPilot follows an origin `Cache-Control` header before it looks at the schema (see how long an entry lives). So you choose who decides the freshness lifetime (TTL): Apollo Server, or GraphPilot reading your schema.

Option A: keep Apollo Server's header (the default)

Change nothing. Declare the directive as Apollo Server asks you to, annotate your types and fields, and GraphPilot caches each response for as long as the header says. A `private` header caches one entry per caller once your service defines a `private` partition key; without one, private responses are not cached. `no-store` is never cached.

schema.graphql

```
enum CacheControlScope {
  PUBLIC
  PRIVATE
}

directive @cacheControl(
  maxAge: Int
  scope: CacheControlScope
  inheritMaxAge: Boolean
) on FIELD_DEFINITION | OBJECT | INTERFACE | UNION

type Query {
  products: [Product!]! @cacheControl(maxAge: 300)
  me: User @cacheControl(maxAge: 60, scope: PRIVATE)
}
```

This is the right choice if your resolvers set hints at runtime (`info.cacheControl.setCacheHint`). Those hints only exist inside Apollo Server, and the header is the only way they reach GraphPilot.

Option B: let GraphPilot read the schema

Turn the header off, and GraphPilot resolves the same `@cacheControl` hints from `schema.graphql` itself:

src/server.ts

```
import { ApolloServer } from "@apollo/server"
import { ApolloServerPluginCacheControl } from "@apollo/server/plugin/cacheControl"

const server = new ApolloServer({
  typeDefs,
  resolvers,
  plugins: [ApolloServerPluginCacheControl({ calculateHttpHeaders: false })],
})
```

What you gain: a stale window per field with `swr`, and named scopes for data shared within an organization but not beyond it. What changes compared with Apollo Server:

- **Runtime hints are invisible.** Anything a resolver sets with `setCacheHint` never leaves your server. If you rely on it, stay with option A.
- **Add `swr` to your directive definition** if you use it (`swr: Int`), so the schema stays valid for Apollo Server as well.

Do not use `calculateHttpHeaders: "if-cacheable"` either. It leaves the header off exactly the responses Apollo Server found uncacheable, and GraphPilot then falls back to the schema for them, which can cache a response Apollo Server would not have.

3. Verify the request signature

GraphPilot signs every request it forwards with your service's signing key (see request signing). Checking that signature lets your origin refuse anything that did not come through GraphPilot.

You need the signing key. The portal and `gpilot service create` each show it once, when the service is created, and neither shows it again. If you did not keep it, rotate it in the portal to get a new one. Store it as `GRAPHPILOT_SIGNING_KEY`.

Copy the Node.js `verifyGraphPilotSignature` function from verify the signature on your origin into `src/graphpilot-signature.ts`. The signature covers the **raw** request body, so capture the bytes while `express.json()` parses them, and verify before Apollo Server runs:

src/server.ts

```
import type { IncomingMessage } from "node:http"
import cors from "cors"
import express from "express"
import { expressMiddleware } from "@as-integrations/express5"
import { verifyGraphPilotSignature } from "./graphpilot-signature"

type WithRawBody = IncomingMessage & { rawBody?: Buffer }

// During a key rotation, add the pending key here as a second entry.
const SIGNING_KEYS = [process.env.GRAPHPILOT_SIGNING_KEY!]
```

```
const app = express()
await server.start() // your ApolloServer instance

app.use(
  "/graphql",
  cors(),
  express.json({ verify: (req, _res, buf) => { (req as WithRawBody).rawBody = buf } }),
  (req, res, next) => {
    const valid = SIGNING_KEYS.some((signingKey) =>
      verifyGraphPilotSignature({
        method: req.method,
        pathAndQuery: req.originalUrl,
        rawBody: (req as WithRawBody).rawBody ?? Buffer.alloc(0),
        headers: req.headers,
        signingKey,
      }),
    )
    if (valid) next()
    else res.status(401).end()
  },
  expressMiddleware(server),
)
```

A request with no body (a `GET`) hashes the empty string, which is what `Buffer.alloc(0)` gives you. `express.json()` only reads `application/json` bodies; if your clients send another content type, widen it with `type`, or a signed request with such a body fails verification. `req.originalUrl` is the path and query string as GraphPilot sent them; if a load balancer in front of your app rewrites the path, verify against the path GraphPilot used. The known test vector checks your setup byte for byte.

4. Purge when data changes

A mutation that returns the changed object purges it for you. GraphPilot tags the mutation response with the keys of the objects in it and purges every cached response that contained them (see automatic invalidation). So let your mutations return the object they changed, not a `Boolean`:

```
type Mutation {
  updateProduct(id: ID!, input: ProductInput!): Product!   # purges Product:id:<id>
  deleteProduct(id: ID!): Boolean!                          # purges nothing
}
```

The client does not have to select `id` for this to work: GraphPilot adds the key fields to the query and removes them again before the response reaches the client.

When data changes **outside GraphQL** (an admin tool, an import job, a queue consumer), purge the keys from that code with `gpilot purge` or the `purgeBySurrogateKey` mutation, for example `Product:id:<id>`. The API key

needs the `cache:purge` permission. See purging through the API and CLI for the call, its payload and where the service id comes from.

Related

- How long an entry lives
- Request signing
- Automatic invalidation
- Deploy from CI

CONNECT YOUR ORIGIN

GraphQL Yoga

This page takes a GraphQL Yoga 5 origin through four steps: export the schema GraphPilot deploys, make responses cacheable, check that requests really came through GraphPilot, and purge when data changes. Each step stands on its own.

> **You do not need Yoga's response cache**
>
> GraphPilot is the cache, in front of your origin. Yoga's response cache plugin (`useResponseCache`) would add a second cache behind it, with its own invalidation to keep in step. Leave it out.

1. Export the schema

A deployment is built from a `schema.graphql` file next to your `gpilot.toml` (see deployments). It has to be the schema Yoga serves, **with its directives left in**: `@cacheControl` and `@surrogateKey` are read from that file, not from your server.

`printSchema` from the `graphql` package drops every directive you applied, so print with `printSchemaWithDirectives` from `@graphql-tools/utils` . It works on the schema `createSchema` returns:

scripts/export-schema.ts

```
import { writeFileSync } from "node:fs"
import { printSchemaWithDirectives } from "@graphql-tools/utils"
import { schema } from "../src/schema" // the result of createSchema({ typeDefs, resolvers
})

writeFileSync("schema.graphql", printSchemaWithDirectives(schema))
```

Run it before every deployment, for example as a step in your CI workflow. Deploy the new schema to GraphPilot before you release the server that adds fields to it: a query selecting a field the deployed schema does not

declare is answered but never cached (`CACHE_SKIPPED_UNRESOLVED_SELECTION`). See deploy the schema before the origin.

2. Make responses cacheable

Yoga sends no `Cache-Control` header of its own, so GraphPilot takes the cache policy from the `@cacheControl` hints in your schema. Declare the directive in your type definitions, so the schema stays valid for Yoga too, and annotate what may be cached:

src/schema.graphql

```
enum CacheControlScope {
  PUBLIC
  PRIVATE
}

directive @cacheControl(
  maxAge: Int
  scope: CacheControlScope
  inheritMaxAge: Boolean
  swr: Int
) on FIELD_DEFINITION | OBJECT | INTERFACE | UNION

type Query {
  products: [Product!]! @cacheControl(maxAge: 300, swr: 3600)
  me: User @cacheControl(maxAge: 60, scope: PRIVATE)
}
```

Yoga itself ignores the directive. A hint without `scope` is `PUBLIC` , so a field whose answer depends on the caller needs a scope: `PRIVATE` for data about one person, a named scope for data shared within an organization; see partitioning your cache. A field hint that states no scope keeps its return type's scope; see a field hint overrides only the arguments it sets.

If you would rather decide the freshness lifetime (TTL) in code, send a `Cache-Control` header from your origin instead. GraphPilot follows it before the schema (see how long an entry lives).

3. Verify the request signature

GraphPilot signs every request it forwards with your service's signing key (see request signing). Checking that signature lets your origin refuse anything that did not come through GraphPilot.

You need the signing key. The portal and `gpilot service create` each show it once, when the service is created, and neither shows it again. If you did not keep it, rotate it in the portal to get a new one. Store it as `GRAPHPILOT_SIGNING_KEY` .

Copy the Node.js `verifyGraphPilotSignature` function from verify the signature on your origin into `src/graphpilot-signature.ts` . The signature covers the **raw** request body. A Yoga plugin's `onRequest` hook runs

before Yoga reads the body, so it can hash a copy of it:

src/graphpilot-plugin.ts

```
import type { Plugin } from "graphql-yoga"
import { verifyGraphPilotSignature } from "./graphpilot-signature"

export function useGraphPilotSignature(keys: string[]): Plugin {
  return {
    async onRequest({ request, url, fetchAPI, endResponse }) {
      // A clone, so Yoga can still read the original body.
      const rawBody = Buffer.from(await request.clone().arrayBuffer())
      const headers = Object.fromEntries(request.headers)
      const valid = keys.some((signingKey) =>
        verifyGraphPilotSignature({
          method: request.method,
          pathAndQuery: url.pathname + url.search,
          rawBody,
          headers,
          signingKey,
        }),
      )
      if (!valid) endResponse(new fetchAPI.Response(null, { status: 401 }))
    },
  }
}
```

src/server.ts

```
import { createServer } from "node:http"
import { createYoga } from "graphql-yoga"
import { schema } from "./schema"
import { useGraphPilotSignature } from "./graphpilot-plugin"

const yoga = createYoga({
  schema,
  // During a key rotation, add the pending key here as a second entry.
  plugins: [useGraphPilotSignature([process.env.GRAPHPILOT_SIGNING_KEY!])],
})

createServer(yoga).listen(4000)
```

`onRequest` runs for every request that reaches your plugin. Yoga's built-in health check (`/health`) answers before it, so probes to that path are unaffected; if your load balancer probes any other path directly, return early for it before verifying. `url` has been through the URL parser, which can re-encode unusual characters; if your clients send such characters in the query string, take the target from the raw request line instead. The known test vector checks your setup byte for byte.

4. Purge when data changes

A mutation that returns the changed object purges it for you. GraphPilot tags the mutation response with the keys of the objects in it and purges every cached response that contained them (see automatic invalidation). So let your mutations return the object they changed, not a `Boolean`:

```
type Mutation {
  updateProduct(id: ID!, input: ProductInput!): Product!   # purges Product:id:<id>
  deleteProduct(id: ID!): Boolean!                          # purges nothing
}
```

The client does not have to select `id` for this to work: GraphPilot adds the key fields to the query and removes them again before the response reaches the client.

When data changes **outside GraphQL** (an admin tool, an import job, a queue consumer), purge the keys from that code with `gpilot purge` or the `purgeBySurrogateKey` mutation, for example `Product:id:<id>`. The API key needs the `cache:purge` permission. See purging through the API and CLI for the call, its payload and where the service id comes from.

Related

- The `@cacheControl` directive
- Request signing
- Automatic invalidation
- Deploy from CI

CONNECT YOUR ORIGIN

PostGraphile

This page takes a PostGraphile v5 origin, served through Grafserv on Express, through four steps: export the schema GraphPilot deploys, make responses cacheable, check that requests really came through GraphPilot, and purge when data changes.

PostGraphile generates its schema from your database, so there is no SDL file of yours to annotate. That shapes the first two steps: the file GraphPilot deploys is one PostGraphile writes.

1. Export the schema

A deployment is built from a `schema.graphql` file next to your `gpilot.toml` (see deployments). PostGraphile writes one for you on every schema build when you set `exportSchemaSDLPath` in your preset:

graphile.config.ts

```
import { PostGraphileAmberPreset } from "postgraphile/presets/amber"
```

```
const preset: GraphileConfig.Preset = {
  extends: [PostGraphileAmberPreset],
  schema: {
    exportSchemaSDLPath: `${process.cwd()}/schema.graphql`,
    sortExport: true,
  },
  // pgServices, grafserv, …
}

export default preset
```

That file is enough to deploy. It carries no cache hints, though, because PostGraphile does not know about any. Step 2 shows how to add them.

PostGraphile derives its schema from your database, so a migration that adds a column or a table adds fields as soon as PostGraphile picks it up. Export and deploy the schema to GraphPilot as part of that migration's release: a query selecting a field the deployed schema does not declare is answered but never cached (`CACHE_SKIPPED_UNRESOLVED_SELECTION`). See deploy the schema before the origin.

2. Make responses cacheable

GraphPilot reads `@cacheControl` hints from the `schema.graphql` you deploy. To get them into a generated schema, attach them to the types in a small plugin and print the schema with `printSchemaWithDirectives` from `@graphql-tools/utils`, which keeps directives that `printSchema` drops. The plugin below replaces `exportSchemaSDLPath` from step 1:

graphpilot-schema-plugin.ts

```
import { writeFileSync } from "node:fs"
import { printSchemaWithDirectives } from "@graphql-tools/utils"

// Freshness per GraphQL type. Row-level security usually makes rows per caller: say
PRIVATE
// unless every caller sees the same data.
const HINTS: Record<string, { maxAge: number; scope?: "PUBLIC" | "PRIVATE"; swr?: number }>
= {
  Product: { maxAge: 300, scope: "PUBLIC" },
  Order: { maxAge: 60, scope: "PRIVATE" },
}

const DIRECTIVE_SDL = `
enum CacheControlScope { PUBLIC PRIVATE }
directive @cacheControl(maxAge: Int, scope: CacheControlScope, inheritMaxAge: Boolean, swr:
Int) on OBJECT | INTERFACE | UNION | FIELD_DEFINITION
`

export const GraphPilotSchemaPlugin: GraphileConfig.Plugin = {
```

```
  name: "GraphPilotSchemaPlugin",
  version: "0.0.0",
  schema: {
    hooks: {
      finalize(schema) {
        for (const [typeName, hint] of Object.entries(HINTS)) {
          const type = schema.getType(typeName) as { extensions: Record<string, any> } |
undefined
          if (!type) continue
          type.extensions = {
            ...type.extensions,
            directives: { ...type.extensions?.directives, cacheControl: hint },
          }
        }
        const sdl = printSchemaWithDirectives(schema, { pathToDirectivesInExtensions:
["directives"] })
        writeFileSync("schema.graphql", DIRECTIVE_SDL + sdl)
        return schema
      },
    },
  },
}
```

Add it to your preset with `plugins: [GraphPilotSchemaPlugin]`. The hints exist only in the exported file; PostGraphile does not execute them, and nothing about your API changes.

> ⚠ **Check the scope of every hint**
>
> A hint without `scope` is `PUBLIC`: one cached entry, served to every caller. With row-level security, two callers asking the same query usually get different rows, so a hint on such a type needs `scope: PRIVATE` (one entry per caller, once your service defines a `private` partition key; without one, private responses are not cached) or a named scope. Only data that is the same for everyone belongs in `PUBLIC`.

If you set hints on fields too, a field hint that states no scope keeps its return type's scope. See a field hint overrides only the arguments it sets.

The alternative is a `Cache-Control` header from your origin, which GraphPilot follows before the schema (see how long an entry lives). In Express that is one middleware, but it applies to every query the route serves, so it only suits an API whose data is the same for everyone.

3. Verify the request signature

GraphPilot signs every request it forwards with your service's signing key (see request signing). Checking that signature lets your origin refuse anything that did not come through GraphPilot.

You need the signing key. The portal and `gpilot service create` each show it once, when the service is created, and neither shows it again. If you did not keep it, rotate it in the portal to get a new one. Store it as

`GRAPHPILOT_SIGNING_KEY`.

Copy the Node.js `verifyGraphPilotSignature` function from verify the signature on your origin into `src/graphpilot-signature.ts`. The signature covers the **raw** request body. Parse the body with `express.json()` before Grafserv, keeping the bytes, and verify in a middleware on the same path. Grafserv's Express adapter uses a body that is already parsed:

src/server.ts

```
import { createServer, type IncomingMessage } from "node:http"
import express from "express"
import { grafserv } from "postgraphile/grafserv/express/v4"
import { pgl } from "./pgl"
import { verifyGraphPilotSignature } from "./graphpilot-signature"

type WithRawBody = IncomingMessage & { rawBody?: Buffer }

// During a key rotation, add the pending key here as a second entry.
const SIGNING_KEYS = [process.env.GRAPHPILOT_SIGNING_KEY!]

const app = express()
app.use(
  "/graphql",
  express.json({ verify: (req, _res, buf) => { (req as WithRawBody).rawBody = buf } }),
  (req, res, next) => {
    const valid = SIGNING_KEYS.some((signingKey) =>
      verifyGraphPilotSignature({
        method: req.method,
        pathAndQuery: req.originalUrl,
        rawBody: (req as WithRawBody).rawBody ?? Buffer.alloc(0),
        headers: req.headers,
        signingKey,
      }),
    )
    if (valid) next()
    else res.status(401).end()
  },
)

const server = createServer(app)
const serv = pgl.createServ(grafserv)
await serv.addTo(app, server)
server.listen(5678)
```

A request with no body (a `GET`) hashes the empty string, which is what `Buffer.alloc(0)` gives you. `express.json()` accepts 100 KB by default; raise its `limit` if your clients send larger queries. It also only reads `application/json` bodies; if your clients send another content type, widen it with `type`, or a signed request with such a body fails verification. The known test vector checks your setup byte for byte.

4. Purge when data changes

A mutation that returns the changed row purges it for you. GraphPilot tags the mutation response with the keys of the objects in it and purges every cached response that contained them (see automatic invalidation). PostGraphile's generated mutations already return the row in their payload (`updateProduct { product { … } }`), so this works as long as the client selects that row. It does not have to select the key fields: GraphPilot adds them to the query and removes them again before the response reaches the client.

Rows often change **outside GraphQL** in a PostGraphile setup: a migration, a job writing straight to the database, a trigger. Purge the affected keys from that code with `gpilot purge` or the `purgeBySurrogateKey` mutation; see purging through the API and CLI. The API key needs the `cache:purge` permission.

A key names the object by the key field GraphPilot found in the response. In PostGraphile's default preset that is `id`, the global Node ID, not the row's primary key, so the key for a product is `Product:id:<node id>`: use the Node ID your API returns for that row. Enable `gp-surrogate-key` on a test service to see the exact keys a response carries.

Related

- The `@cacheControl` directive
- Surrogate keys
- Request signing
- Deploy from CI

CONNECT YOUR ORIGIN

Federation routers

GraphPilot does not compose subgraphs. It sits **in front of** your router or gateway, which keeps doing the composition and query planning it already does. To GraphPilot, the router is the origin of one service, and the composed schema is that service's schema.

```
client → GraphPilot → your router or gateway → subgraphs
```

This works with any router that speaks GraphQL over HTTP: Apollo Router, Cosmo Router, Hive Gateway, or your own gateway. It is available on every plan.

1. Export the composed schema

A deployment is built from a `schema.graphql` file next to your `gpilot.toml` (see deployments). For a federated graph, that file is the **composed** schema: the one your router serves to clients, not a single subgraph's.

Your composition tool produces it, whether that is Rover, the Cosmo CLI, Hive, or a script of your own. Write it to `schema.graphql` as part of the same pipeline that publishes a new composition to your router, so GraphPilot and

the router always run the same schema. See Deploy from CI for the GraphPilot half of that pipeline. Deploy to GraphPilot first and publish to the router after it: while the router serves fields the deployed schema does not declare, a query selecting them is answered but never cached (`CACHE_SKIPPED_UNRESOLVED_SELECTION`). See deploy the schema before the origin.

Entity keys work without extra setup when your types have an `id`, `_id` or `key` field, or any field of type `ID`. If your entities are keyed on other fields, tell GraphPilot which ones with `@surrogateKey`, or map an existing directive to it.

2. Make responses cacheable

GraphPilot takes a response's cache policy from one of two places, in this order (see how long an entry lives):

1. **A `Cache-Control` header on the router's response.** If your router sends one, for example because it passes on or computes a header from what the subgraphs return, GraphPilot follows it. Check what your router actually sends before relying on it: send a query straight to the router and read the response headers.
2. **`@cacheControl` hints in the `schema.graphql` you deploy.** GraphPilot reads them from that file, not from the router or the subgraphs. Hints written in a subgraph only count if your composition keeps them in the composed schema (in Apollo Federation 2, a directive has to be marked with `@composeDirective` for that).

If neither is there, nothing is cached.

> ⚠ **Scope follows the whole query**
>
> A composed query can mix subgraphs: one field public, another per user. GraphPilot keys the response on every scope any selected field asks for, so one `PRIVATE` field makes the whole response private. A hint without `scope` is `PUBLIC`, so mark per-user fields explicitly. Private responses are cached only once your service defines a `private` partition key. See scopes in the schema.

3. Verify the request signature

GraphPilot signs every request it forwards with your service's signing key (see request signing). With a router behind it, the request GraphPilot signs is the one that reaches the **router**. Your subgraphs receive new requests the router makes, which carry no GraphPilot signature.

So verify at the first hop GraphPilot talks to:

- **In the router**, through whatever extension point it offers for inspecting an incoming request with its body: a plugin, a coprocessor, or a custom module. Hive Gateway is built on GraphQL Yoga and takes plugins with the same `onRequest` hook, so the Yoga plugin is a starting point there.
- **In a reverse proxy in front of the router**, if the router has no suitable hook. The proxy checks the signature and forwards only requests that pass.

Either way, the check needs the raw request body, the method, the path with its query string, and the `gp-timestamp` and `gp-signature` headers, exactly as described in verify the signature on your origin. To keep the

subgraphs private as well, make them reachable only from the router.

4. Purge when data changes

Mutations sent through GraphPilot purge automatically, whichever subgraph resolves them. GraphPilot reads the keys of the objects in the mutation response and purges every cached response that contained them (see automatic invalidation). Let each mutation return the object it changed.

When a subgraph's data changes **outside GraphQL** (a job, an import, another service writing to the same database), call the purge API from there. The API key needs the `cache:purge` permission.

Related

- How long an entry lives
- Surrogate keys
- Request signing
- Deploy from CI

CONNECT YOUR ORIGIN

Other servers

GraphPilot works with any origin that speaks HTTP. The pages for Apollo Server, GraphQL Yoga, PostGraphile and federation routers show the steps for those servers. This page gives the same four steps in general form, then what differs on a few other servers.

The four steps

1. **Export the schema.** A GraphQL service deploys a `schema.graphql` next to its `gpilot.toml`, and it has to be the schema your server runs **with its directives left in**, because `@cacheControl` and `@surrogateKey` are read from that file. Most printers drop applied directives unless you ask for them. Export it before every deployment, for example from CI, and deploy it before the server that adds fields to it: a query selecting a field the deployed schema does not declare is never cached. See deploy the schema before the origin.
2. **Make responses cacheable.** Either your origin sends `Cache-Control`, or your schema carries `@cacheControl` hints and your origin sends no `Cache-Control` of its own. An origin header that states a lifetime always wins; see how long an entry lives. Check what your server actually sends before you decide: some frameworks and hosting platforms add a header you did not write, see why is nothing cached?
3. **Verify the request signature**, so your origin refuses anything that did not come through GraphPilot. The scheme and verifiers for Node.js and Python are on verify the signature on your origin. Hash the raw body as received, before your framework parses it.
4. **Purge when data changes.** A GraphQL mutation that returns the object it changed purges it on its own; see automatic invalidation. For changes that do not pass through GraphPilot, use the purge API or the CLI.

Hot Chocolate

Hot Chocolate exports its schema from the command line once your app runs through `RunWithGraphQLCommandsAsync`:

```
dotnet run -- schema export --output schema.graphql
```

Its cache-control feature (`AddCacheControl()` with `UseQueryCache()`) computes a `Cache-Control` header from its own `@cacheControl` directive, including `s-maxage` from `sharedMaxAge`, `public` or `private` from `scope`, and `Vary` from `vary`. GraphPilot follows that header, so this is the simplest setup: keep it, and give your service a `private` policy for responses it marks `private`. If you would rather GraphPilot decide from the schema, for example to use named scopes, switch the header off and annotate the exported schema with GraphPilot's `@cacheControl`.

graphql-java

graphql-java, and Spring for GraphQL on top of it, sends no `Cache-Control` of its own, so the schema decides unless your application adds a header.

- **Schema first**: your `.graphqls` files are the schema. Declare `@cacheControl` in them as shown in its definition, so graphql-java accepts the annotations, and concatenate the files into `schema.graphql`.
- **Code first**: print the built schema with directives included, for example `new SchemaPrinter(SchemaPrinter.Options.defaultOptions().includeDirectives(true)).print(schema)`, and write the result to `schema.graphql`.

For the signature, read the raw body before Spring's message converters do, for example in a servlet filter that wraps the request in a caching wrapper.

Strawberry

Strawberry exports its schema with its CLI:

```
strawberry export-schema app.schema:schema --output schema.graphql
```

Strawberry has no `@cacheControl` of its own. Declare it as a schema directive, so it appears in the exported file:

```
import strawberry
from strawberry.schema_directive import Location

@strawberry.schema_directive(
    name="cacheControl",
    locations=[Location.OBJECT, Location.FIELD_DEFINITION],
)
```

```python
class CacheControl:
    max_age: int = strawberry.directive_field(name="maxAge")

@strawberry.type(directives=[CacheControl(max_age=300)])
class Product:
    id: strawberry.ID
    name: str
```

Add a `scope` field in the same way for types that are not `PUBLIC`, with a `CacheControlScope` enum that matches your schema; see scopes in the schema. In ASGI frameworks, read the raw body for the signature before Strawberry parses it.

Any HTTP server

A REST or other plain HTTP API has no schema to annotate, so step 1 does not apply. Step 2 is always your origin's `Cache-Control`. Deploy with `gpilot.toml` alone: without a schema file, `gpilot deploy` deploys the service as plain HTTP, with GraphQL features off.

Step 4 uses the `Surrogate-Key` header: on the response to a read it tags the entry, on the response to a write it purges. Caching a REST API walks through it.

Related

- Quickstart and Quickstart: HTTP
- Request signing
- Lifetime from the schema
- Deploy from CI

Caching

CACHING

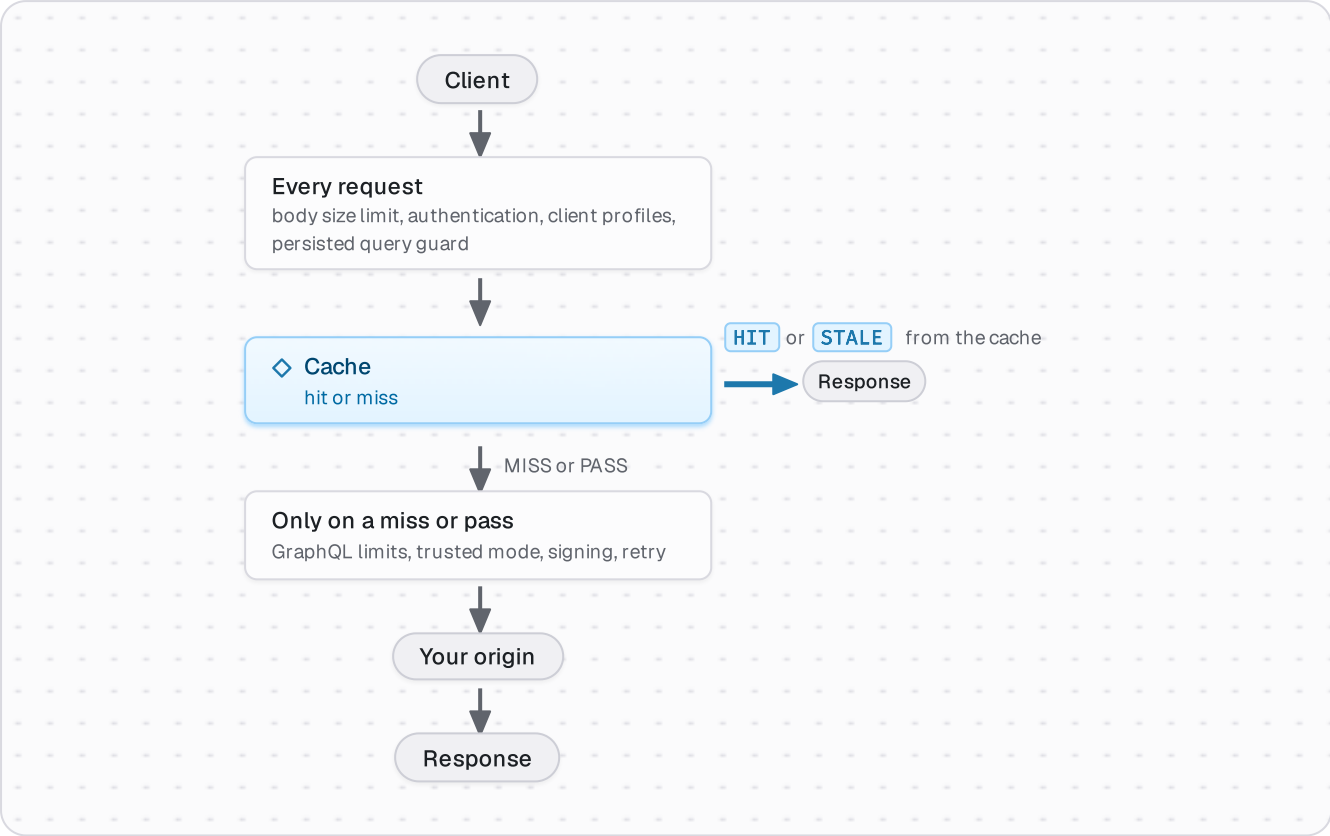
How a request flows

Every request goes through the same three blocks. Knowing which block a check sits in answers most “does X apply on a cache hit” questions on its own.

Ways to control caching

Through	What it controls
Your origin’s response headers	How long an entry lives (`Cache-Control`), whether it is shared or private, what it varies on (`Vary`), and the keys it can be purged by (`Surrogate-Key`).
Your GraphQL schema	Lifetime and stale window (`@cacheControl`), who shares an entry (scopes), and purge keys (`@surrogateKey`).
`gpilot.toml`	Who a caller is and how the cache is partitioned (`[cache.policy]`), which headers and cookies reach the cache (filters), what a client’s own `Cache-Control` may do (`[cache.request_directives]`), and per-client cache rules (client profiles).
The API, the CLI and the portal	Removing entries before they expire: purge on demand.

Three blocks



The first block runs on every request, a cache hit included. The third runs only when the request goes on to your origin.

- **Every request.** The request body size limit is applied, credentials are verified, a client profile is matched, and the persisted query guard rejects what the mode does not allow.
- **Cache.** The request is answered from the cache, or it goes on. What decides this is on when a response is cached.
- **Only on a miss or pass.** The GraphQL document is parsed and checked against the GraphQL limits, trusted mode refuses an unregistered operation, the request is signed and sent to your origin, and a failed origin request is retried. A mutation's response then purges what it named.

Authentication runs on every request

Authentication sits **above** the cache, so a credential is verified whether the answer comes from the cache or from your origin. A caller with an expired or forged token is refused at the edge, and never receives a cached response.

This is also why cache partitioning can use claims at all: the claims a partition is derived from have already been verified.

> ⚠ **Authentication is not authorization**
>
> Verifying who is calling is not the same as deciding what they may see. That decision is your origin's, and on a cache hit your origin is not consulted. An entry keyed to one caller keeps being served to that caller for its whole lifetime, so a permission revoked in the meantime is not reflected until the entry expires or is purged. See revoking access.

The GraphQL layer runs only on a miss

Parsing, planning and selection set expansion sit **below** the cache. A hit does not parse the document, which is where most of the latency saving comes from.

Runs on every request, hit included	Runs only when the request reaches the GraphQL layer (a miss or a pass)
The body size limit (`request.max_body_bytes`)	The depth limit (`request.max_query_depth`)
Authentication	The document size limit (`request.max_document_bytes`)
Client profiles: version floor, body size cap, `cache = "off"` , `max_age`	The selection limit (`request.max_selections`)
The persisted query guard	The introspection setting (`request.introspection`)
	The refusal of an unregistered operation in trusted mode

Consequences worth knowing:

- **A cached response is never re-checked.** A document that was under every limit when it was stored keeps being served from the cache if you tighten a limit later, until the entry expires or is purged. Refusals themselves are never cached.
- **Query validation happens at your origin, not at the edge.** GraphPilot parses and plans; it does not validate the operation against your schema. Two exceptions: a document GraphPilot cannot read unambiguously (one that does not parse, names an operation it lacks, or spreads fragments in a cycle) is refused with `400` and `INVALID_GRAPHQL_REQUEST` , and a query that selects a field your deployed schema does not declare is forwarded but its answer is never stored.
- **A cached response is replayed, not regenerated.** It is served as stored.

Trusted mode is enforced on misses

The refusal of an unregistered operation sits **below** the cache, so it runs on a miss and not on a hit. A response cached while an operation was still permitted stays reachable after you switch to `trusted` mode, until it expires or is purged: the cache is keyed by the query hash, and nothing in that key changes with the mode.

`gpilot pq rm` deletes the registry entry **and** purges the cache key for it, which is why clearing what your clients registered is part of switching modes. See persisted queries.

The headers GraphPilot adds to a response, and the ones it removes from your origin's, are listed in HTTP headers. With shielding active, a miss at an edge location is sent on to the shield, which runs the same blocks with its own cache.

What is not in the path

Stating the absences, because they are what an evaluation actually turns on:

- **No conditional revalidation.** GraphPilot never sends a conditional request and never handles a `304`. A background revalidation is a full request.
- **No change to what your response says.** The data your origin produced is the data the client receives. A plain HTTP body is passed through byte for byte, on a hit and on a miss. A GraphQL body is not re-serialized: GraphPilot decodes a compressed body and cuts out only what it has to (the fields it injected, an error on one of those fields, and an `errors` array that this leaves empty), keeping your origin's key order and number formatting. In the rare case a cut cannot be made in place, the edited response is serialized again, and its formatting can then differ while its content does not. Nothing else about the content changes.
- **No query rewriting beyond selection set expansion**, which adds key fields and removes them again before responding.
- **No composition of subgraphs.** One service is one origin. To cache a federated graph, put GraphPilot in front of your federation router, on any plan: see federation routers.
- **No rate limiting and no request cap.** See request limits.
- **No schema validation of client operations at the edge.** Your origin remains the authority.

Where a WebSocket leaves the path

A subscription handshake is an ordinary HTTP request and goes through all three blocks, so the body size limit, authentication and signing all apply to it. It is never retried. Once it is upgraded, the connection is a tunnel: nothing on this page applies to the messages that follow, and your origin is on its own for them.

Related

- When a response is cached
- Cache statuses
- Hardening a public API

CACHING

When a response is cached

Two questions decide what the edge cache does with a request: whether an entry may be **served**, and whether the origin's answer may be **stored**. This page walks both, in the order GraphPilot evaluates them.

Every refusal below reports `gp-cache: PASS` and a stable code on the `gp-cache-reason` response header and in your logs. Those codes are listed with the cache statuses.

Reading: can this be served from the cache?

The lookup happens before anything is parsed. In order:

1. **Is this a WebSocket upgrade?** A subscription handshake is never cacheable. `PASS`, `CACHE_PASS_WEBSOCKET_UPGRADE`.
2. **Is the request method one you never cache?** `cache.policy.uncacheable_methods` lists them, and it holds `OPTIONS` unless you change it. `PASS`, `CACHE_PASS_UNCACHEABLE_METHOD`. See preflights and the methods you never cache.
3. **Does the matched client profile take this client off the cache?** `PASS`, `CACHE_PASS_CLIENT_CACHE_OFF`. See client profiles.
4. **Is caching bypassed for this request?** If your service configures `cache.policy.bypass` and the expression matches, the request goes to your origin and its response is not stored. `PASS`, `CACHE_PASS_BYPASS_POLICY`.
5. **Did the client ask for a response the origin has just confirmed?** Only when `cache.request_directives.no_cache` is `"honor"`: a request `Cache-Control` of `no-cache` or `max-age=0` goes to your origin and its response is not stored. `PASS`, `CACHE_PASS_REQUEST_NO_CACHE`. Ignored by default; see the client's own `Cache-Control`.
6. **Is there a cache key at all?** `PASS`, `CACHE_PASS_MISSING_CACHE_KEY`. A GraphQL request whose body carries both a document and a persisted query hash that is not that document's hash has none, so it is never looked up or stored; it still runs as before. An APQ retry that sends its own document's hash keeps its key.
7. **Is edge caching withdrawn?** In lean mode, after the plan's request quota is used up, the cache stands aside entirely: no lookup, no write, and no `gp-cache` header. Invalidation keeps running, see automatic invalidation.
8. **Can the cache be reached?** If the lookup fails, the request is served from the origin rather than failing. `PASS`, `CACHE_PASS_READ_DISABLED`.

If an entry matching the key exists and can be decoded, it is served, unless the matched client profile caps the age it accepts with `max_age` and the entry is older. Then this request goes to your origin and reports `PASS` with `CACHE_SKIPPED_ENTRY_NOT_REPLACED` (the log line `CACHE_CLIENT_MAX_AGE_MISS` records why the entry was declined). The origin's answer is served without being stored, and the entry itself is unchanged and keeps serving other clients.

An entry that cannot be read is treated as absent: the answer from your origin is written over it and reports `MISS`, or `PASS` with `CACHE_SKIPPED_ENTRY_NOT_REPLACED` if that write is not possible. Otherwise:

- `HIT` if it is still fresh.
- `STALE` if it is past its freshness lifetime but inside a stale window. The stored response is returned immediately and the origin is asked for a fresh one in the background.

> **Note**
> A hit answers without your origin being consulted, so authorization decisions your origin makes are not re-run for the length of the entry's lifetime. Authentication of the caller still runs on every request. See revocation.

Writing: may this response be stored?

On a miss, the origin's response runs through the following. The first rule that refuses ends it, and the response is still delivered to the client in full: only storing is refused.

With shielding, an edge location that got its answer from the shield checks one thing before all of these: a response the shield served `STALE` or `PASS`, or one whose stored form or age the edge location cannot read, is passed on and not stored there, `CACHE_SKIPPED_SHIELD_RESPONSE_NOT_STORABLE`. A response it does store lives only for the lifetime the shield's copy has left, so an entry never outlives what your origin allowed across both locations. When nothing is left, it is not stored, `CACHE_SKIPPED_SHIELD_ENTRY_EXHAUSTED`.

0. Only a query is ever stored

A mutation or a subscription response is never written to the cache, and no header can opt back in. `CACHE_SKIPPED_UNCACHEABLE_OPERATION`.

A plain HTTP response, or the response to a body forwarded unchecked under `request.forward_unchecked_bodies`, is not refused here: nothing knows it to be a write. A GraphQL document that does not parse never gets this far: it is refused with `400` and `INVALID_GRAPHQL_REQUEST` before your origin is called. Any method except those in `cache.policy.uncacheable_methods` can be cached when the response says so.

Then the selection: a field your deployed schema does not declare has no hint and no scope GraphPilot could read, so nothing says who may share its value or for how long. Data for a query that selects such a field, or a fragment on such a type, at any depth, is not written, and no header can opt back in. `CACHE_SKIPPED_UNRESOLVED_SELECTION`. It usually means your origin serves a newer schema than the one you deployed to GraphPilot; see deploy the schema before the origin. `__typename`, `__schema` and `__type` never count.

Next, the request's own `Cache-Control`: if it says `no-store` and `cache.request_directives.no_store` is `"honor"`, which is the default, the response is not written. `CACHE_SKIPPED_REQUEST_NO_STORE`. Nothing a policy says outranks it, because it is a statement about the exchange rather than about the response. See the client's own `Cache-Control`.

1. An error status is not stored unless you opt in

A response with a `4xx` or `5xx` status is not stored by default, even when it carries a storable `Cache-Control` or your schema gives it a lifetime. Many frameworks and hosting platforms attach a blanket `Cache-Control` to every response, errors included, and an error served from the cache repeats one failure to every caller until the entry

expires. It reports `PASS` with `gp-cache-reason: CACHE_SKIPPED_ERROR_STATUS`. This runs before the refusals of step 2, so a `503` that also sets `Set-Cookie` reports `CACHE_SKIPPED_ERROR_STATUS`.

To store an error response your origin marked cacheable, such as a `404` for a resource that does not exist, list its status in `cache.policy.store_error_statuses`. A listed status goes on to the rules below, starting with step 2. This applies to plain HTTP and GraphQL alike; a GraphQL `200` that carries `errors` keeps its own rule.

gpilot.toml

```
[cache.policy]
store_error_statuses = [404, 410]
```

2. Refusals that depend only on what the response carries

These stand ahead of both policy sources. Whatever made the response cacheable, it is not storable:

The response carries	Code
`Set-Cookie`, `Set-Cookie2`	`CACHE_SKIPPED_SET_COOKIE`
`WWW-Authenticate`, `Proxy-Authenticate`	`CACHE_SKIPPED_AUTHENTICATION_HEADER`
`Authentication-Info`, `Proxy-Authentication-Info`	`CACHE_SKIPPED_AUTHENTICATION_HEADER`
`Clear-Site-Data`, `Set-Login`	`CACHE_SKIPPED_AUTHENTICATION_HEADER`
A GraphQL `errors` array, including partial data beside it	`CACHE_SKIPPED_GRAPHQL_ERRORS`
A GraphQL response body GraphPilot cannot parse as JSON	`CACHE_SKIPPED_UNPARSEABLE_GRAPHQL_RESPONSE`
`Vary: *`	`CACHE_SKIPPED_VARY_WILDCARD`

Each header group states something about **one caller's session at one instant**, which a cache entry would replay on every hit: a stored `Set-Cookie` re-asserts that cookie on the next caller, a stored `Clear-Site-Data` wipes their session. GraphQL errors describe the moment, not the data. `Vary: *` can never match a later request; see your origin's `Vary` header.

A GraphQL response with a JSON content type is read before it is stored, to find errors and derive the surrogate keys a purge needs. When the body cannot be read, neither is possible: an entry made from it could hold an error or be out of reach of every keyed purge, so it is not stored. The client still receives the body exactly as your origin sent it, including any key fields selection set expansion added, because removing those needs the parse that failed. The common causes are a body that is not valid JSON, a body larger than 133 MiB, and an integer larger than 18446744073709551615, which is valid JSON that GraphPilot does not read. Return such numbers as strings (an `ID` or a custom scalar); most GraphQL clients need that anyway, since JavaScript loses integer precision far below that value. A plain HTTP body is never parsed, so this rule does not apply to it.

Presence alone is enough. `Set-Cookie` is checked per cookie: the log line names the cookie that blocked it, and a cookie on `cache.cookies.allow` does not count. The other headers in the table stop refusing once you name them in `cache.headers.allow`.

> ⚠ **The most common cause of a service that caches nothing**
>
> A framework that touches the session on every request, reads included, sends `Set-Cookie` on every response and loses you edge caching for all of it. If caching stopped working after you introduced cookies, this is why, and `gp-cache-reason` says so.

There is **no exception for private, partitioned entries**: a partition does not stop a caller being handed their own old cookie, which would revert a rotation or a sign-out on the next hit. GraphPilot refuses to store rather than stripping the header, so your client receives what your origin sent.

To get caching back, stop setting these headers on cacheable reads, for example by refreshing a session only on the requests that actually change it, or mark those responses `Cache-Control: no-store` so the intent is explicit.

3. How long, and for whom

What is left is the policy: how long the response may be kept, and who may be served it.

- **How long** comes from your origin's `Cache-Control` header if it states storage or freshness, otherwise from your schema's `@cacheControl`, otherwise nowhere, and the response is not cached. See how long an entry lives for the precedence, from the origin header for the order the header is checked in, and from the schema.
- **For whom** comes from `private` on the header, the scopes the query selected, your vary rules, and, for a request that carried a credential, whether the origin said the response may be shared. A response that cannot be placed is refused rather than shared. See who shares an entry and authenticated requests.

The refusals from this step:

Code	Why
`CACHE_SKIPPED_NO_STORE`	The origin said `no-store`.
`CACHE_SKIPPED_NO_CACHE_DIRECTIVE`	The origin said `no-cache`.
`CACHE_SKIPPED_NO_MAX_AGE`	The deciding source stated no lifetime.
`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`	A private or named-scope response could not be partitioned for this request.
`CACHE_SKIPPED_CREDENTIALED_REQUEST`	A signed-in caller's response did not say it may be shared, and nothing identified the caller.
`CACHE_SKIPPED_NO_CACHE_POLICY`	Neither your origin nor your schema stated a policy.

4. The entry has to fit

A response that reaches this step carrying no surrogate keys at all is not cached, because no purge could reach it: `CACHE_SKIPPED_NO_SURROGATE_KEYS`. This is rare; see the code's entry.

Surrogate keys are compressed to fit a 16,000-byte budget. A response whose keys do not fit even in the most compact form is not cached, rather than cached with keys missing, so that a purge always reaches everything it should. `CACHE_SKIPPED_SURROGATE_KEYS_TOO_LARGE`, and see how many keys fit on one response.

A response that cannot be encoded, or that the cache refuses to write, reports `CACHE_SKIPPED_SERIALIZATION_FAILED` or `CACHE_SKIPPED_INSERT_FAILED`. Both are faults on GraphPilot's side, not configuration problems, and like every refused store they report `PASS`.

What the entry is keyed by

The cache key is derived from the request before anything is parsed, which is why a hit costs no parsing.

> **The route is not in every runtime yet**
>
> Runtime 1.4.0 and earlier do not put the route at the front of a GraphQL key. On such a runtime, a service's GraphQL key starts at the query hash, so the same operation sent to two paths or with two query strings shares one entry. Everything else in this section already applies.

For a GraphQL operation it is four parts, in this order, the last two only when present:

```
<route>#<sha256 of the query text>[#<operationName>][:<hash of the variables>]
```

- The **route** is the path the request was sent to and its query string without the GraphQL parameters (`query`, `operationName`, `variables` and `extensions`), for example `/graphql/acme?locale=de`. A path below a GraphQL path counts as that path, and an origin may answer `/graphql/<tenant>`, `?locale=de` or `?preview=1` differently, so each of them keeps its own entry:
  - The path is taken exactly as sent. `/graphql/Acme`, `/graphql/acme` and `/graphql/acme/` are three entries.
  - The other parameters are read the way a server reads them and sorted by name, so `?locale=de&preview=1` and `?preview=1&locale=de` are one entry, and `%20` and `+` are the same space. The values of a repeated parameter keep the order they were sent in.
  - A parameter your origin does not read still splits the entry. A cache buster such as `?_=<timestamp>`, or a tracking parameter with a value per visit, makes every request a miss.
  - Scheme and hostname are not part of it, whatever `cache.http_key.host` says: that key applies to plain HTTP only.
- The **query hash** is a SHA-256 of the document **exactly as sent**, not normalized: two documents that differ only in whitespace, field order or aliases are two cache entries.
- The **operation name** is included when the request names one.
- The **variables** are included as a hash of the variables object when the request carries one. The hash is per service and does not depend on the order the members were sent in, so two clients sending the same variables in a different order share an entry.

Without its route, the key is the string request telemetry records (the cache key and the variables hash of a request's detail in the portal) and the string the entry's `ck:` surrogate key carries:

```
<sha256 of the query text>[#<operationName>][:<hash of the variables>]
```

So the name a request reports is always the name its entry can be purged by, and one purge of it removes the response on every route it was stored under. The variables hash cannot be recomputed from the variables, and the same variables in two services give two different values: copy it from the request detail rather than computing it.

For anything that is not a GraphQL operation, the key is the request method and the path the request was sent to, query string included, joined by a colon:

```
GET:/products/42?color=blue
```

Scheme and hostname are not part of it, so the same path requested through your service hostname and through a custom domain is one entry. If your origin answers the same path differently per hostname, set `cache.http_key.host = "include"`: the key is then the full URL, `GET:https://catalog-api.k7m2qp.graphpilot.cloud/products/42?color=blue`, and each hostname keeps its own entry.

A request that carries a body is keyed by a SHA-256 of its body as well, so two `POST` requests to the same path with different bodies never share an entry:

```
POST:/search#9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
```

A request without a body keeps the key above.

The method is not part of a GraphQL key, on purpose: the same operation sent to the same route as `POST`, as `GET` with a query string, or as `QUERY` shares one entry. On a `GET` the operation travels in `query`, `operationName`, `variables` and `extensions`, which the route leaves out, because they are already in the key as the query hash, the operation name and the variables hash.

The document is part of the key

Clients generated by `graphql-client`, Apollo and urql send the whole `.gql` file on every request and pick the operation with `operationName`. For those clients the document in the key is the file, not the operation:

- **Adding or editing any operation in a document changes the key of every operation in it.** A one-line change to an unrelated mutation is a cache flush for that file's reads, from the first deployment that ships it.
- **Two clients share an entry only when their surrounding documents are byte-identical.** A web client and a mobile client that both run `GetProduct` from different files warm two entries.

Nothing incorrect is served either way, because the operation name is part of the key. The fix is to register your operations: a persisted query is addressed by the hash of the one operation, so every client sending that hash

shares one entry.

On top of the key, an entry is separated by who may share it: its bucket, your vary rules and whatever your origin's `Vary` names. See which bucket an entry goes into.

Limits and headers

What a request, a response and a cache entry may be at most is on limits. What the client's own `Cache-Control` and a CORS preflight do is on headers.

CACHING

Cache statuses

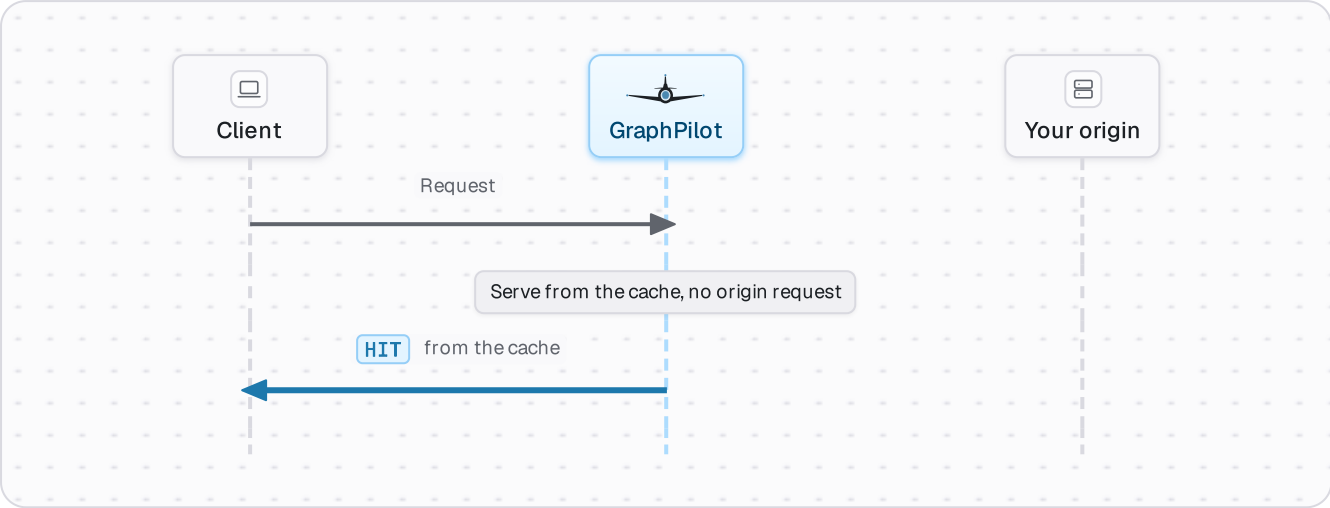
Every response that went through the cache carries a `gp-cache` header with one of four values: `HIT`, `MISS`, `PASS` or `STALE`. It tells you what the cache did with the request without opening a log, so it is the first thing to check when you want to know whether your caching rules apply.

A few things about the header itself:

- GraphPilot removes the `x-cache` and `x-cache-hits` headers Fastly (the edge network GraphPilot runs on) adds, so `gp-cache` is the only cache status on a response.
- It can be switched off with `response.cache_headers = false`. Most troubleshooting reads it, so a common pattern is to leave it on in a staging service and switch it off in production.
- A request refused before the cache (a `401` from authentication, for example) carries no `gp-cache` header.
- A service that runs in lean mode sends no `gp-cache` header on responses the cache would otherwise have looked up, because edge caching is withdrawn. A request that bypasses the cache still reports `PASS` with its reason.

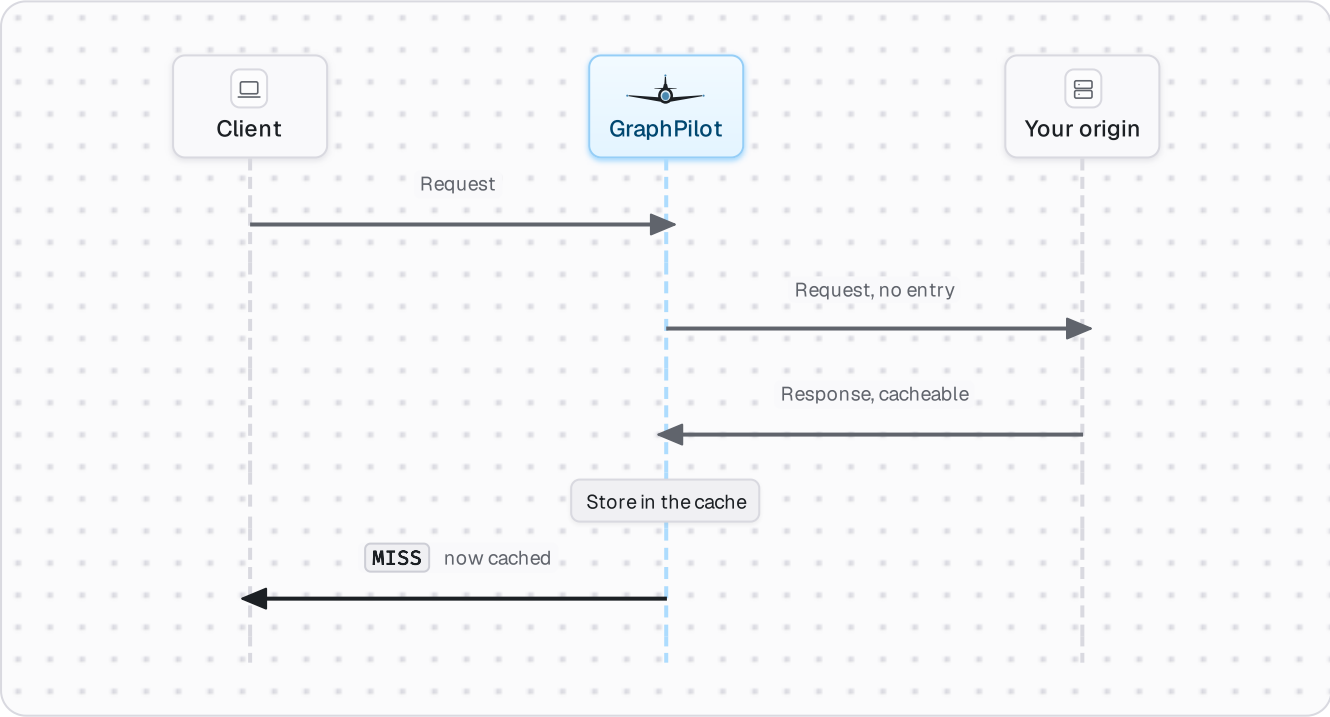
All response headers GraphPilot sets are listed in the headers reference.

Cache HIT



The response came from the cache. A stored entry matched the request and was still fresh, so your origin was not asked. A `HIT` carries no `gp-cache-reason`.

Cache MISS

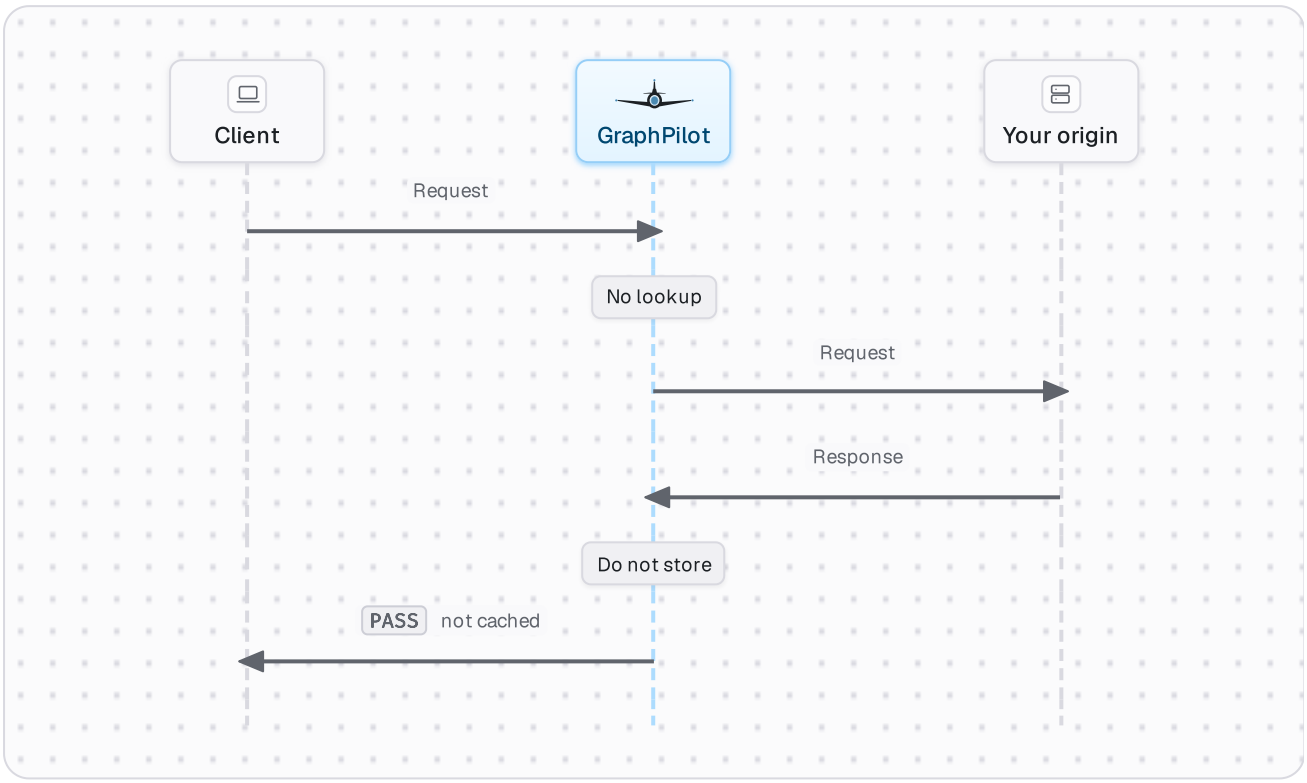


No usable entry existed, so the request went to your origin, and the response **was stored**. The next matching request at the same edge location can be a `HIT`. A `MISS` happens the first time an operation runs at a location, after an entry expires, and after a purge.

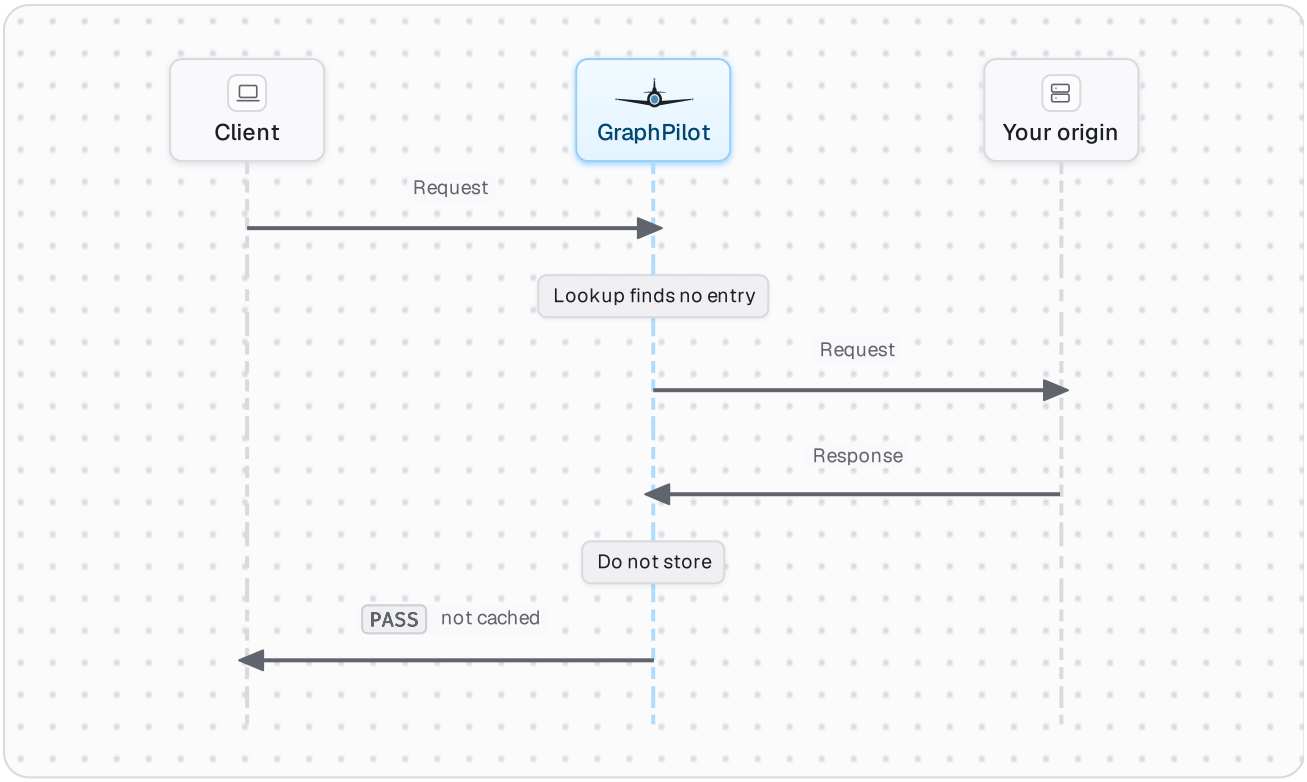
A `MISS` never carries `gp-cache-reason`. If the response was not stored, for whatever reason, the status is `PASS`.

Cache PASS

Cache not used (CACHE_PASS_*)



Not stored (CACHE_SKIPPED_*)



A `PASS` means **nothing was stored**. The response came from your origin and was delivered to the client in full, but no cache entry answered the request and none was written. Every `PASS` carries `gp-cache-reason`, and the prefix of that code tells you where it was decided.

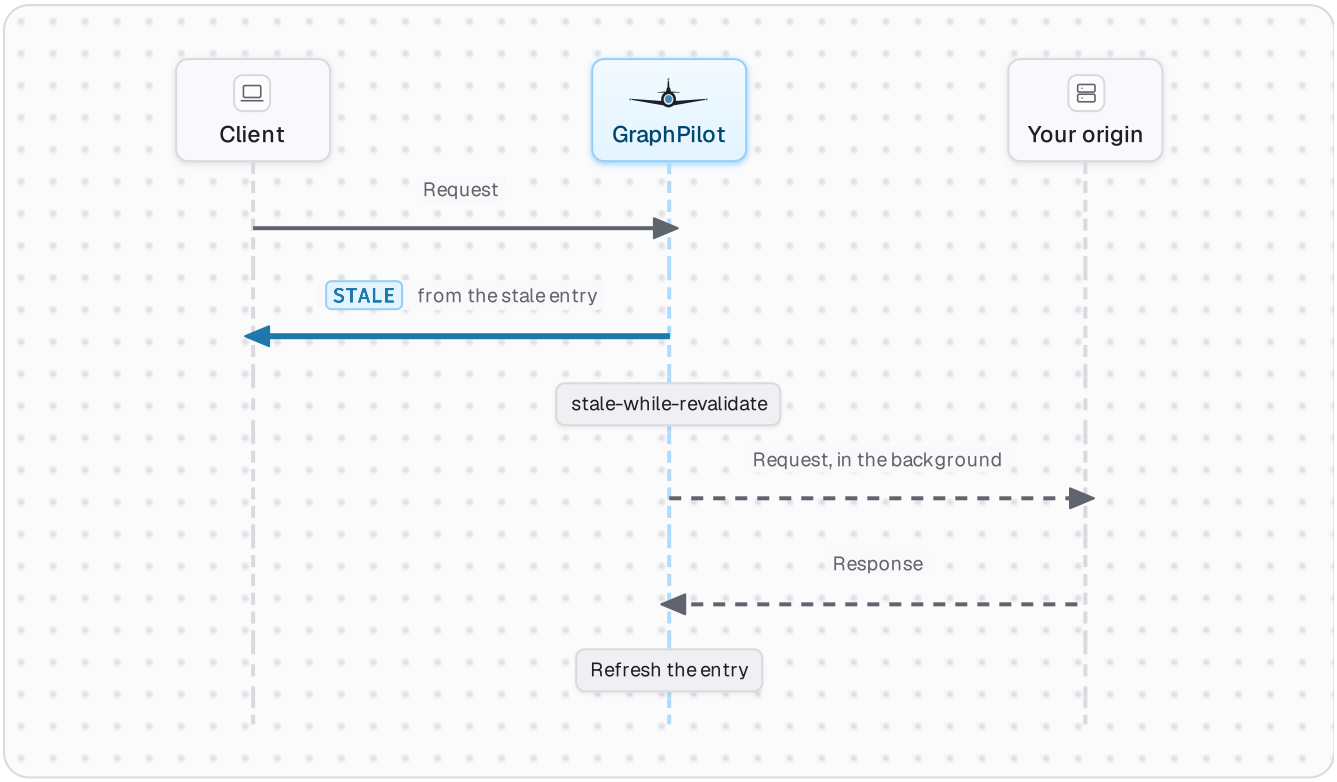
The cache was never consulted (`CACHE_PASS_*`). No lookup happened. This is decided before anything about the request is parsed, so it says nothing about the response: a bypass expression matched, the request was a WebSocket upgrade or used a method you never cache, a client profile takes the caller off the cache, and a few more. Every code is on the cache was not consulted.

The response was not stored (`CACHE_SKIPPED_*`). Usually the cache was looked up and missed, the request went to your origin, and then the request or the response ruled storing out: a mutation, a `Set-Cookie`, a `no-store`, no freshness lifetime, a GraphQL error. Two codes do not fit that pattern: `CACHE_SKIPPED_ENTRY_NOT_REPLACED`, where the lookup found an entry this request did not use, and `CACHE_SKIPPED_NO_ORIGIN_RESPONSE`, where no response came back from your origin at all. A few of these codes are faults on GraphPilot's side rather than properties of your response, and they are still a `PASS`, because nothing was stored. Every code is on the response was not cached, and the order they are decided in is on when a response is cached.

A `4xx` or `5xx` response from your origin is also a `PASS`, with `CACHE_SKIPPED_ERROR_STATUS`, unless its status is listed in `cache.policy.store_error_statuses`. A GraphQL response that is not JSON GraphPilot can parse is delivered unchanged and not stored, with `CACHE_SKIPPED_UNPARSEABLE_GRAPHQL_RESPONSE`.

In Insights, both kinds of code are one column and one filter, **Not cached because**, grouped by who can act on the reason. The request's **Cache** tab still says whether the cache was consulted before the decision.

Cache STALE



The response came from the cache although the entry had passed its freshness lifetime, because it was still inside a stale window. The client got the stored answer immediately, and GraphPilot asked your origin for a fresh copy in the background. That background response goes through the normal store decision.

How long the window is, where it comes from and what switches it off is on stale windows. A `MISS` inside the window is possible too, when the request is handled by a cache server that does not hold the entry; see stale responses within a location.

Why a response was not cached

`gp-cache` tells you **what** happened. On a `PASS`, a second header, `gp-cache-reason`, tells you **why**:

```
gp-cache: PASS
gp-cache-reason: CACHE_SKIPPED_SET_COOKIE
```

`gp-cache-reason` appears only on a `PASS`:

- with a `CACHE_PASS_*` code, the cache was never consulted.
- with a `CACHE_SKIPPED_*` code, the request reached your origin, the answer came back, and it was not stored.

A `HIT`, `MISS` or `STALE` carries no reason, because there is nothing to explain. A `MISS` is itself the signal that the entry was written and the next matching request at that location should be a `HIT`.

Which status a refused store reports

Every `CACHE_SKIPPED_*` code arrives with `gp-cache: PASS`, never with `MISS`. Every code, with what to do about it, is in the codes reference.

The value of `gp-cache-reason` is a stable code, not prose. The same code appears in your logs on the same request, so you can start from either end: notice the header in a browser's network tab and search your logs for the code, or the reverse. `gp-cache` itself stays one of `HIT`, `MISS`, `PASS` or `STALE` and never carries the reason, so anything matching on it is unaffected.

CACHING · HOW LONG

How long an entry lives (TTL)

Every entry GraphPilot stores has a freshness lifetime (TTL). While it lasts, the entry is served without asking your origin. When it runs out, the entry stops being served, whether or not anything invalidated it.

The lifetime comes from one of two sources:

- **Your origin's `Cache-Control` response header.** Works for every response, plain HTTP and GraphQL alike, and is the only source for a plain HTTP response. See from the origin header.
- **Your schema's `@cacheControl` annotations.** Only for a GraphQL operation, and only when the origin header says nothing about lifetime. See from the schema.

How long is one of the two questions a cache entry answers. The other, who else may be served it, is partitioning.

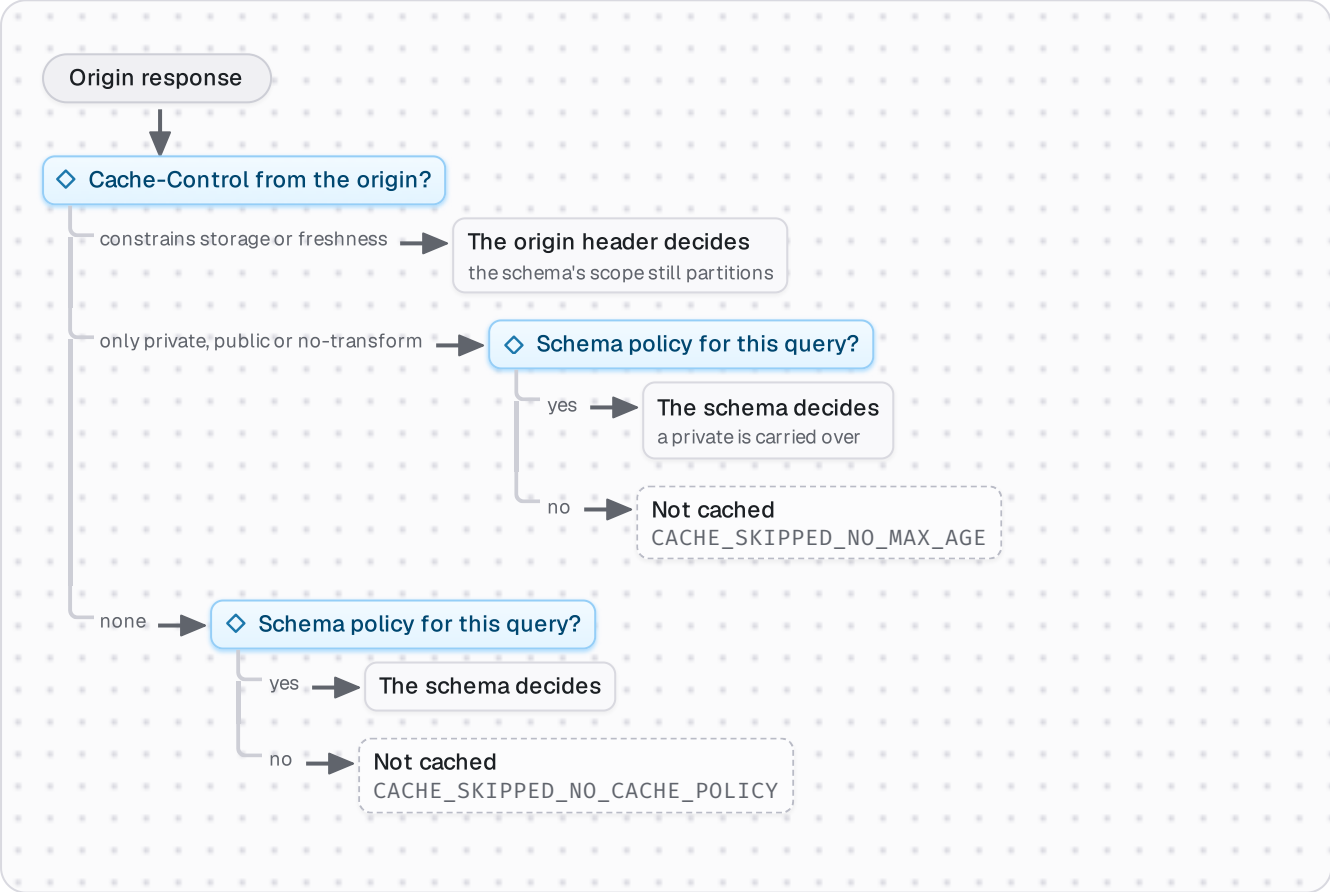
Ways to control this

Through	What it controls
Your origin	`max-age`, `s-maxage`, `stale-while-revalidate`, `no-store` and `no-cache` in the `Cache-Control` response header, for plain HTTP and GraphQL.
Your GraphQL schema	`maxAge` and `swr` on `@cacheControl`, when your origin's header says nothing about lifetime.
`gpilot.toml`	Whether a header without a lifetime yields to the schema (`cache.ignore_origin_header_without_freshness`), the `Cache-Control` your clients receive (`cache.policy.downstream`), and how old an entry one client accepts (client profile `max_age`).
The API, the CLI and the portal	No lifetime setting. To end an entry early, purge it.

Precedence: the origin header decides if it says anything

GraphPilot decides a response's lifetime in this order:

1. **The origin sent a `Cache-Control` header that constrains storage or freshness.** The header decides: its `max-age` or `s-maxage`, its `stale-while-revalidate`, `no-store` and `no-cache`. The schema's `maxAge` and `swr` are not consulted. The schema's **scope still partitions the entry**: a header says how long, a scope says for whom, so a `PRIVATE` or named scope on the selected fields keeps its discriminators.
2. **The origin sent a header that constrains neither.** A header made only of `private`, `public` or `no-transform` says nothing about how long, so it yields and the schema's policy decides. Any `private` it stated is still carried over, so the entry stays partitioned. This applies only when the query selected an annotated field and so resolved a schema policy. Otherwise the bare header decides, and without a lifetime the response is not cached (`CACHE_SKIPPED_NO_MAX_AGE`).
3. **The origin sent no `Cache-Control` at all.** The schema's `@cacheControl` policy decides.
4. **Neither says anything.** The response is not cached (`CACHE_SKIPPED_NO_CACHE_POLICY`).



The four rules above, as a decision.

Every directive the header can carry, and what each one does, is on from the origin header. How the schema's `maxAge` resolves across a query is on from the schema.

Rule 2 exists because a bare `Cache-Control: private` is often infrastructure noise rather than a policy: several hosting platforms add it to every response. `CACHE_POLICY_ORIGIN_HEADER_IGNORED` in your logs says the header yielded and prints it verbatim. Set `cache.ignore_origin_header_without_freshness = false` to turn the yield off, so that the mere presence of an origin `Cache-Control` decides the response.

> ⚠ **An origin header always wins over your schema**
>
> If your origin already sends `Cache-Control` on every response, your `@cacheControl` hints are read and never applied. Apollo Server is the common case: its cache-control plugin sends `Cache-Control: no-store` for operations it considers uncacheable. See Apollo Server. Check what your origin actually sends before relying on schema hints.

The lifetime an entry was stored with is on every hit as `gp-cache-max-age`, with `gp-cache-age` and `gp-cache-remaining` beside it.

Lifetimes in one catalog

A lifetime follows from how fast the data changes and what else ends it. The GraphPilot demo API, a product catalog, states these in its schema:

Data	`@cacheControl`	Why
`Category`	`maxAge: 86400, swr: 604800`	Changes rarely, and a rename runs through `renameCategory`, a mutation that purges it
`Product`	`maxAge: 3600, swr: 3600`	A price changes through `setPrice`, which purges the product, so the hour rarely decides how old a price is
`Product.inventory`	`maxAge: 5`	Stock moves with every order
`Me`	`maxAge: 60, scope: PRIVATE`	One caller's own profile

A response lives as long as the shortest-lived thing it selects. Ask for a product's name and price and the entry is stored for an hour; add its stock and the same product is stored for five seconds:

```
curl -sS -D- -o/dev/null https://demo-api.graphpilot.cloud/graphql \
  -H 'content-type: application/json' \
  -d '{"query":"query ProductWithStock { product(id: \"p01\") { id name price inventory { available } } }"}' \
  | grep -i '^gp-cache'
```

Run it twice within five seconds and the second response, a `HIT`, carries `gp-cache-max-age: 5`; without `inventory` it carries `3600`. So a page that shows a product and its stock is cheaper as two queries, one stored for an hour and one for five seconds. The worked examples on the schema page go through each rule.

When the lifetime runs out

Without a stale window, the entry stops being served. The next request is a `MISS`, goes to your origin, and its response is stored as a new entry.

With a stale window, the expired entry keeps being served for that many seconds more, reported as `STALE`, while a background request refreshes it. Within an edge location, a request handled by a cache server that does not hold the entry can occasionally be a `MISS` instead. See stale responses within a location.

Expiry is not your invalidation strategy

A lifetime answers “how long may this be wrong”, and every answer is a trade: short means a low hit rate and a busy origin, long means users see old data. You do not have to sit on that line. GraphPilot can remove an entry when the data actually changes, which lets the lifetime be long:

- Automatic invalidation purges what a GraphQL mutation touched.
- Surrogate keys let a purge name what changed.
- The purge API covers changes that never went through GraphPilot, such as a database write or a background job.

So the useful question is not “how fresh does this need to be” but **“what could change this data without GraphPilot finding out, and how long can that be wrong for”**. For data only ever written through requests that purge, that window is short by construction and the lifetime can be hours. For data a nightly import touches, it is the import.

> **Permissions are the exception**
>
> A revoked session or a withdrawn permission is not a data change, and nothing purges an entry for it. Until it expires, an entry keyed to one caller keeps being served to that caller. This is the case where a short lifetime genuinely is the control. See revocation.

Changing a lifetime

A new lifetime, and any other policy change (a scope, a `cache.policy` expression), applies to responses stored after the change. Entries already in the cache keep the lifetime they were written with until they expire or are purged. If you shorten a lifetime because it was too long, purge the affected keys as well, or the old entries outlive the change by their own full lifetime.

Related

- From the origin header: every `Cache-Control` directive
- From the schema: `@cacheControl` for GraphQL
- Stale windows: serving past the lifetime while refreshing
- Downstream header: what the client is told
- When a response is cached: the whole store decision

CACHING · HOW LONG

Lifetime from the origin header

Your origin's `Cache-Control` response header is how any origin, plain HTTP or GraphQL, tells GraphPilot how long a response may be kept. For a plain HTTP response it is the only source: no header, or a header without a lifetime, means the response is not cached. For a GraphQL response it outranks your schema whenever it says anything about storage or freshness; see precedence.

```
Cache-Control: max-age=30, s-maxage=600, stale-while-revalidate=60
```

The directives GraphPilot reads

Directive	What GraphPilot does
`max-age=<s>`	The lifetime, unless `s-maxage` is also present.
`s-maxage=<s>`	The lifetime at the edge. Overrides `max-age`, which is left to the browser. Also permission to share a response to a signed-in caller.
`public`	Permission to share a response to a signed-in caller. States no lifetime on its own.
`private`	The entry is partitioned per caller under your `cache.policy.private` expression. Not stored if that does not resolve for the request (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`).
`no-store`	Not stored (`CACHE_SKIPPED_NO_STORE`).
`no-cache`	Not stored (`CACHE_SKIPPED_NO_CACHE_DIRECTIVE`). See below.
`stale-while-revalidate=<s>`	A stale window after the lifetime.
`must-revalidate`, `proxy-revalidate`	The stale window is dropped: the entry is stored for its lifetime only (`CACHE_STORED_WITHOUT_STALE_WINDOW` in your logs). Not accepted as permission to share.
`stale-if-error=<s>`	Parsed, and not acted on. GraphPilot does not serve a stale entry when your origin fails.
`immutable`, `must-understand`	Parsed. They change nothing beyond making the header one that decides.
`no-transform`	Parsed. States no lifetime, so a header made only of it yields to the schema.

Directives are matched case-insensitively, a header sent several times is read as one list, and a directive GraphPilot does not know is ignored, as RFC 9111 asks.

When the header decides, it is checked in this order, and the first rule that refuses ends it:

1. `no-store` refuses.
2. `no-cache` refuses.

3. The lifetime comes from `s-maxage`, otherwise `max-age`. With neither, the response is refused as `CACHE_SKIPPED_NO_MAX_AGE`. This comes before the partitioning check, so a `private` response with no `max-age` reports `CACHE_SKIPPED_NO_MAX_AGE`.
4. Every discriminator the entry is keyed on has to be available: your `cache.policy.private` value for a `private` response, and the names of any scope your schema set for the same query. See who shares an entry.
5. A response to a request that carried a credential is shared only when the header says `public` or `s-maxage`, or names the credential in `Vary`. Otherwise it is partitioned or not stored. See authenticated requests.

The `CACHE_POLICY_ORIGIN_HEADER` log line records that the header decided, carries it verbatim as `cache_control`, and reports `overrode_schema_policy = true` when there was a schema policy it outranked.

`s-maxage`: a different lifetime at the edge than in the browser

`max-age` is addressed to every cache, the browser's included. `s-maxage` is addressed to **shared** caches only, and GraphPilot is one. Where both are present, `s-maxage` decides how long the edge keeps the response and `max-age` is left to the browser, per RFC 9111.

The right answer is rarely the same number for both:

```
Cache-Control: max-age=30, s-maxage=600
```

Ten minutes at the edge, thirty seconds in the browser. The edge is one cache serving everybody, so a long lifetime there absorbs a lot of origin traffic, and you can purge it the moment the data changes. A browser cache is one per person, you cannot reach it, and nothing you do invalidates it before it expires.

Reversed, it says "cache this for the user, not for everyone":

```
Cache-Control: max-age=600, s-maxage=0
```

`s-maxage=0` keeps a response from ever being served fresh from the edge. If you mean that the edge should keep nothing at all, send `no-store`: `s-maxage=0` next to a `stale-while-revalidate` is still served stale.

`no-cache` is refused, not revalidated

`no-cache` permits storing a response but forbids serving it without revalidating with the origin first, on every use. GraphPilot never sends a conditional request and never handles a `304`, so an entry it stored could only be served in breach of that instruction. It does not store one.

Use `max-age` if you want the response cached, and `no-store` if you want it neither stored nor served from a cache.

Headers GraphPilot does not read

GraphPilot reads exactly one response header to decide a lifetime: `Cache-Control`.

- `Surrogate-Control` is ignored. If you come from a CDN that reads it, move its `max-age` into `Cache-Control` as `s-maxage`, which says the same thing to every shared cache.
- `Expires` is ignored. State the lifetime with `max-age` or `s-maxage`.
- `Age`, `Date`, `ETag` and `Last-Modified` are not consulted either. An entry's age comes from the cache itself.

Which responses are refused whatever their `Cache-Control` says (`Set-Cookie`, `Vary: *` and others) is on when a response is cached.

Apollo Server sends its own header

Apollo Server's cache-control plugin computes a `Cache-Control` header from the same `@cacheControl` annotations GraphPilot reads, and sends it on every response: `max-age=N, public`, `max-age=N, private`, or `no-store` for an operation it considers uncacheable. Because an origin header wins, GraphPilot then reads your schema hints and never applies them: the header your server already sends is authoritative.

That changes nothing when the header is what you want. If you would rather GraphPilot decide from the schema, stop Apollo Server from sending the header; see Apollo Server.

Related

- How long an entry lives: precedence between header and schema
- Stale windows
- Authenticated requests: when `public` or `s-maxage` is required
- Downstream header: your header reaches the client unchanged

---

CACHING · HOW LONG

Lifetime from the schema

> **GraphQL only** A plain HTTP response takes its lifetime from the origin header only.

`@cacheControl` lets you state cache lifetimes in your GraphQL schema, the same way Apollo Server's cache-control plugin does. Annotate a type or a field with a `maxAge`, and GraphPilot derives a lifetime for every response that selects it, without any change to your server code.

It is for origins that do not compute a `Cache-Control` header themselves. An origin header that states a lifetime always wins over the schema; see precedence. The same annotation also carries a `scope`, which decides who shares the entry; that half is on scopes in the schema.

Declaring the directive

schema.graphql

```
directive @cacheControl(
    maxAge: Int
    scope: CacheControlScope
    inheritMaxAge: Boolean
    swr: Int
) on OBJECT | INTERFACE | UNION | FIELD_DEFINITION

enum CacheControlScope {
    PUBLIC
    PRIVATE
}
```

Declare the directive in your SDL like this. Every argument, the default `maxAge`, what the deployment build reports and how to rename the directive are on the directive reference. None of the build's findings fails a deployment; they are recorded as diagnostics.

Locations

A hint can be placed on a **type** (`OBJECT`, `INTERFACE`, `UNION`) or on a **field** (`FIELD_DEFINITION`):

- **Type-level hint:** applies to every field in the schema that returns this type, unless that field has its own hint.
- **Field-level hint:** applies to that field only, and always takes precedence over a type-level hint on its return type for the arguments it sets; the rest come from the type, see below.

schema.graphql

```
type Book @cacheControl(maxAge: 60) {
    title: String
}

type Query {
    # Inherits Book's type-level hint: maxAge 60
    book: Book

    # Field-level hint overrides Book's type-level hint for this field only
    freshBook: Book @cacheControl(maxAge: 0)
}
```

A field hint overrides only the arguments it sets

When a field carries its own `@cacheControl` and its return type has one too, the two are merged argument by argument, as in Apollo Server. Each argument the field hint sets wins; each one it leaves out (`maxAge`, `scope`,

`swr`, `inheritMaxAge`) is taken from the type's hint. The return type is the named type under any list and non-null wrappers.

schema.graphql

```
type User @cacheControl(maxAge: 300, scope: PRIVATE) {
    email: String
}

type Query {
    # maxAge 60 from the field, scope PRIVATE from the type
    viewer: User @cacheControl(maxAge: 60)
}
```

`inheritMaxAge` follows Apollo Server too. On the field it takes effect only when the type states no `maxAge`; a type that does state one wins. On the type it carries over to a field that states no `maxAge` of its own. A missing `scope` counts as `PUBLIC` only when neither hint states one.

The merge is done when your schema is built into a deployment, so it applies to a service from its next deployment on.

Resolution order

For every field in the query, GraphPilot resolves an effective `maxAge` using Apollo's own rules, checked in this order:

1. **An explicit `maxAge` on this field's hint always wins.**
2. **Else, if `inheritMaxAge: true` is set and this is not a root field:** take the parent field's already-resolved effective `maxAge`.
3. **Else, if this is a root field or a field returning a composite type** (an object, interface, or union): fall back to the default `maxAge` (`0`, not cacheable, unless you set one, see Default `maxAge`).
4. **Else** (a non-root scalar leaf field with no hint): inherit the parent field's effective `maxAge`.

The response's overall policy is then the **minimum** `maxAge` across every field selected by the query, so the response can only be as fresh as its most restrictive field. Scopes accumulate instead; see how scopes compose.

Worked examples

schema.graphql

```
type Query {
    book: Book
    cachedBook: Book @cacheControl(maxAge: 60)
    reader: Reader @cacheControl(maxAge: 40)
}
```

```
type Book {
    title: String
    cachedTitle: String @cacheControl(maxAge: 30)
}

type Reader {
    book: Book @cacheControl(inheritMaxAge: true)
}
```

Query	Effective `maxAge`	Why
`{ book { cachedTitle } }`	`0`	`book` is an unannotated root field returning a composite type: rule 3 gives it `0`. That `0` is the minimum across the whole response regardless of `cachedTitle`'s own hint.
`{ cachedBook { title } }`	`60`	`cachedBook` has an explicit hint (rule 1). `title` is a non-root scalar with no hint, so it inherits `cachedBook`'s effective value (rule 4). Minimum of the two is `60`.
`{ cachedBook { cachedTitle } }`	`30`	`cachedBook` resolves to `60` (rule 1). `cachedTitle` has its own explicit hint of `30` (rule 1), which is stricter. The minimum wins: `30`.
`{ reader { book { title } } }`	`40`	`reader` resolves to `40` (rule 1). `book` sets `inheritMaxAge: true` with no explicit `maxAge`, so it takes `reader`'s effective value, `40` (rule 2). `title` inherits `book`'s value (rule 4). All three agree: `40`.

In the demo catalog

The GraphPilot demo API runs this schema behind GraphPilot, so each row below can be checked against `gp-cache-max-age` on a `HIT`:

schema.graphql (excerpt)

```
type Product @cacheControl(maxAge: 3600, scope: PUBLIC, swr: 3600) {
    id: ID!
    name: String!
    price: Int!
    inventory: Inventory! @cacheControl(maxAge: 5, scope: PUBLIC)
}

type Category @cacheControl(maxAge: 86400, scope: PUBLIC, swr: 604800) {
    id: ID!
    name: String!
    products: [Product!]! @cacheControl(inheritMaxAge: true, scope: PUBLIC)
}

type Activity @cacheControl(maxAge: 5, scope: PUBLIC, swr: 60) {
    observedAt: String!
```

```
    entries: [ActivityEntry!]! @cacheControl(inheritMaxAge: true, scope: PUBLIC)
}

type ActivityEntry {
    productId: ID!
    at: String!
}

type Query {
    product(id: ID!): Product
    categories: [Category!]! @cacheControl(maxAge: 86400, scope: PUBLIC)
    activity(first: Int = 5): Activity! @cacheControl(maxAge: 5, scope: PUBLIC, swr: 60)
}
```

Query	`gp-cache-max-age`	Why
`{ product(id: "p01") { id name price } }`	`3600`	`product` states no hint and takes `Product`'s, and the scalars inherit it (rule 4).
`{ product(id: "p01") { id name price inventory { available } } }`	`5`	`inventory` states its own `5` (rule 1), and the response takes the minimum.
`{ categories { id name products { id name } } }`	`3600`	`products` asks to inherit the category's day, but `Product` states its own hour, and a type that states a `maxAge` wins over `inheritMaxAge`.
`{ activity { observedAt entries { productId at } } }`	`5`	`ActivityEntry` states nothing, so `inheritMaxAge` takes the feed's `5`. Without it, `entries` would fall back to the default of `0` (rule 3) and nothing would be cached.

The second row also shows how the stale window resolves: `inventory` states no `swr`, so the response keeps `Product`'s hour. Five seconds after it was stored, the entry is served `STALE` while a refresh runs, and a reader can see a stock level that is older than five seconds. If stock must never be served stale, state `swr: 0` on the field: a stated `0` wins the minimum.

A field returning a union or interface

For a field returning a union or an interface, the concrete type is known only once your origin has answered, and the policy is resolved from the query alone. The two arguments are therefore treated differently:

- `scope`**: every member counts.** The scope of every possible type (each member of a union, each object type implementing an interface) joins the field's, as a second field in the same selection would; see how scopes compose. One `PRIVATE` member makes the field private, whichever member the query selects and whatever the field or the union states. A `scope: PUBLIC` on the field or on the union does not override it, and there is no opt-out. A named scope on a member adds the names it partitions on.

- `maxAge`**: the members are not read.** The `maxAge` comes from the field itself or from the union or interface type. When a member states a lower `maxAge` than the field resolves to, the deployment build reports a warning naming the field and those members; the response is still cached for the field's value.

schema.graphql

```
union SearchResult = Book | Account

type Book @cacheControl(maxAge: 300) {
    title: String
}

type Account @cacheControl(maxAge: 30, scope: PRIVATE) {
    email: String
}

type Query {
    search: SearchResult @cacheControl(maxAge: 60, scope: PUBLIC)
}
```

Here `{ search { ... on Book { title } } }` resolves a `maxAge` of `60`, from the field, and is private, because `Account` is: the query names only `Book`, but the origin decides what `search` returns. The build warns that `Account` states `30`, below the `60` of `Query.search`. To cache `search` for at most 30 seconds, state that on the field or on the union.

A member's scope is part of a schema policy, not one of its own: a query that selects no annotated field, as with an unannotated `search` above and nothing else, resolves no schema policy at all, as before. Without an origin header the response is not cached (`CACHE_SKIPPED_NO_CACHE_POLICY`). If the query also selects an annotated field elsewhere, an unannotated `search` counts as the default `maxAge` by rule 3, and its members' scopes still join the response's.

The member scopes are read when your schema is built into a deployment, so the change applies to a service from its next deployment on.

The portal's **Cache rules** and the `schemaAnalysis` of a deployment show this effective scope: a field returning a union or an interface, and the union or interface's own hint, read `private` when any member is private, or the named scope its members add, rather than the scope written on the field.

A hint on a member's own field is still read through its type condition: in `{ search { ... on Book { publisher { name } } } }`, a `@cacheControl` on `Book.publisher`, or on the type `publisher` returns, applies as usual.

A field the deployed schema does not declare

The policy is read from the schema you deployed to GraphPilot, not from the one your origin runs. A query that selects a field that schema does not declare, at any depth, or a fragment on a type it does not know, has no hint to resolve for that selection and no scope to partition it by. Such a response is not stored (`CACHE_SKIPPED_UNRESOLVED_SELECTION`), whatever the rest of the query resolves to and whatever your origin's

`Cache-Control` says, and its policy drops to a `maxAge` of `0` with no stale window. The usual cause is an origin deployed ahead of its schema: deploy the schema to GraphPilot first, as described under deployments.

Seeing what the schema decided

A resolved `maxAge` of zero, which is also the default when your directive declares none, means the response is not cached: `CACHE_SKIPPED_NO_MAX_AGE`. A mutation or subscription is never cached, whatever its hints; see when a response is cached.

When the schema decided, your logs carry the `CACHE_POLICY_SCHEMA` log line with:

- `max_age`: the resolved lifetime,
- `max_age_set_by`: the fields that set it (only those that declared the value rather than inherited it),
- whether the result is private,
- `scope_defaulted`: `true` when a selected hint stated no scope and was read as `PUBLIC`. It is the first thing to check when one caller reports another's data.

Renaming or turning it off

The directive name and its argument names are configurable, and `directives = []` under `[cache_control]` switches schema hints off entirely. See renaming directives.

CACHING · HOW LONG

Stale windows

A stale window is a number of seconds after an entry's lifetime during which GraphPilot still answers from the cache, **while a fresh copy is fetched behind it**. Inside the window the caller gets the stored answer immediately, with no origin round trip, and the response reports `gp-cache: STALE` rather than `HIT`. The refresh runs after the answer has left, and the next caller gets the new copy. Inside an edge location, the stale copy is served by the cache server that holds the entry, so a request that another server handles can occasionally be a `MISS` (see stale responses within a location).

That makes it the cheap half of a hit rate. The lifetime decides how fresh the served copy is *guaranteed* to be; the stale window decides how long you keep answering instantly after that guarantee runs out. Widening it costs nothing in latency and buys back every request that used to wait for your origin at the moment an entry expired.

Where the window comes from

From the same source that decided the lifetime (see precedence):

- **Your origin's header**, as `stale-while-revalidate`:

```
Cache-Control: max-age=60, stale-while-revalidate=86400
```

- **Your schema**, as the `swr` argument of `@cacheControl` (GraphQL only):

  schema.graphql

```
type Query {
    # Guaranteed fresh for a minute. For the day after that, still answered
    # from the cache while a refresh runs behind the response.
    invoices: [Invoice!]! @cacheControl(maxAge: 60, swr: 86400)
}
```

A response with neither goes straight from fresh to a `MISS`.

How the schema resolves a window across a query

A field that states no window does not switch one off. The response's window is the minimum over the fields that *declared* one, and a field that declares none contributes nothing. This is deliberately not the rule `maxAge` follows, where an unannotated field counts as zero and takes the whole query with it.

An unannotated field is a field nobody has said is safe to hold, so `maxAge` treats silence as a refusal. Staleness is not that kind of statement: it moves no audience, the discriminators an entry is keyed on are untouched, and a purge ends staleness the moment something writes. If silence meant zero here, one unannotated wrapper type anywhere in a selection would switch the window off for the whole query, with no symptom but a hit rate that never moves.

A stated `swr: 0` is still a statement, and it wins the minimum like any other number. That is how you switch staleness off for a response on purpose.

What switches a window off

- `must-revalidate` or `proxy-revalidate` **on the origin header.** Both forbid serving a response once it is stale, and GraphPilot has no conditional revalidation. So `max-age=60, stale-while-revalidate=600, must-revalidate` is stored for 60 seconds and never served `STALE`. Your logs record `CACHE_STORED_WITHOUT_STALE_WINDOW` with the window that was dropped.
- **A hard purge.** A purge removes the entry outright rather than aging it. Only a soft purge leaves it to be served stale.

`stale-if-error` is not a stale window in this sense: GraphPilot parses it and does not act on it, so an entry is not served stale because your origin failed. See from the origin header.

`s-maxage=0` with a stale window

`s-maxage=0` keeps a response from ever being served fresh from the edge, and it combines with `stale-while-revalidate` in a way that is specified rather than a quirk. RFC 5861 lets a cache serve a response "after it becomes stale, up to the indicated number of seconds", with no exemption for *why* it became stale. So `s-maxage=0, stale-while-revalidate=60` reads as "stale to a shared cache immediately, and you may serve it stale for a minute", and GraphPilot does exactly that: `STALE` for the window.

If what you mean is that the edge should keep nothing, `Cache-Control: no-store` says so with no interactions at all.

Stale windows and request collapsing

When a popular entry expires without a window, the next requests for it wait on one origin request: request collapsing sends one and holds the rest. With a window, a request that reaches the entry inside the window is served it immediately, and the refresh runs behind it. A request that does not reach it goes to your origin, as described in stale responses within a location.

The background response goes through the whole store decision like any other. If it is not storable, for example because your origin now answers with `Set-Cookie`, the stale entry is not replaced, and the refusal code appears in your logs rather than on a response.

Stale responses within a location

An edge location (a Fastly POP) is not one cache but several cache servers. Each cached entry has one server in the location that holds it. While the entry is fresh, the location's other servers can answer from it too, as a `HIT`. Once its lifetime has run out, the stale copy is served by the server that holds the entry.

In practice, most requests inside the window are `STALE`. Occasionally a request that another server handles finds no entry: it is a `MISS`, goes to your origin, and stores a fresh copy that the following requests are served from. This is expected behavior, not an error. It never serves wrong or older data; it costs one origin request per such miss.

An example, for an entry stored with `maxAge: 60, swr: 3600`, with all four requests at the same location:

Time	Server	`gp-cache`	`gp-cache-age`	Insights **Lookup**	What happened
0 s	A	`MISS`		No entry	Your origin answered; the entry was stored.
0.2 s	B	`HIT`	`0`	Entry found	Fresh, so server B could answer from it.
93 s	A	`STALE`	`93`	Entry found	Past its lifetime, inside the window. Refreshed in the background.
93 s, 16 ms later	B	`MISS`		No entry	Server B did not find the stale entry. Your origin answered; a fresh copy was stored.

To recognize it, compare the requests in Insights: the request detail panel's **Cache** tab shows the status and the **Lookup** (**Entry found** or **No entry**), and its **Overview** tab shows the edge location and the **Server** within it. A `MISS` with **No entry** at the same location and within a few seconds of a `STALE`, on a different **Server**, is this case. The detail panel shows these fields on every plan.

With shielding

With shielding (paid plans, not Glider (Free)), such a miss is sent to the shield location instead of your origin. If the shield still holds the entry, fresh or inside its stale window, it answers and your origin is not contacted. The edge location still reports what it saw itself:

- **The shield's copy was fresh:** `gp-cache: MISS`, **Shield result** `HIT`. The edge location stores the copy for the lifetime it has left.
- **The shield's copy was stale:** `gp-cache: PASS` with `gp-cache-reason: CACHE_SKIPPED_SHIELD_RESPONSE_NOT_STORABLE`, **Shield result** `STALE`. The edge location does not store a stale copy; the shield refreshes its own in the background.

Shielding does not change how a location's servers share entries, and the shield is itself a location with several servers: a request that reaches a shield server without the entry after its lifetime is a miss there and goes on to your origin. Shielding reduces how often your origin is asked; it does not guarantee a `STALE` at the edge.

If origin load in the window matters

- **Switch on shielding**, so most of these misses are answered by the shield instead of your origin.
- **Use a longer lifetime** (`maxAge` or `s-maxage`). While an entry is fresh, every server in the location answers from it, so fewer requests fall into the window at all. With invalidation on write, a long lifetime does not mean old data.

Choosing a window

> **A long window is cheapest where you invalidate on write**
>
> A window only ever applies when nothing has written, because a purge removes the entry. So if your writes invalidate by surrogate key, a window of hours or a day is close to free, and the number to think about is how stale an answer may be when *nothing at all* has happened. Where data changes without any write of yours to hang a purge on, keep the window short: that is the case a clock is the only thing bounding.

A window also extends how long an entry keyed to one caller may be served after that caller's access changed: the bound is the lifetime plus the window. See revocation.

A private or named-scope response keeps its window in GraphPilot's cache, but the window is not announced to the client. See downstream header.

CACHING · HOW LONG

Downstream header

GraphQL only A plain HTTP response reaches the client with your origin's Cache-Control unchanged.

The precedence rule decides whether GraphPilot **stores** a response in its own cache. Separately, `cache.policy.downstream` decides whether GraphPilot **writes a `Cache-Control` header derived from your schema** on the response it sends back to the client, similar to what Apollo Server's cache-control plugin does.

The setting only ever synthesizes a header for a response your schema's `@cacheControl` policy decided. It never rewrites, removes or replaces a `Cache-Control` header your origin sent:

- **Your origin sent `Cache-Control`.** The client receives that header unchanged, in every mode, `off` included. A `private, max-age=60` from your origin reaches the client as `private, max-age=60`.
- **Your origin sent none, and the schema decided.** This is the case the setting is for. With `off`, the client gets no `Cache-Control`. With `if_cacheable` or `always`, GraphPilot writes one from the schema policy, as described below.

These are independent of what GraphPilot stores. It can cache a schema-decided response without telling the client, or a downstream CDN, anything about it, and the header it writes does not by itself change what GraphPilot stores.

Configuration

gpilot.toml

```
[cache.policy]
downstream = "off" # "off" | "if_cacheable" | "always"
```

`off` (default)

GraphPilot never synthesizes a `Cache-Control` header. An origin header still passes through unchanged. This is the default so that a service without the setting keeps sending exactly what its origin sent.

`if_cacheable`

GraphPilot writes a `Cache-Control` header for a schema-decided response only when the schema policy is actually cacheable, that is, it has a positive `maxAge`. A `Cache-Control` header from your origin is forwarded as it is, as in every mode.

`always`

GraphPilot writes a `Cache-Control` header whenever a schema policy resolved, even when that policy is not cacheable. If no policy resolved, no header is written, in `always` mode either: there is nothing to reflect, so GraphPilot never fabricates a `public` guess.

A query that selects no field carrying a `@cacheControl` hint, directly or through its return type, resolves no schema policy at all. Neither mode writes a header for it. The client then sees only what your origin sent, if anything.

A query that selects a field your deployed schema does not declare resolves a policy with a `maxAge` of `0` and no stale window, however long its other fields could be kept (see a field the deployed schema does not declare).

`if_cacheable` writes no header for it, and `always` reflects that `0`, so the header never advertises a lifetime GraphPilot itself did not store the response for.

What gets written

For a response your schema's policy decided, the header never claims to be more permissive than the policy GraphPilot used for its own storage decision:

Schema policy	Header value
No policy resolved at all	No header written, in any mode
Private, or any named scope	`Cache-Control: private`
Public, with a vary rule that is not `scope_only`	`Cache-Control: private`
Public, with a known `maxAge`	`Cache-Control: max-age=<n>, public`
Public, with a `maxAge` and an `swr`	`Cache-Control: max-age=<n>, stale-while-revalidate=<s>, public`
Public, with no known `maxAge`	`Cache-Control: public`

A stale window is written only beside a `max-age`, because that is what it qualifies: it says how long past the fresh window a stale answer stays acceptable, and with no window to be past it describes nothing. A stated `swr: 0` is left off for the same reason, so the token that appears always means a window somebody asked for.

When the schema policy is **private** or names a named scope, GraphPilot writes `Cache-Control: private` in both non-`off` modes, with no freshness of any kind on it, so that a downstream CDN or shared cache does not store a response that is meant for one user or one group only. A private policy's own `swr` still governs GraphPilot's cache; it is simply not announced downstream. See who shares an entry for how a response ends up private in the first place.

The same holds for a public policy while any `[[cache.policy.vary]]` entry that is not `scope_only` is configured. Such an entry splits the edge's entry by a value GraphPilot derives itself, so no `Vary` header could tell a downstream cache about the split, and the response is announced as `private`. An entry marked `scope_only` partitions only named scopes and leaves a public policy public.

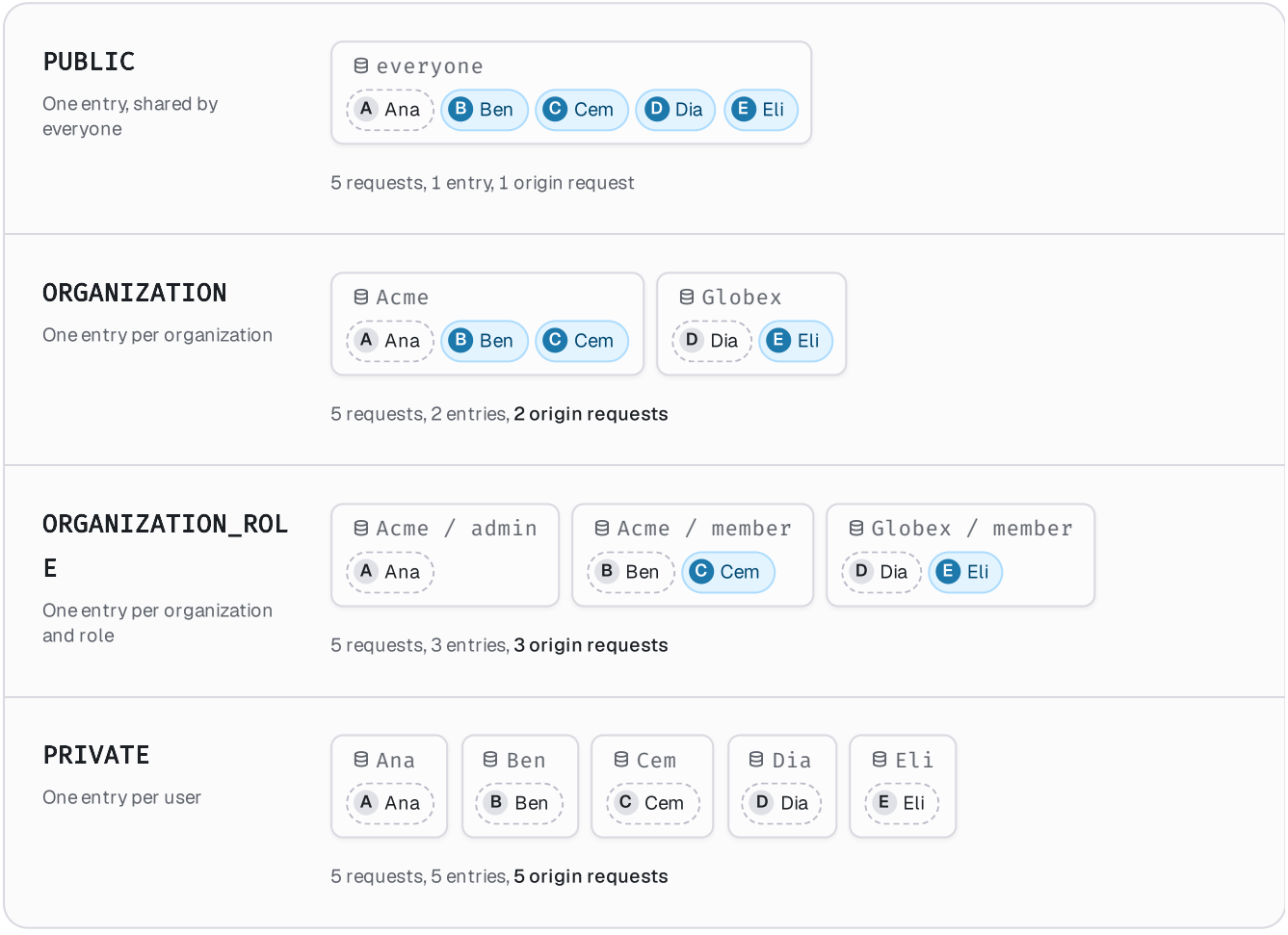
> **Note**
>
> The downstream header switch only affects what the **client** sees. It has no effect on GraphPilot's own cache storage decision, which is governed entirely by the precedence rule.

Who shares an entry

A cached response is served to everyone who sends the same request **and** belongs to the same audience. Partitioning is how you say what that audience is. Ask one question of every cached answer:

> **Who gets exactly the same answer?**

The answer is almost never “only this one person”. A price list is the same for everybody. A deployment list is the same for everyone in an organization. A billing page is the same for every admin in that organization. Each wider audience is one entry serving more callers, which is what keeps a hit rate up. Choose the widest audience that is still correct.



Five people in two organizations send the same request on a cold cache. The narrower the audience, the more entries, and the more requests reach your origin.

Ways to control this

Through	What it controls
Your origin	`private`, `public` or `s-maxage` in `Cache-Control` (authenticated requests), and the `Vary` header.

Through	What it controls
Your GraphQL schema	The `scope` of `@cacheControl`: `PUBLIC`, `PRIVATE`, or a named scope such as an organization.
`gpilot.toml`	`[cache.policy]`: `public`, `private`, `bypass` and vary rules, built from the claims your auth providers verify.
The API, the CLI and the portal	No partitioning setting. To take access away before an entry expires, revoke, then purge.

The ladder

From widest to narrowest. Each rung keeps a separate entry per distinct value, so each costs hit rate against the one above it. Entries are also kept per edge location.

Audience	One entry per	Hit rate	Configure it with
Everybody (`PUBLIC`)	cache key	Highest: every caller warms the entry for every other caller	Nothing: a response is shared unless something narrows it. From a signed-in caller, your origin has to say `public` or `s-maxage`; see authenticated requests.
An organization	key and organization	One miss per organization, then shared by every member	A vary rule on the organization claim; for GraphQL, a named scope such as `ORGANIZATION` on the types it is true of
An organization and role	key, organization and role	One miss per role in each organization	Two vary rules, or one combining both; for GraphQL, a named scope such as `ORGANIZATION_ROLE`
A tenant or a language	key and each value	Divided by the number of values, on every request the rule applies to	A vary rule (`auth:` for a tenant, the primary language of `Accept-Language` for a language, never the raw header; see caching per language), or your origin's `Vary` header
One caller (`PRIVATE`)	key and caller	Lowest: follows how often the *same* caller repeats a request, and the first request of each caller always misses	Your origin's `Cache-Control: private`, or `scope: PRIVATE` in the schema, plus a `private` expression that identifies one caller

Each rung describes a service configured for that rung. By default a vary rule partitions every response, so a service that also configures a role rule for `ORGANIZATION_ROLE` keys its `PUBLIC` and `ORGANIZATION` entries on the role too. Mark the rule `scope_only` and each rung holds on its own for the responses your schema decides: a named scope, and `PUBLIC`, then partition only on the rules they list. Configure only the rules some data needs; partitioning your cache shows the effect with numbers.

Rungs combine. An entry keyed on organization and language is served only to callers who match both, and a query that selects one organization-wide field and one private field is private as a whole.

`PRIVATE` **is the last rung, not the default for anything signed in.** Moving a type from `PRIVATE` to an organization scope turns one entry per member into one per organization. Reserve `PRIVATE` for what really differs per person: a user's own profile, their inbox, rows only they own.

Three rungs in one service

The GraphPilot demo API, a product catalog served at `https://demo-api.graphpilot.cloud`, puts three rungs into one schema:

Data	Who gets the same answer	Rung
`Product`, `Category`, `Review`	Every visitor: a price is the same for everybody	`PUBLIC`
`Report` (revenue and orders)	Everyone in one organization with the same role, because an admin may see figures a viewer may not	Organization and role, as the named scope `ORGANIZATION`
`Me` (the caller's profile and favorites)	One person	`PRIVATE`

schema.graphql (excerpt)

```
enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION @vary(names: ["x-gp-organization", "x-gp-role"])
}

type Product @cacheControl(maxAge: 3600, scope: PUBLIC) { id: ID! name: String! price: Int!
}
type Report  @cacheControl(maxAge: 300, scope: ORGANIZATION) { revenue: Int! orders: Int! }
type Me      @cacheControl(maxAge: 60, scope: PRIVATE) { id: ID! role: String! }
```

gpilot.toml

```
[cache.policy]
private = 'auth:demo.sub'

[[cache.policy.vary]]
name       = "x-gp-organization"
value      = 'auth:demo.org_id'
scope_only = true

[[cache.policy.vary]]
name       = "x-gp-role"
value      = 'auth:demo.role'
scope_only = true
```

Four callers ask for the same report. The demo issues its own tokens, so the request can be run as written:

```
mint() {
  curl -sS https://demo-api.graphpilot.cloud/auth/token \
    -H 'content-type: application/json' \
    -d "{\"sub\":\"$1\",\"org\":\"$2\",\"role\":\"$3\"}" | jq -r .token
}

for token in "$(mint user-ana org-acme admin)" "$(mint user-ben org-acme admin)" \
             "$(mint user-ben org-acme viewer)" "$(mint user-dia org-globex admin)"; do
  curl -sS -D- -o/dev/null https://demo-api.graphpilot.cloud/graphql \
    -H 'content-type: application/json' \
    -H "authorization: Bearer $token" \
    -d '{"query":"query Numbers { reports(range: MONTH) { organizationId role revenue }
}"}' \
    | grep -i '^gp-cache:'
done
```

```
gp-cache: MISS     Ana, admin at Acme, fills the entry
gp-cache: HIT      Ben, admin at Acme, is served Ana's entry
gp-cache: MISS     Ben as a viewer is a different bucket
gp-cache: MISS     Dia is in a different organization
```

Each response says only its `gp-cache` line; the notes on the right are added here. Ben's second request shows why the role is in the key: keyed on the organization alone, Ben the viewer would have been served the admin's figures, and no permission check would have run. A `me` query sent by the four is four entries.

A product query sent by all four is one entry, because `PUBLIC` names no rule and both rules are `scope_only`: Ana and Dia asking for the same product are one miss and one hit, and an anonymous visitor is served the same entry. Without `scope_only`, the product query would be stored once per organization and role as well, which is the cost described under the ladder.

Choosing a rung

- **Preferences**, such as a language, a country or a device class, are chosen by the caller, and a caller who picks the wrong one harms nobody. A `header:` or `query:` source is fine.
- **Groups**, such as an organization, a customer or a role, are security boundaries. Derive them from a verified `auth:` claim, never from a header the caller sets.
- **One caller** is identified by a verified claim such as `sub`, never by a group id.
- **A value that names a person**, such as a user id or an email address, can be wrapped in `pseudonymize`. The partition stays exactly as narrow, because one person always derives the same pseudonym, but the entry is partitioned on a 43-character keyed value instead of the id, and a `plain` origin receives that value too.

For example, a per-user vary rule that keeps the organization readable, for a named scope to list in its `@vary`:

gpilot.toml

```
[[cache.policy.vary]]
name       = "x-gp-organization-user"
value      = "auth:demo.org_id + ':' + pseudonymize(auth:demo.sub)"
scope_only = true
```

Ben at `org-acme` gets an entry of his own, as under `PRIVATE`. Insights records his segment as an `hmac-sha256:` digest under the default `telemetry.vary_values = "pseudonymized"`, and as `8:org-acme:43:` followed by his pseudonym under `vary_values = "clear"`: the organization readable, `user-ben` nowhere.

The trust rules behind this are on identifying callers.

> ⚠ **A group scope is an assertion about authorization**
>
> A cache hit never reaches your origin, so no permission check runs on it. Scoping data to an organization says that everyone in it may see all of it. If some members may see less, key on what separates them too, or use `PRIVATE`. See is your field really organization-uniform?

Which bucket an entry goes into

Every stored response goes into exactly one bucket, and the bucket is part of what a later lookup matches on:

Bucket	When
Private	The origin header said `private`, a selected field is scoped `PRIVATE`, or a signed-in caller's unmarked response was moved here. Partitioned by your `cache.policy.private` expression.
Named	The schema scope named its own audience. Partitioned by the vary rules the scope lists, and by every other configured rule not marked `scope_only`. When your origin's `Cache-Control` decided the lifetime, every configured vary rule applies, `scope_only` or not.
Public	Everything else, including a schema hint that states no scope. Partitioned by your `cache.policy.public` expression, if you configured one.

A response that selected fields from more than one of these is keyed on all of them at once, and is served only to a caller who matches every one. On top of the bucket, the entry keys on whatever your origin's `Vary` names.

Every configured vary rule keys the entry too, whether or not anything names it, except a rule marked `scope_only` on a response your schema decided: that rule keys it only when its bucket names it above. A response whose lifetime your origin's `Cache-Control` decided keys on **every** configured rule.

A response that cannot be placed is not stored rather than shared: a `private` or named-scope response whose discriminator is missing for the request reports `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`.

Where to go next

- Scopes in the schema: `PUBLIC`, named scopes and `PRIVATE` on your GraphQL types.
- Identifying callers: the `private` expression, vary rules, your origin's `Vary`, and where a value may come from. Works for plain HTTP and GraphQL.
- Authenticated requests: what happens when a signed-in caller's response does not say who may share it.
- Revoking access: taking access away before a cached entry expires.

CACHING · WHO SHARES AN ENTRY

Scopes in the schema

> **GraphQL only** For plain HTTP, partition with vary rules on identifying-callers.

The `scope` argument of `@cacheControl` says who may be served the entry a response is stored in. You write it on the types and fields it is true of, and GraphPilot works out the audience of each response from what the query selected.

Scope	Who shares the entry
`PUBLIC`, or no `scope` at all	Every caller
A named scope, such as `ORGANIZATION`	Every caller with the same values for the vary rules it lists
`PRIVATE`	Every caller with the same `cache.policy.private` value, which should be one person

`PUBLIC` is the default

An annotation with no `scope` is `PUBLIC`, as in Apollo Server. A schema written for Apollo Server's plugin caches the same way behind GraphPilot: `@cacheControl(maxAge: 300)` on its own stores one entry that every caller is served, whether or not the request carried a credential.

schema.graphql

```
type Query {
    # The same answer for everybody. No scope written, so PUBLIC: stored once.
    publicHoliday: Holiday @cacheControl(maxAge: 3600)

    # Per caller, so say so: one entry per caller, and only if
    # cache.policy.private resolves for the request.
    viewer: User @cacheControl(maxAge: 60, scope: PRIVATE)
}
```

That makes the default the line to review. A field whose answer depends on who asks has to say so; left unscoped, it is shared. A field hint that states no scope keeps its return type's scope; see a field hint overrides only the arguments it sets. The `CACHE_POLICY_SCHEMA` log line's `scope_defaulted = true` marks responses shared by default.

Declaring a named scope

Most real data lives between `PUBLIC` and `PRIVATE`. A deployment list is identical for every member of an organization. Scoped `PRIVATE`, five colleagues opening the same page produce five entries and five origin requests; a named scope stores it once per organization.

You add a value to your own `CacheControlScope` enum and name the vary rules it partitions on. The schema names the audience; your service configuration defines what each name is derived from.

schema.graphql

```
directive @vary(names: [String!]!) on ENUM_VALUE

enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION      @vary(names: ["x-org"])
    ORGANIZATION_ROLE @vary(names: ["x-org", "x-role"])
}

type Deployment @cacheControl(maxAge: 300, scope: ORGANIZATION) {
    id: ID!
    createdAt: String!
}
```

gpilot.toml

```
[[cache.policy.vary]]
name       = "x-org"
value      = 'auth:idp.org_id'
scope_only = true

[[cache.policy.vary]]
name       = "x-role"
value      = 'auth:idp.role'
scope_only = true
```

Every response that selects a `Deployment` is now stored for five minutes and served only to callers whose verified token carries the same `org_id`. Both rules are marked `scope_only`, so `ORGANIZATION` partitions only on the rules it names: configuring `x-role` for a different scope does not key this entry on the role too, and it stays stored once per organization. Without `scope_only`, every configured rule partitions every response, and this

entry would be stored once per organization and role. The names in `@vary` are header names, matched case-insensitively against your `[[cache.policy.vary]]` entries.

> **A name with no entry behind it gives you a cache that never fills**
>
> This is the mistake to know about before you write your first named scope, because nothing about it looks broken. `@vary(names: ["x-org"])` with no `[[cache.policy.vary]]` entry named `x-org` is not an error that stops your deployment, and it does not silently share the response either. It refuses to store it, on every single request, forever.
>
> The refusal reports `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR` on `gp-cache-reason`, and the log line distinguishes a name the request did not carry from a name nothing configures. The deployment build also reports it, at the annotation that named it. See what the deployment build reports.

Three rungs in one service runs this shape against a live service with four tokens: two colleagues, one of them again as a viewer, and someone from another organization.

A schema that declares only `PUBLIC` and `PRIVATE` needs no `@vary` anywhere and behaves exactly as before. Moving a type from `PRIVATE` to a named scope only ever reduces the number of entries. The exact definition of `@vary` and its rules are on the directive reference, which also covers renaming it.

How scopes compose across a query

`scope` is a single value per type or field. What accumulates across the fields a query selects is the **set of names they partition on**, unioned:

Scope	Partitions on
`PUBLIC`, or no `scope`	nothing
`PRIVATE`	your `cache.policy.private` value
`ORGANIZATION`	`x-org`
`ORGANIZATION_ROLE`	`x-org` and `x-role`

The table assumes every rule is marked `scope_only`, as in the example above. A rule without it keys every response on top of what its scope names, `PUBLIC` included; see vary rules.

Adding a name to an entry's key narrows the set of requests it can answer, so a union of names is an intersection of audiences. A query selecting one `ORGANIZATION` field and one `PRIVATE` field is keyed on both and served only to a caller who matches both, which is one person. One `PRIVATE` field anywhere in the selection makes the whole response private. `PUBLIC` contributes nothing, so it can never widen what a sibling field narrowed. A field returning a union or an interface counts the scope of every member as well, whichever one the query selects; see a field returning a union or interface.

When your origin sends `Cache-Control` too, the header decides how long, and the scope's names still key the entry. A request a named scope partitions carries a token, so the header has to say the response may be shared:

a `Deployment` scoped `ORGANIZATION` behind an origin answering `Cache-Control: s-maxage=60` (or `max-age=60, public`) is stored for 60 seconds. On this path every configured vary rule keys the entry as well, so with `x-role` configured as above it is stored once per organization and role. A bare `max-age=60` does not say the response may be shared: it is filed per caller under `cache.policy.private`, or not cached, as authenticated requests describes. The rule below applies on this path as well.

The runtime rule: only names GraphPilot derives itself

A name counts for a request only when both hold:

1. **GraphPilot owns the name.** A `[[cache.policy.vary]]` entry defines it, or it is in the reserved `gp-` namespace GraphPilot injects its own buckets into.
2. **The header is present** on the request.

Either failure refuses the store with `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`; the response still reaches your client in full. The first condition is what makes the second trustworthy: a configured name cannot be forged, because GraphPilot removes whatever the client sent under it from the incoming request and re-derives the value from your expression. A caller whose token carries no `org_id` therefore gets a response that is not stored, never somebody else's entry.

That guarantee ends where your expression reads from the caller. A scope standing for an organization, a customer, a role or a permission has to read from `auth:`; see a value is only as trustworthy as its source. An expression that resolves to the empty string satisfies the rule; see an empty value is a value.

What the deployment build reports

The runtime refusal keeps the cache safe but is silent: you would learn about a misspelled name from a hit rate of zero. So the deployment build reports what it can see, in the portal under **Schema → Findings** and **Config → Findings** (see findings) and in the output of `gpilot deploy`:

The deployment build found	What to do
A `@vary` name no `[[cache.policy.vary]]` entry defines	Add the entry.
A `@vary` name inside the reserved `gp-` namespace	Rename it, and the entry with it. A `gp-` name is treated as GraphPilot's own, so it is not checked against your configuration, and `pass_to_origin` cannot work for it.
A `@vary` name whose entry is not a legal HTTP header name	Correct the entry's name. GraphPilot skips such an entry, so the header is never derived.
A `[[cache.policy.vary]]` entry whose `name` is not a legal HTTP header name, referenced or not	Correct it. The entry is dead on every path.

An entry no schema references is **not** reported: it may exist for your origin, which receives the value through `pass_to_origin`. Such an entry still partitions every response, or, marked `scope_only`, every response whose lifetime your origin's `Cache-Control` decided.

These findings are advisory and do not fail a deployment. Treat one as the reason your cache is empty, not as a warning you can defer.

A scope value that is neither `PUBLIC` nor `PRIVATE` and carries no usable `@vary` (missing, an empty `names` list, or not a list of strings) **resolves to** `PRIVATE` and raises a finding. Private is the narrowest bucket the author could have meant, so this cannot widen the audience. A misspelled value (`scope: ORGANISATION` against an enum that does not declare it) is rejected by ordinary GraphQL validation first.

Is your field really organization-uniform?

`scope: ORGANIZATION` asserts that the response depends on nothing narrower than the organization. It is the one assertion GraphPilot cannot check for you.

> **A cache hit never reaches your authorization**
>
> Authorization runs at your origin. A cache hit does not reach your origin. So if **anything** about who may see these rows is decided per user, the first caller who passes that check fills the entry and every later member of the organization is served it without the check running at all.

Belonging is rarely the only condition. An administrator and a read-only member, or someone whose access to one project was revoked last week, see different amounts of the same organization. A deployment list every engineer may read and a member without deployment access may not, scoped `ORGANIZATION`, is filled by the first engineer and then read by that member. Fields that are blanked or redacted for some members have the same shape.

Write down everything the visibility of these rows and fields depends on, and key the scope on every one of them. There are two ways to spell organization plus role:

- **Separate names**, `@vary(names: ["x-org", "x-role"])`, as above. A token missing either claim produces no entry at all, so this shape fails closed. The audience is composed at each annotation, though, and naming only `x-org` on a type whose rows depend on a permission is a leak nothing reports.
- **One combined name** whose rule joins both claims (`auth:idp.org_id + ":" + auth:idp.role`, see vary rules), referenced as `@vary(names: ["x-audience"])`. The audience is defined once and a schema author cannot under-specify it, and `pass_to_origin` can hand your origin the same value. A token missing either claim derives nothing, with `+` and `&&` alike, so this shape fails closed as well.

Prefer the combined name when the audience is fixed and your origin needs it too. Prefer separate names when different types need different subsets.

If visibility depends on the identity itself, a row owned by one user, then `PRIVATE` is the only correct scope. A role bucket would be the same trap one level up.

A per-user scope that keeps the organization readable

`PRIVATE` keys on `cache.policy.private`, and Insights records that value only as a keyed digest, so per-user traffic cannot be grouped by organization there. A named scope can be per user and still show the organization,

by joining the organization claim with a pseudonym of the user:

schema.graphql

```
enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION_USER @vary(names: ["x-org-user"])
}

type Me @cacheControl(maxAge: 60, scope: ORGANIZATION_USER) {
    id: ID!
    favorites: [Product!]!
}
```

gpilot.toml

```
[[cache.policy.vary]]
name       = "x-org-user"
value      = "auth:idp.org_id + ':' + pseudonymize(auth:idp.sub)"
scope_only = true

[telemetry]
vary_values = "clear"
```

Every caller still gets an entry of their own, because the user is part of the value. For a caller in `org-acme`, Insights shows the segment as `8:org-acme:43:` followed by the pseudonym: the organization in clear, then the user as 43 characters that group one person's requests without naming them. The length prefixes are there because the value combines two sources; see vary rules.

	`auth:idp.sub` in the rule	`pseudonymize(auth:idp.sub)` in the rule
The rule derives, and the entry is partitioned on	`8:org-acme:8:user-ben`	`8:org-acme:43:` and Ben's pseudonym
Insights, `vary_values = "pseudonymized"` (default)	`hmac-sha256:` and 16 hex digits	`hmac-sha256:` and 16 hex digits
Insights, `vary_values = "clear"`	`8:org-acme:8:user-ben`	`8:org-acme:43:` and Ben's pseudonym

Entries and hit rate are the same in both columns: one person always derives the same pseudonym. Under the default the organization is no more readable than under `PRIVATE`, so the setting is what makes this scope worth having. Every place `pseudonymize` is accepted, and what each setting records, is on keeping identifiers out of Insights.

- `scope_only` **is what keeps this to** `Me`**.** Without it the rule partitions every response, so every product in the catalog would be stored once per signed-in user.
- `vary_values = "clear"` **applies to every rule of the service**, not only this one. Check that no other rule's value is something you would not store in clear, and read storing values in clear is your choice.
- **It fails closed.** A token without `org_id` or `sub` derives nothing, so the response is not stored.

The same scope beside an organization and a role scope, end to end, is in partitioning your cache.

Why the definition lives in the configuration

The annotation **references** a vary rule by name and carries no expression of its own, on purpose. Your origin can already partition on a derived header through its `Vary` header, so an axis defined in the SDL as well would have two definitions that can drift apart silently. And an expression that decides who shares an entry is a security control: in `gpilot.toml` it is reviewed with your service configuration, not written into a schema assembled from several teams' subgraphs.

CACHING · WHO SHARES AN ENTRY

Identifying callers

The cache policy answers one question: **two requests arrive, may they share an answer?** Nothing about a URL or a GraphQL document says whose data it returns, so only context the request carried can tell two callers apart. The cache policy, in `gpilot.toml`, is where you say which context that is. It works for plain HTTP and GraphQL alike, in three parts that build on each other:

1. **Providers** verify a credential and make its claims readable.
2. **Expressions** derive a value from those claims, or from headers, cookies and query parameters.
3. **Policy fields** decide what that value does: partition private entries (`private`), partition entries by a group or a preference (`[[cache.policy.vary]]` and `public`), or skip the cache (`bypass`).

Where each of these sits among the other ways to partition is on who shares an entry.

> ⚠ **Caution**
>
> Getting this wrong is how one caller sees another's data. GraphPilot fails closed wherever it can, but nothing can check that the value your expression returns identifies one person rather than a group. Read `private` and where a value comes from before you rely on it.

Authentication providers

gpilot.toml

```
[auth.providers.idp]
type = "jwt"
```

```
allowed_algorithms = ["RS256"]
jwks_url = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers = ["https://idp.example.com/"]
allowed_audiences = ["my-api"]
```

A provider verifies a credential on the incoming request and makes its claims addressable as `auth:<provider>.<claim>`. The block name (`idp` above) is what an expression refers to. It runs on every request, cache hits included. Set `allowed_issuers` and `allowed_audiences`: a claim partitions the cache only as well as the provider checks whose token it is, and a deployment warns about a provider that leaves either out. See authentication for the provider types and what happens to a request whose token does not verify.

Expressions

The `private`, `public` and vary `value` fields are policy expressions over four sources. They combine with `||` (the first value that resolves), `&&` and `+` (concatenate; in these three fields every part is required, and a value combining two or more sources keeps each one's boundary, see combining sources in a cache key), comparisons, parentheses, functions such as `lower()` and `split()`, and string literals.

gpilot.toml

```
[cache.policy]
private = 'auth:idp.sub || cookie:session_id'
```

A value is only as trustworthy as its source

Source	Reads	Who decides the value
`auth:<provider>.<claim>`	A claim from a verified credential, nested paths allowed (`auth:idp.org.id`), and a quoted step for a URL-named claim (`auth:idp.'https://acme.example/org_id'`)	**GraphPilot.** Signature, expiry and algorithm were checked before any expression ran, so the caller cannot choose what a claim says.
`header:<name>`	A request header	The caller
`cookie:<name>`	A cookie	The caller
`query:<name>`	A URL query parameter	The caller

A caller-controlled source is right for a preference: `header:accept-language` partitions by something the caller is entitled to choose, and a wrong choice harms nobody.

Never for private, and never for a boundary

`private` asserts that an entry **belongs to one specific caller**. A value the caller picked cannot establish who they are: one person can send somebody else's value and land in their entry, and a cache hit never reaches your origin, so no authorization check runs to catch it.

The rule: any value carrying a security boundary (who the caller is, which organization they belong to, what they may see) must derive from `auth:`. Reserve `header:`, `cookie:` and `query:` for preferences. GraphPilot does not check this for you: a `header:`-derived `private` resolves to a value, and a value is all the cache needs.

gpilot.toml (correct)

```
[cache.policy]
private = 'auth:idp.sub'

[[cache.policy.vary]]
name  = "x-org"
value = 'auth:idp.org_id'
```

gpilot.toml (broken)

```
[cache.policy]
private = 'header:x-user-id'

[[cache.policy.vary]]
name  = "x-org"
value = 'header:x-org-id'
```

In the broken version, sending `x-user-id: <a colleague's id>` is enough to be served that colleague's cached response, and `x-org-id` works the same way across organizations.

A session cookie is the narrow exception, and still weaker

`cookie:session_id` identifies one caller only while the cookie is an unguessable secret your origin issues and validates, a property GraphPilot cannot see, so prefer a claim wherever your callers carry a token. A cookie or header holding a plain identifier (a user id, a customer name, a role) is never in this category.

An empty value is a value

An expression that resolves to **nothing** (its source is missing) leaves the caller unplaced: a `private` response is not stored, and a named scope refuses to store. One that resolves to the **empty string** (a blank header, `?x=`, a cookie sent as `name=`) has resolved: that caller shares the empty value's entry with every other such caller. Derive what you partition on from claims your identity provider never issues empty.

`private`

The value a private response's entry is partitioned by. A response is private when your origin says `Cache-Control: private` or a selected field is scoped `PRIVATE`.

- **If it resolves**, the response is stored under that value. Two requests resolving to the same value share one entry; two resolving to different values never meet.
- **If it does not resolve** (not configured, or its source is missing on this request), the response is **not cached**: `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`. It is never stored once and served to the next caller.
- **It must identify one caller, not their group.** `auth:idp.sub` and `cookie:session_id` do. An organization, a customer id, a role or a plan name do not: every member of the group then shares one entry, and nothing can validate this.

`private` also files a signed-in caller's response that your origin did not mark `public` or `private`, where a group-wide expression becomes a disclosure; see authenticated requests.

For data that really is the same for a whole organization, do not widen `private`. Use a vary rule, and for GraphQL a named scope on the types it is true of.

Vary rules

A `[[cache.policy.vary]]` rule evaluates an expression and writes the result onto the request as a header:

Field	Type	Description
`name`	String	The header the derived value is written to
`value`	Policy expression	Computes the value
`pass_to_origin`	`"never"`, `"anonymized"` or `"plain"`	Whether your origin receives the value, and in what form. Default `never`.
`scope_only`	Boolean	Whether the rule is left to your schema's named scopes: a response your schema decided is partitioned on it only when its scope names it. Default `false`. Accepted only with `pass_to_origin = "never"`.

gpilot.toml

```
[[cache.policy.vary]]
name  = "x-org"
value = 'auth:idp.org_id'
```

By default, a rule partitions every response, whether your schema or your origin's `Cache-Control` decided it, and whether or not a scope names the rule. Two callers whose rule resolves to different values never share an entry: that is the safe direction, because not keying on a value that shaped the response would serve one caller's answer to another. The price is hit rate. Each rule multiplies the entries a request needs, including requests

whose answer does not depend on it, so a role rule configured for one type also splits `PUBLIC` data, and data scoped to an organization, by role. Configure only the rules that matter.

`scope_only = true` leaves a rule to your named scopes. A named scope lists the rules that shape it in `@vary`. A rule marked `scope_only` partitions a response your schema decided only when that response's scope names it. This is what lets a named scope share one entry across its whole audience, and `PUBLIC` data be shared by everyone, even where a finer rule such as `x-role` is configured for a different type. `PUBLIC` and `PRIVATE` name no rules: their meaning is built in, and a `@vary` written on either is ignored.

gpilot.toml

```
[[cache.policy.vary]]
name       = "x-role"
value      = 'auth:idp.role'
scope_only = true
```

> **Every pair of values is its own stored variant**
>
> An entry is stored once for each combination of values its vary rules derive. With an organization rule and a role rule, every organization and role pair that reads a response holds its own variant of that entry, beside the variant shared by callers the rules derive nothing for. For data that is the same for everyone, mark the rules `scope_only = true`, so a `PUBLIC` answer your schema decided stays one entry shared by every caller. Fastly limits the number of variants it keeps for one entry, so a rule on a high-cardinality value, such as a user id, multiplies the entries and lowers the hit rate.

`scope_only` is accepted only with `pass_to_origin = "never"`, the default. A value your origin receives can change a response no scope names, and keying on it is then the only thing that keeps that response from reaching another caller, so any other combination refuses the deployment, with a message naming the entry.

A response whose lifetime your origin's `Cache-Control` decided keys on every configured rule, `scope_only` or not, and whether or not anything names it. Nothing above the vary rules themselves can tell which of them shaped such a response, so every one keeps partitioning it.

A rule that resolves to nothing writes no header. What happens to the response then depends on whether the request carried a credential:

- **Without a credential**, the response is stored as the variant where the header is absent. Only callers without a credential read that variant, so an anonymous caller never gets a signed-in caller's answer from it.
- **With a credential** (an `Authorization` header, a session cookie, or a header an auth provider reads), the response is not stored: `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`. A rule that resolves to the empty string counts as unresolved here. Otherwise a signed-in caller whose token lacks the claim would be stored in the variant every anonymous caller reads. A rule marked `scope_only` is not checked this way on a response your schema decided: there, only a scope that names it partitions on it.

A named scope listing the rule refuses in both cases. A rule whose `name` is not a valid header name is skipped.

A claim that is a number or a Boolean resolves to its JSON text (`42`, `1.5`, `true`), so an organization id your identity provider issues as a number partitions like a string one. `null`, an object and an array resolve to nothing. Values are partitioned under an unambiguous encoding, so two different values never share a variant. `pass_to_origin` still sends your origin the value as it was derived, and nothing for a value that a header cannot carry, such as one with a control character.

A `value` may combine claims, for an audience your origin also needs to tell apart:

gpilot.toml

```
[[cache.policy.vary]]
name           = "x-audience"
value          = 'auth:idp.org_id + ":" + auth:idp.role'
pass_to_origin = "plain"
```

For org `org_7` and role `admin` this derives `5:org_7:5:admin`: each claim is written as its length in UTF-8 bytes, a colon, and the value, and the literal `":"` stays as written. So no separator has to be chosen with care: org `a:b` with role `c` and org `a` with role `b:c` derive different values. A token missing either claim derives nothing, `+` or `&&` alike: the caller then shares the variant without the header, or, where a named scope lists the rule, gets no entry at all. See combining sources in a cache key.

Do not give a rule a `gp-` name. That prefix is reserved for headers GraphPilot sets itself and is removed from every request before it leaves for your origin, so `pass_to_origin` cannot work for it.

What your origin receives

`pass_to_origin`	What the origin receives
`never` (default)	Nothing. The header is removed before the request is forwarded.
`anonymized`	`v1=` followed by an unpadded base64url HMAC-SHA256 digest, under a per-service key, of the rule's header name and value.
`plain`	The derived value, verbatim. For a value that combines two or more sources with `+` or `&&`, read it left to right: each part is either a literal from your expression, or a decimal length in UTF-8 bytes, `:`, and exactly that many bytes of the source's value (`5:org_7:5:admin` for `auth:idp.org_id + ":" + auth:idp.role`).

`anonymized` lets your origin tell two contexts apart without learning what distinguishes them. The digest is keyed, because a bare hash of a low-entropy value can be reversed by trying every candidate. The key is per service, never sent to you or your origin, and not your signing key. The same value under two rules produces two digests. Parse the `v1=` prefix rather than assuming the whole value is the digest.

Configuring a rule claims its header name: a value a client sends under that name is removed on the way in, even when the rule resolves to nothing, so a value under that name reaching your origin was always derived by GraphPilot.

What Insights records

Every request whose entry a rule partitioned records the rule's value in its telemetry, shown in Insights as the request's **Segment** and under **Segmentation** on its **Cache** tab. `pass_to_origin` and `telemetry.vary_values` decide the form of that value:

`pass_to_origin`	Recorded with `vary_values = "pseudonymized"` (default)	Recorded with `vary_values = "clear"`
`never` (default)	A keyed pseudonym, `hmac-sha256:` and 16 hex digits.	The derived value, in clear.
`anonymized`	The same `v1=` digest your origin receives.	The same `v1=` digest your origin receives.
`plain`	A keyed pseudonym, `hmac-sha256:` and 16 hex digits.	The derived value, in clear.

The pseudonym is made with the same per-service key as the `v1=` digest, for your telemetry alone: it is not what a `plain` origin receives, and it cannot be turned back into the value. Equal values give equal pseudonyms, so requests of one segment still group together, and the **Segment** filter matches a pseudonym copied from a request. The Cache tab marks each value as **pseudonymized**, **anonymized** or **clear**.

> ⚠ **Storing values in clear is your choice**
>
> With `vary_values = "clear"`, every rule at `never` or `plain` stores its value in clear on every request it partitions. A rule whose value is a session cookie, an `Authorization` header, an email address or any other personal data then stores that data in your request telemetry, where Insights shows it, and the responsibility for that lies with you. The `[telemetry.headers]` sanitize rules do not apply to it: they cover the recorded headers, not the segment value. Keep the default, partition on a value that identifies without being a secret, such as a verified claim (`auth:idp.org_id`) rather than the token it came from, or wrap the value in `pseudonymize` so that it never leaves the edge.

A value in clear never equals its pseudonym. Requests recorded before a change of `vary_values` keep the form they were stored in, so requests of one segment can appear as two groups across that deployment. Requests recorded before runtime 1.6.0 store the value of a `never` or `plain` rule in clear, the way `"clear"` does now.

The `private` and `public` expressions have no `pass_to_origin`. Their values are always recorded as a keyed digest, never in clear.

`public`

An expression that partitions the entries of shared, non-private responses, for example per language. Like `private`, its value partitions the entry internally and is not part of the cache key itself.

gpilot.toml

```
[cache.policy]
# An origin that answers in German or in English.
public = "starts_with(lower(header:accept-language), 'de')"
```

The predicate collapses every browser's `Accept-Language` string into `true` or `false`, so each response has two entries rather than one per distinct string; see cardinality. Derive the value the way your origin picks the language: a caller placed in the `false` entry must be one your origin answers in English. The value is not sent to your origin; for more than two languages, or an origin that should be told the language, use the vary rule in caching per language.

The `private` and `public` values are never sent to your origin and are not stored in the cached response. A client cannot supply them either: GraphPilot removes every request header in the reserved `gp-` namespace a client sends (see headers).

Your origin's `Vary` header

Your origin can partition without any configuration. Every request header it names in its `Vary` response header becomes part of that entry's key, and the edge keeps one variant per distinct value:

```
Cache-Control: max-age=60, public
Vary: accept-language
```

This adds to the vary rules: beside an `x-org` rule it keeps one entry per organization **and** language. The headers your vary rules derive do not need to be named: a response whose lifetime your origin's `Cache-Control` decided keys on every rule already.

Naming the credential itself, `Vary: authorization`, is what lets a response to a signed-in caller stay shared; see keeping the shared entry with `Vary`.

The one value that does not work is the literal `Vary: *`, which is not a wildcard over header names: RFC 9111 specifies that it always fails to match a later request. GraphPilot does not store such a response (`CACHE_SKIPPED_VARY_WILDCARD`). If you mean "do not reuse this", send `Cache-Control: no-store`; if the response varies by something, name the headers.

`bypass`

When this expression matches a request, the request goes to your origin without the cache being consulted, and its response is not written back. Reported as `PASS` with `CACHE_PASS_BYPASS_POLICY`.

gpilot.toml

```
[cache.policy]
bypass = "header:x-preview-token"
```

`bypass` *decides* rather than derives a value, so it is evaluated as a truth; see values versus truths. A request carrying none of the context the expression names never bypasses. Unset by default: every bypass is an origin

request the cache would otherwise have absorbed.

A bypass on a header alone, as above, is open to anyone who learns the header's name. To give only your own staff a live view, require a verified claim as well:

gpilot.toml

```
[cache.policy]
bypass = "auth:idp.role == 'admin' && header:x-preview"
```

An admin who sends `x-preview: 1` reaches your origin and is shown what it answers right now, for example a price that is not yet published, and that answer is not written back. Everyone else is served from the cache, header or not: for a caller with another role the comparison is false, and for a caller with no role it has no answer, and an expression with no answer does not bypass. To keep a debugging header such as `server-timing` from everyone but the same admins, use a conditional response rule.

Keeping identifiers out of Insights with `pseudonymize`

`pseudonymize(s)` replaces a value with a keyed pseudonym, 43 characters made with a secret key of your service, before the cache, your origin or your telemetry see it. Equal inputs give equal pseudonyms within the service, so a partition built on one still groups the same callers. It is accepted only in the values that partition the cache: `cache.policy.public`, `cache.policy.private` and a vary rule's `value`. Anywhere else, such as `bypass`, a response rule's `when`, `telemetry.record` or a client profile, the deployment is refused with a message that says `pseudonymize() is only allowed in cache.policy.public` and names the keys that call it.

What it changes depends on where the value goes. For a verified token with `org_id = "acme"`, a `sub` and an `email`:

A vary rule's `value`	Insights, `vary_values = "pseudonymized"` (default)	Insights, `vary_values = "clear"`	A `plain` origin receives
`auth:idp.org_id`	`hmac-sha256:` and 16 hex digits	`acme`	`acme`
`auth:idp.email`	`hmac-sha256:` and 16 hex digits	The email address, in clear	The email address
`pseudonymize(auth:idp.email)`	`hmac-sha256:` and 16 hex digits	The 43-character pseudonym	The pseudonym
`auth:idp.org_id + ':' + pseudonymize(auth:idp.sub)`	`hmac-sha256:` and 16 hex digits	`4:acme:43:` and the pseudonym	`4:acme:43:` and the pseudonym

- **Under the default, Insights shows a digest either way.** There `pseudonymize` changes what the cache partitions on and what a `plain` origin receives, not what Insights shows.
- **Under `"clear"`, it decides which part of a value is readable.** That setting applies to every rule of the service, so it is the reason to wrap the personal part of a value: an organization id or a language can be

read in clear, an email address or a user id should not be.

- **In `private` and `public` it changes nothing Insights shows**, because those values are always recorded as a keyed digest. It still keeps the clear value out of what the entry is partitioned on.
- **The last row is length-prefixed** because it combines two sources, see vary rules; a single source wrapped in `pseudonymize` is the pseudonym alone.

The table assumes `pass_to_origin = "plain"` or `"never"`. Under `"anonymized"`, the origin and Insights both get the `v1=` digest of the derived value, pseudonym or not. How the three settings combine is on combining the options.

Example: an origin that personalizes without learning who is asking

A recommendations origin, run as a service of its own, answers per shopper and should never receive an email address or a token:

gpilot.toml

```
[[cache.policy.vary]]
name           = "x-shopper"
value          = "pseudonymize(auth:idp.email)"
pass_to_origin = "plain"

[origin]
headers = { hide = ["authorization"] }
```

Each signed-in shopper gets an entry of their own (a visitor without a token derives no value and shares the entry kept for requests without one), and the origin receives `x-shopper` with the pseudonym, the same one on every request from that shopper, which it can keep per shopper without being able to turn it back into the address. The token is dropped just before the request is forwarded, after it was verified and the cache key was built, so the origin sees no credential at all; have it check request signing to know a request came through GraphPilot.

- **A service of its own,** because a rule the origin receives partitions every response and cannot be marked `scope_only`.
- **`auth:idp.sub` is the stabler input** where your identity provider issues one: an email address can change, and with it the pseudonym and every entry keyed on it.
- **`pass_to_origin = "anonymized"`** on `auth:idp.email` would hand the origin an equally opaque `v1=` value. `pseudonymize` is the one to reach for when only part of a value should be hidden, as in the last row of the table.

A per-user scope that keeps the organization readable in Insights is on scopes in the schema, and worked through for five users in partitioning your cache. A `private` expression built on a pseudonym is on authenticated requests, and what a pseudonym changes about purging on revoking access.

Examples

A JWT for the caller and a claim for a group

gpilot.toml

```
[auth.providers.my_auth_provider]
type = "jwt"
allowed_algorithms = ["HS256"]
secret = "secret"
allowed_issuers = ["https://idp.example.com/"]
allowed_audiences = ["my-api"]

[cache.policy]
private = 'auth:my_auth_provider.sub'

[[cache.policy.vary]]
name = "x-org"
value = 'auth:my_auth_provider.org'
pass_to_origin = "anonymized"
```

Private responses are stored once per user (`sub`), and every response, plain HTTP included, once per organization as well. The rule cannot be marked `scope_only` , because your origin receives its value. `sub` survives a token refresh, where keying on the token itself would start a new entry per token.

One entry per language

gpilot.toml

```
[[cache.policy.vary]]
name           = "x-locale"
value          = "lower(split(split(split(header:accept-language, ',', 0), ';', 0), '-',
0)) || 'en'"
pass_to_origin = "plain"
```

A browser sends `Accept-Language: de-DE,de;q=0.9,en;q=0.8` . The rule takes the first language range (`de-DE`), drops a `;q=` weight if it has one, keeps the language in front of the region (`de`) and lowercases it, so callers sending `de-DE` , `de-AT` or `DE` all derive `de` and share one entry. A rule on the raw header would keep one entry per distinct browser string instead; see cardinality.

Your origin receives `x-locale: de` and has to render from that header, not from `Accept-Language` . `|| 'en'` gives a caller without the header the `en` entry; without it, a request with a credential and no `Accept-Language` is not stored (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`). The full walk-through, with the requests to send and what Insights shows, is the recipe caching per language.

The raw `Authorization` header

gpilot.toml

```
[cache.policy]
private = 'header:authorization'
```

This is safe only if an `[auth.providers.*]` block reads `Authorization`, so a token that does not check out is refused before it can key anything. Without one, an expired or revoked token whose exact value was cached keeps being served that entry until it expires. Prefer reading a claim, as above.

Two criteria for one caller

gpilot.toml

```
[cache.policy]
private = 'auth:my_auth_provider.sub && cookie:session_id'
```

`&&` resolves to nothing if either part is missing, so an entry is served only to a caller who matches both, and a request without the token or the cookie is not cached. The key carries each part with its length, so no `sub` and cookie pair can derive another pair's key.

Partitioning answers "who", not "until when"

An entry keyed to one caller is still served for its whole lifetime without your origin being asked, so a revoked session or a withdrawn permission is not reflected until it expires or you purge it. See revocation.

CACHING · WHO SHARES AN ENTRY

Authenticated requests

A client usually attaches its credential to **every** request: an API client sets the `Authorization` header once, a browser sends its session cookie on every call. A request for public data and a request for someone's inbox look identical apart from what they ask for. Whether a response was computed for one caller or is the same for everybody is a question only your origin can answer, and by default it does not. This page describes what GraphPilot does when the answer is missing.

The rule in one table

The rule applies to plain HTTP and GraphQL alike, whenever your **origin sent a `Cache-Control` header** and that header is what made the response cacheable. A request counts as credentialed when it carries any of these:

- a non-empty `Authorization` header;
- a non-empty value in a header an `[auth.providers.<name>]` block is configured to read, such as `x-api-key`;
- a `Cookie` header naming at least one cookie that is not on `cache.cookies.allow`.

The cookie check is per cookie, not per header. A pair with no readable name, such as the `=secret` in `Cookie: GAESA=abc; =secret`, counts as a credential on its own, because it cannot be checked against the allowlist. Separator noise does not: `Cookie: GAESA=abc;` is an ordinary allowlisted jar. A value that is nothing but separators (`;;`), or that GraphPilot cannot read as text, counts as a credential.

Your origin's `Cache-Control`	Request carries a credential	Result
`max-age=60, public`	yes	Cached and **shared**
`s-maxage=60`	yes	Cached and **shared**
`max-age=60` + `Vary: authorization`	yes	Cached and **shared**, keyed by the credential
`max-age=60, private`	yes	**Partitioned**, as always
`max-age=60`	no	Cached and **shared**
`max-age=60`	yes, and a private cache policy resolves	**Partitioned**
`max-age=60`	yes, and no private cache policy resolves	**Not cached**

The last two rows are where this rule changes the outcome, and only the last one loses you a cache entry. "Partitioned" means filed under your `cache.policy.private` value, and nothing more; read your expression before you rely on those rows.

Why a bare `max-age` is not enough

RFC 9111 section 3.5 is unusually direct here, and it is one of the few places the specification does not leave the decision to the cache:

> A shared cache MUST NOT use a cached response to a request with an `Authorization` header field to satisfy any subsequent request unless a cache directive that allows such responses to be stored is present in the response.

A shared cache needs a statement it could only have been given on purpose. `max-age` says how long the answer stays fresh, which is just as true of a response meant for one person. `public` and `s-maxage` mean nothing except to a cache that other people also read from. Many shared caches simply bypass the cache for a request carrying `Authorization` or `Cookie`; GraphPilot is deliberately gentler, because refusing outright would punish the ordinary client described above.

If your origin runs Apollo Server, this changes nothing

Apollo Server's cache-control plugin always writes the scope into the header it emits, as `max-age=60, public` or `max-age=60, private`, never a bare `max-age`, so its responses take the first or fourth row of the table above.

What happens instead of refusing: the partitioned entry

If a private cache policy is configured and resolves for the request, GraphPilot files the entry **under that policy's value**, exactly where a response marked `Cache-Control: private` would have gone.

gpilot.toml

```
[cache.policy]
private = 'auth:idp.sub || cookie:session_id'
```

That one line turns the last row of the table into the second-to-last, and it also decides what happens to a response your origin *does* mark `private`. This shows up only as an edge code on the request, which the portal lists under **Edge codes** in the Insights **Overview** tab:

```
CACHE_STORED_PRIVATE   header=authorization
```

`header` names the credential header: `authorization`, `cookie`, or the header an auth provider reads. The response carries `gp-cache: MISS` and **no** `gp-cache-reason`, because the entry was stored, and the next matching request from the same caller is a `HIT`. See cache statuses for how the two headers relate.

Read your `private` expression before you rely on this

> **The partition is only as narrow as you wrote it**
>
> GraphPilot files the entry under whatever `cache.policy.private` resolves to. That is the caller **only if you wrote it that way**. With `auth:idp.sub` an unmarked response is shared with nobody else; with `auth:idp.org` or `header:x-customer-id` it is shared with **everyone in that organization or at that customer**, although your origin computed it for one person and did not say so. Nothing validates that, because nothing can.
>
> `cache.policy.private` has to identify **one caller**, and has to be read from a source the caller cannot choose. See `private`. If yours names a group, narrow it, or mark per-person responses `Cache-Control: private` so nothing has to be inferred from the request at all.

A partitioned entry also costs hit rate: it follows how often the *same* caller repeats a query, not how often *anyone* does. For genuinely public data behind a signed-in session that is a large difference, and the fix is one word, `public`.

Partitioning on a pseudonym instead of the user id

A REST endpoint of the shop, `GET /me/favorites`, answers each signed-in shopper with their own list, and the origin sends a bare `Cache-Control: max-age=60`. The partition only has to tell shoppers apart, not say who they are, so it can be built on a pseudonym of the subject instead of the subject itself:

gpilot.toml

```
[cache.policy]
private = 'pseudonymize(auth:idp.sub)'
```

- **The cache key is unchanged, and so is the hit rate.** The entry is still keyed on the method and path, `GET:/me/favorites`, and filed under the caller's partition. One shopper always derives the same 43 characters, so each still has exactly one entry, and it survives a token refresh as `auth:idp.sub` does. Only the value the partition is built on is the pseudonym rather than the `sub`.
- **Insights shows a keyed digest either way.** A `private` value is always recorded as a keyed digest, under `telemetry.vary_values = "pseudonymized"` and `"clear"` alike, and your origin never receives it.
- **The pseudonym is keyed.** It is made with a secret key of your service that neither you nor your origin holds, so it cannot be turned back into the `sub`. The same `sub` gives a different pseudonym on another service.

To see per-shopper traffic grouped in Insights, or to let your origin tell shoppers apart without learning who they are, the value has to go into a vary rule instead, where `vary_values` and `pass_to_origin` apply; see an origin that personalizes without learning who is asking.

When nothing can be cached

If no private policy resolves for the request, there is nothing to key a narrower entry on, and the only safe thing left is not to store. The response is delivered normally and reported as:

```
gp-cache: PASS
gp-cache-reason: CACHE_SKIPPED_CREDENTIALED_REQUEST
```

The log line names the request header that triggered it, so you can tell an `Authorization` case from a `Cookie` one or from an auth provider's header, and for a `Cookie` it also names the first cookie that was not on `cache.cookies.allow`. A pair with no name is reported as `<unreadable>`, the same word the `CACHE_SKIPPED_SET_COOKIE` line uses for a `Set-Cookie` it could not read: there is no name to add to the allowlist, and the fix is to stop sending the malformed pair.

To get caching back, pick whichever is true of the response:

- It is the same for every caller. Send `Cache-Control: max-age=60, public`. Best outcome: one shared entry, served to everybody.
- It differs per caller. Either name the credential in `Vary`, as described below, or configure `cache.policy.private` so the entry can be partitioned, and ideally also mark the response `Cache-Control: max-age=60, private` so the intent is in the response rather than inferred from the request.
- It should not be cached at all. Send `Cache-Control: no-store`, which is explicit and does not depend on this rule.

Keeping the shared entry with `Vary`

An origin whose response really does depend on who asked can name the credential in `Vary`:

```
Cache-Control: max-age=60
Vary: authorization
```

The entry is then keyed by that header's exact value, so a request carrying a different credential can never match it, and GraphPilot stores the response normally. See your origin's `Vary` header for how `Vary` becomes part of the cache key.

> ⚠ **Honored, but not the shape to reach for**
>
> Keying on a raw token is safe and a poor cache key: the value changes on every refresh, so each reissue is a miss. A `cache.policy.private` of `auth:idp.sub` separates the same callers and survives a refresh, which is why the policy examples read a claim out of the JWT rather than the token itself. `Vary: authorization` is honored because it is correct, not because it is the best option.

Three conditions:

- **Every credential the request carried has to be named.** If your clients send an `Authorization` header *and* a `Cookie`, then `Vary: authorization` alone is not enough, and such a response falls back to the partitioned entry above. The shape this protects against is a customer-wide API key in `Authorization` next to an individual session in `Cookie`: keying on the API key would share one person's answer with everyone at that customer while looking as though the credential had been accounted for.
- **A header an auth provider reads counts too.** If an `[auth.providers.<name>]` block reads `x-api-key`, a request carrying it needs `Vary: x-api-key` (next to any other credential it carried). A provider configured with `header = "Cookie"` declares the cookie its credential, so the whole `Cookie` header counts and `cache.cookies.allow` no longer excuses any of it.
- `Vary` **must name the credential itself.** A `Vary` naming something else does not count, even when it partitions the cache usefully. This includes a header a vary rule of yours derives:

  gpilot.toml

  ```
  [[cache.policy.vary]]
  name = "x-org"
  value = 'auth:idp.org'
  ```

  Naming it in `Vary` is not permission to share, because it partitions the caller's *group*, not the caller. Such a response is filed under your private policy as well, so it ends up keyed by organization **and** by caller.

Revocation

Every cached authenticated response, shared, partitioned or `Vary`-keyed, stays servable until it expires or is purged. Taking access away is yours to act on: see revoking access.

Details worth knowing

`@cacheControl` in your schema is exempt

The rule reads your origin's HTTP header, and only that. If a response is cacheable because of a `@cacheControl` annotation in your schema instead, it is stored the way the annotation says, credential or no credential.

An annotation is a statement you wrote on one field, which cannot get there by accident the way a framework's default `max-age` can. Apollo Server reads it the same way: its plugin turns `@cacheControl(maxAge: 60)` into `Cache-Control: max-age=60, public`.

> **An unscoped hint is shared, credential or not**
>
> An annotation with no `scope` resolves `PUBLIC`, so a signed-in caller's response to a query whose hints are all unscoped is stored in the shared bucket. Per-user fields need a scope; see scopes in the schema.

`Cookie` counts, even though the RFC only names `Authorization`

A session cookie is the same credential in a different envelope, and a rule ignoring cookies would miss most of the traffic it exists for. A header an auth provider reads is a credential for the same reason.

An empty `Authorization` value does not count. `Authorization:` with nothing after it identifies nobody, and some HTTP clients emit one by accident.

`Cookie` is judged differently: not by presence, but by whether it names anybody. A `Cookie` header whose every cookie is on `cache.cookies.allow` identifies nobody either: those are load balancer affinity and bot-management cookies, not a session. The exception is a provider that reads `Cookie`, described above.

`must-revalidate` is not accepted as permission

RFC 9111 names three directives that permit sharing: `public`, `s-maxage`, and `must-revalidate`. GraphPilot honors the first two and deliberately not the third.

`must-revalidate` is on that list because a cache that obeys it asks the origin before every use past the lifetime. GraphPilot has no conditional revalidation: it honors the directive by dropping the stale window, and an expired entry is an ordinary miss. Accepting the directive as permission would mean accepting an allowance granted on a promise GraphPilot does not keep, so this is stricter than the specification, on purpose.

Send `public` or `s-maxage` to express what `must-revalidate` was standing in for.

It does not affect what your client receives

This rule only decides whether and where GraphPilot stores a copy. The response your client gets is your origin's response, headers included, on the stored path and the refused path alike.

Revoking access

Revocation is the ordinary division of responsibility between an origin and a cache, and it applies to **every** cached authenticated response: the shared one, the partitioned one, and the `Vary`-keyed one alike.

What the edge undertakes

An entry never crosses callers, and it stops being served when the freshness you set runs out. If you configure an auth provider for the header, authentication runs on every request, cache hit included: a credential whose signature does not check out, whose `exp` has passed or is missing (unless the provider sets `allow_missing_exp`), or that fails a required claim is refused and gets nothing back. Without a provider, GraphPilot does not verify the credential at all, and an entry keyed on its raw value keeps being served to whoever sends that exact value until it expires or is purged.

What is yours

A change that leaves the credential itself valid: a session revoked server-side, a role taken away, an item shared and then unshared. Your origin knows, and during the freshness window it is not being asked. Nothing at the edge can infer it, and a cache that tried would have to consult your origin on every hit, which is no longer a cache.

Revoke, then purge

Revoking access is two steps rather than one, exactly as it is behind any CDN: revoke at your origin, then **purge**. A purge by surrogate key reaches every cached response that touched a given key across all its variants, so if your responses carry a per-user key such as `User:id:482910`, purging that key is the second step. See surrogate keys for attaching one.

Two things bound how much that matters. The window is exactly what you set, the lifetime plus any stale window, so a short lifetime on permission-sensitive data is the control you have without acting at all. And a response whose correctness depends on a permission check at the moment of reading should not be cached: `Cache-Control: no-store`, or leave it out of the annotated part of your schema.

Example: demoting a member

Ben is an admin in the organization `org-acme` and becomes a viewer. The service caches three kinds of answer for him, partitioned as in three rungs in one service: products (`PUBLIC`), the organization's report (a named scope keyed on organization and role) and his own profile, `Me` (`PRIVATE`).

1. **Change the role and stop new tokens carrying the old one.** Update the role at your identity provider and revoke Ben's sessions or refresh tokens there, so the next token he is issued says `viewer`. From that token on, his requests derive `x-gp-role: viewer` and land in the viewer entry of the report.

2. **Know the window that is left.** The token Ben already holds still says `admin` and still verifies, so until it expires he keeps landing in the admin entry, served from the cache without your origin being asked. The edge checks `exp` on every request, cache hits included, so the window ends with that token. Shorter-

lived access tokens shorten it, and `max_validity` on the provider refuses any token issued longer ago than the limit, whatever its `exp` says.

3. **Leave the shared entries alone.** The admin entry of the report is still correct for the other admins in `org-acme`, and purging it would not keep Ben out: his next request with the old token misses, reaches your origin and is stored in the admin entry again.

4. **Purge what was computed for him alone.** His `Me` entry was answered while he was an admin. A response containing a `Me` object carries its entity key, here `Me:id:user-ben`, so one purge removes it wherever it was stored:

   ```
   gpilot purge --service shop-api 'Me:id:user-ben'
   ```

   Read the key off the response's `gp-surrogate-key` header, if your service sends it, rather than constructing it; see verifying a purge.

> ⚠ **Authorize from the claim the entry is keyed on**
>
> Step 3 holds only while your origin decides what Ben may see from the role in his token. If it reads the role from its own database instead, a request still carrying the old token is keyed as `admin` and answered as `viewer`, and that answer is stored where the other admins read it. The reverse change is worse: a promotion answered from the database before the token changes stores an admin's answer where viewers read it. Key the entry on what the origin authorizes from, or keep the data out of the cache.

Partitions built on a pseudonym

A per-user partition built with `pseudonymize`, such as `Me` under a named scope keyed on the organization and a pseudonym of the user, is revoked exactly like the example above:

gpilot.toml

```
[[cache.policy.vary]]
name       = "x-gp-organization-user"
value      = "auth:demo.org_id + ':' + pseudonymize(auth:demo.sub)"
scope_only = true
```

- **You purge by the same key.** A purge never addresses a partition: it addresses surrogate keys, and those come from the response (its entity keys, such as `Me:id:user-ben`, and its types) and from the request's cache key, never from a vary rule's value or the `private` value. So `gpilot purge --service shop-api 'Me:id:user-ben'` removes Ben's entry whether the rule reads `auth:demo.sub` or its pseudonym, and you never need to know the pseudonym.
- **A purge by cache key reaches every partition.** `ck:` and `purgeByRequest` remove the entry for that request in every caller's partition, not only Ben's. Use them when the answer is wrong for everybody, not

to take one person's access away.

- **A demotion does not change the pseudonym.** It is derived from `sub` alone, so Ben keeps the same partition after the role change, and his old token keeps landing in it until it expires. If what `Me` contains depends on the role, purge it as in step 4. Moving Ben to another organization changes the organization part of the value, so tokens that carry the new organization land in a new entry, while the old token still reads the old one until it expires.

CACHING

Header and cookie filters

GraphPilot lets you drop specific headers and cookies by name at four points in a request's life. Each point answers a different question, and they do not share a list.

The four filter points

Section	Applies to	When
`[origin.headers]`, `[origin.cookies]`	The request, on its way to your origin	Just before the request is forwarded, after the cache decision has already been made from the request as the client sent it
`[response.headers]`, `[response.cookies]`	The response, on its way to your client	On delivery, which includes a cache **hit** served from an entry written before this configuration existed
`[cache.headers]`	Which response headers are written into a cache entry	When the entry is stored
`[cache.cookies]`	Which cookie names the cache treats as carrying no session	Read by both the `Set-Cookie` storage rule and the credentialed-request rule, and by `[cache.headers]`'s own handling of `Set-Cookie`

```
[origin]
headers = { hide = ["x-internal-debug"] }
cookies = { hide = ["internal_ab_test"] }

[response]
headers = { hide = ["x-origin-latency-ms"] }
cookies = { hide = ["__internal_session"] }

[cache.headers]
exclude = ["x-request-id"]
allow   = ["www-authenticate"]

[cache.cookies]
```

```
allow = ["GAESA", "my_custom_lb_cookie"]
```

`[origin.headers]` and `[origin.cookies]` have no cache semantics. The cache lookup, the store decision and policy expressions have already run against the request as the client sent it, so a name dropped here never reaches your origin but was never withheld from the cache decision.

`[cache.headers]` is two independent lists. `exclude` names headers that are not written to the entry at all: the client that caused the miss still gets them, a later hit does not. `allow` names headers that no longer refuse the store, out of the fixed list GraphPilot would otherwise refuse a response for: `WWW-Authenticate`, `Proxy-Authenticate`, `Authentication-Info`, `Proxy-Authentication-Info`, `Clear-Site-Data`, `Set-Login`. An allowed header is also never written to the entry, since allowing it says it carries nothing worth replaying.

> **allow only accepts a name from that fixed list**
>
> Naming an ordinary header in `[cache.headers].allow` is a **deployment failure**. `allow` only has meaning for the fixed list above; use `[cache.headers].exclude` to drop an ordinary header from the cache entry.

`Set-Cookie` and `Set-Cookie2` are refused in `[cache.headers].allow` too, even though they are on the fixed refusal list: each cookie a `Set-Cookie` sets is checked against `[cache.cookies].allow` first, per name, so allowing the header could never have affected them. Use `[cache.cookies].allow` to allow a specific cookie.

`hide` versus `only`

`[origin.headers]`, `[origin.cookies]`, `[response.headers]` and `[response.cookies]` share one shape:

Key	Names	Everything else
`hide`	Dropped	Survives
`only`	Survives	Dropped, except the protected names below

Setting both `hide` and `only` on the same section is refused at deploy time, because “hide `a`, keep only `b`” has no unambiguous reading.

Conditional rules for the response

`[response.headers]` and `[response.cookies]` also accept rules that apply only to some callers. Each `[[response.headers.rules]]` or `[[response.cookies.rules]]` entry holds a `hide` or an `only` list and an optional `when`, a policy expression evaluated as a truth for each request:

```
[response.headers]
hide = ["x-origin-debug"]

[[response.headers.rules]]
hide = ["x-debug-trace"]
when = "auth:idp.role ≠ 'admin'"
```

Everyone loses `x-origin-debug`. Only admins see `x-debug-trace`.

The same shape keeps an origin's `Server-Timing` header, which tells anyone who reads it how long each of your backend calls took, for your own staff:

```
[[response.headers.rules]]
hide = ["server-timing"]
when = "!auth:idp.role || auth:idp.role ≠ 'admin'"
```

A filter on delivery applies to a hit as well, so a hit served to a customer carries no `Server-Timing` even when the entry was filled by an admin's request.

For a caller with no role, that `when` is **undecidable**, and an undecidable condition reads as *the rule applies*, so an anonymous caller loses `x-debug-trace` too: a filter that cannot tell whether a condition held must not be the one that reveals. Writing `!auth:idp.role || auth:idp.role ≠ 'admin'` says the same thing explicitly. See a request the expression cannot decide.

How the lists combine:

- The plain `hide` or `only` list applies first, then every rule whose `when` matched, in the order they are written. A name reaches the caller only if **every** list that applies keeps it: a `hide` rule adds to what is dropped, an `only` rule narrows what is kept.
- A rule without `when` applies to every request.
- A rule has to name something in `hide` or `only`, and may not set both. Its names follow the same protected names and matching rules as the plain lists. Each of these mistakes refuses the deployment, and the message names the rule by its position, such as `response.headers.rules[0]`.
- `when` may read `auth:` claims: the rules are decided after authentication has run. A request that is refused or passed on before authentication completes is decided without claims, the same as every other policy sees it.

> ⚠ **Only the response filters take a condition**
>
> `[origin.headers]`, `[origin.cookies]`, `[cache.headers]` and `[cache.cookies]` refuse `rules` and `when`, and the deployment fails. The message says why, and names the plain lists that section does support, so you know what to write instead:
>
> | Section | The refusal names |
> | --- | --- |
> | `[origin.headers]`, `[origin.cookies]` | `hide` and `only` |
> | `[cache.headers]` | `exclude` and `allow` |
> | `[cache.cookies]` | `allow` |
>
> On the two origin sections the reasoned message is for a `rules` list; a bare `when` there is refused as an unknown key.
>
> Those sections act under a cache key that does not include the condition, so two callers would share one key while producing different entries. The response filters apply on delivery, after the cache, so a hit and a miss are filtered the same way, per caller.

Protected names

Some names a filter is never allowed to touch, whatever `hide` or `only` says:

- GraphPilot's own `gp-` namespace (`gp-cache`, `gp-cache-reason`, `gp-surrogate-key`, `gp-origin-retries`, and every other header GraphPilot sets itself). To withhold the cache status or surrogate key headers from your clients, use `response.cache_headers` and `response.surrogate_key_header` instead. See switching the headers on and off.
- A fixed list per direction: `host`, `content-type`, `content-encoding`, `content-length` and `transfer-encoding` for a request-side filter (`[origin.headers]`); `content-type`, `content-encoding`, `cache-control`, `vary` and `surrogate-key` for a response-side or cache-side filter (`[response.headers]`, `[cache.headers]`).

The bar for being on these lists is silent corruption. GraphPilot compresses the response toward the client itself, so a filter that removed `Content-Encoding` would hand your client compressed bytes that no longer describe themselves, and nothing would report an error. See compressed origin responses.

Headers GraphPilot never reads are deliberately **not** protected, including `Age`, `Date`, `Expires`, `ETag` and `Last-Modified`: hiding one costs exactly what you asked to hide.

Naming one of these in `hide` or `only` is a **deployment failure**, not a line that is quietly ignored. An `only` list does not have to enumerate them either: they survive whether or not they are listed, so allowlisting one header does not cost you your content type.

Cookie names are not checked against this list. A cookie called `vary` is an ordinary cookie, not the header of the same name, so `[origin.cookies]`, `[response.cookies]` and `[cache.cookies].allow` accept it.

Matching

Every list matches either as an exact name or, if the entry ends in a trailing `*`, as a prefix. `BIGipServer*` matches `BIGipServerpool_web` and also matches `BIGipServer` itself. A `*` anywhere except the very end is an ordinary character: there is no general glob and no leading wildcard, because a match wider than its author expected is the expensive direction to be wrong in.

Case sensitivity depends on what is being matched, and the two halves disagree on purpose:

Names	Case
Header names (`[origin.headers]`, `[response.headers]`, `[cache.headers].exclude`, `[cache.headers].allow`)	Case-**insensitive**, because HTTP treats a header name that way. `X-Debug` and `x-debug` are the same header.
Cookie names (`[origin.cookies]`, `[response.cookies]`, `[cache.cookies].allow`)	Case-**sensitive**, because RFC 6265 treats a cookie name that way. `GAESA` and `gaesa` are two different cookies.

> **A wrong case does not fail the deployment, it just never matches**
>
> If you write `cache.cookies.allow = ["gaesa"]` and your origin actually sets `GAESA`, the entry never matches: the deployment succeeds, and the cookie is still treated as carrying a session, exactly as if you had written nothing. Copy the case your origin actually sends.

The trailing-`*` prefix form follows the same rule per list: `BIGipServer*` in `[cache.cookies].allow` matches `BIGipServerpool_web` and does **not** match `bigipserverpool_web`, while `X-Debug*` in `[response.headers]` matches `x-debug-info` just as it matches `X-Debug-Info`.

What `cache.cookies.allow` actually does

A name on `[cache.cookies].allow` has three effects, and only three:

1. A `Cookie` value made only of allowed names does not make the request **credentialed** for the purposes of the authenticated-request rule.
2. An allowed name inside a `Set-Cookie` value does not by itself refuse storing the response.
3. An allowed name is **never written to the cache entry**, so a hit never replays it.

The cookie still reaches your origin and your client untouched; allowing a name only tells GraphPilot that it carries no session.

The default sessionless cookies (`cache.cookies.allow`)

Infrastructure in front of your origin routinely sets its own cookies, none of them carrying any identity, and each would otherwise disable the cache for the traffic behind it. GraphPilot ships a default `cache.cookies.allow` list:

```
GAESA, GCLB, AWSALB, AWSALBCORS, AWSELB, ARRAffinity, ARRAffinitySameSite,
BIGipServer*,
```

```
__cf_bm, __cflb, incap_ses_*, visid_incap_*
```

Three things worth knowing before you touch this key:

- **Setting `cache.cookies.allow` replaces the shipped list, it does not add to it.** If you add your own cookie, restate every default you still need alongside it.
- **Analytics cookies (`_ga`, `_gid`, `_fbp`) are deliberately not on the list.** They identify a visitor, and silently dropping them from a cache entry would change a customer's analytics without anybody asking for it.
- **Matching is case-sensitive.** See matching.

> **The App Engine case, end to end**
>
> Google App Engine sets `GAESA`, its session-affinity cookie, and the browser returns it on every later request. Without the allowlist it trips two refusals, `CACHE_SKIPPED_SET_COOKIE` on the response and `CACHE_SKIPPED_CREDENTIALED_REQUEST` on the next request, and a service behind App Engine caches nothing. Both refusal lines name the cookie, so the cause is findable from the log. Because `GAESA` is on the default list, both pass it, and it still reaches your origin and client untouched, which keeps App Engine's session affinity working.

Example: an A/B test cookie

Your client assigns each visitor a bucket in a cookie, `ab_bucket=a` or `ab_bucket=b`, and your origin answers the two buckets differently. Out of the box that cookie works against the cache: a request that sends it counts as credentialed, so a response whose `Cache-Control` says neither `public` nor `s-maxage` is filed per caller or not stored, and a response carrying `Set-Cookie: ab_bucket=...` is not stored at all (`CACHE_SKIPPED_SET_COOKIE`). Tell GraphPilot the cookie carries no session, and key the cache on the variant:

gpilot.toml

```
[cache.cookies]
allow = [
  # The defaults, restated: setting the key replaces the shipped list.
  "GAESA", "GCLB", "AWSALB", "AWSALBCORS", "AWSELB", "ARRAffinity", "ARRAffinitySameSite",
  "BIGipServer*", "__cf_bm", "__cflb", "incap_ses_*", "visid_incap_*",
  "ab_bucket",
]

[[cache.policy.vary]]
name           = "x-ab-variant-b"
value          = "in(cookie:ab_bucket, ['b']) || 'false'"
pass_to_origin = "plain"
```

The rule derives exactly two values, `true` for bucket `b` and `false` for everything else (bucket `a`, no cookie, or a value somebody made up), so each response has two entries however many cookie values arrive. Your origin

receives `x-ab-variant-b: true` or `false` and answers from that header.

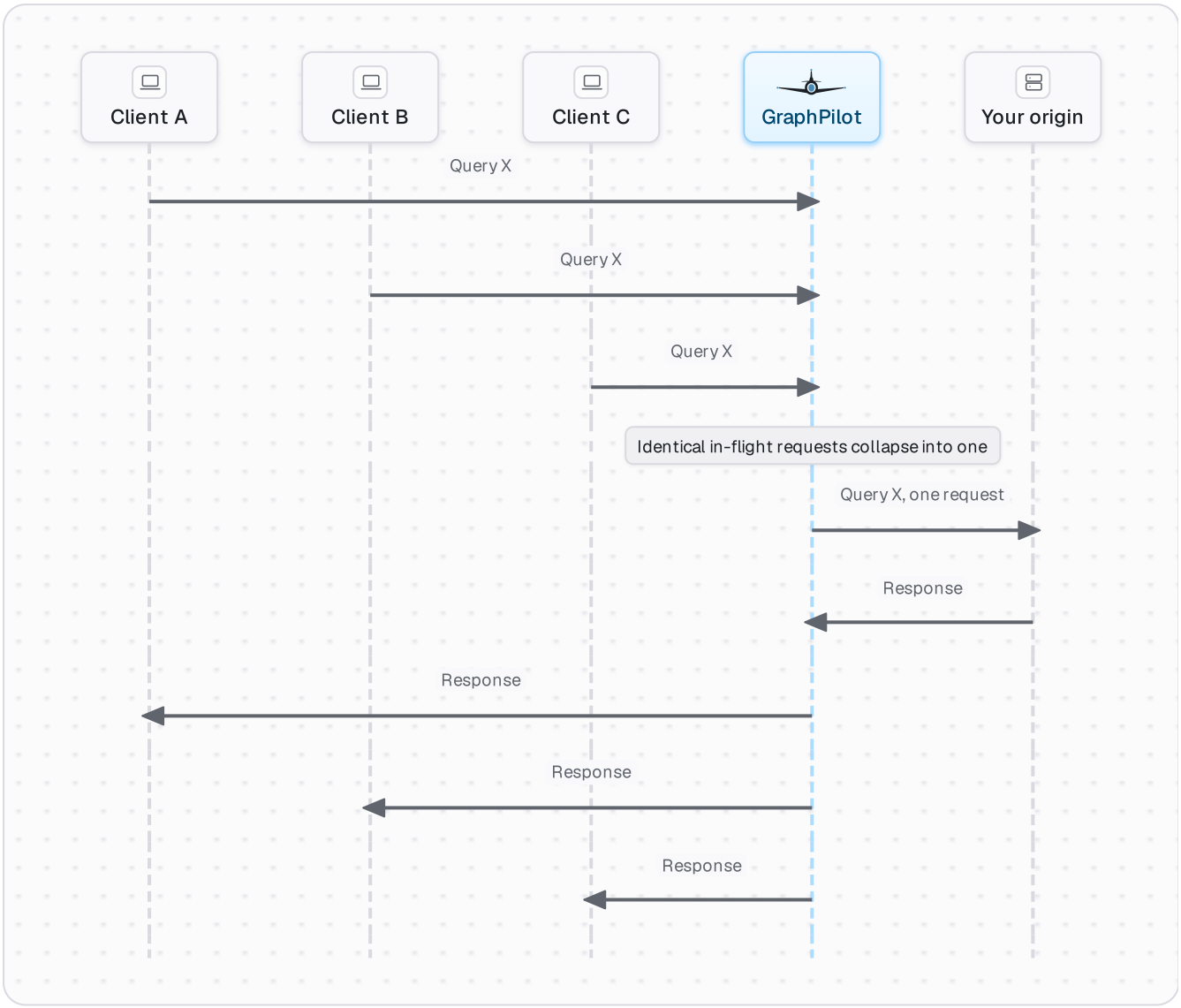
> ⚠ **A hit never hands out a bucket**
>
> An allowed cookie is never written to the cache entry, so a `Set-Cookie` that assigned the bucket would reach only the visitor whose request filled the entry. Assign the bucket where no cache stands in the way, in the browser or on a response GraphPilot does not store.

Assignment is the visitor's to change: anyone can send `ab_bucket=b`. That is fine for a test and never for anything a visitor should not be able to choose; see a value is only as trustworthy as its source. The requests to send, what Insights shows and the pitfalls are in the recipe A/B testing with a cookie.

CACHING

Request collapsing

Request collapsing (also called request deduplication) turns identical requests that arrive at the same time into a single origin request. When a popular entry expires, your origin answers once, not once per waiting client.



Three waiting clients, one origin request.

How it works

1. Several clients send the same operation at the same time.
2. They all resolve to the same cache key.
3. One of them goes to your origin. The rest wait.
4. The response is stored once and served to all of them.

Nothing is configured for this, and nothing can turn it off.

What it applies to

Collapsing follows the cache, so it covers exactly what the cache covers:

	Collapsed
A `MISS` on a cacheable operation	**Yes**, this is the case it exists for.

	Collapsed
A `PASS` with a `CACHE_PASS_*` reason	No. The cache was never consulted, so there is nothing to wait for.
A response that turns out not to be storable (a `PASS` with a `CACHE_SKIPPED_*` reason)	No. No entry is written, so there is nothing to hand to the waiting requests. They are released and go to your origin themselves.
Requests with different variables	No. Different variables mean a different key.
Requests in different buckets	No. A private entry and a public one are different entries.

The not-storable row is worth a second look. An operation whose responses are never stored gains nothing from collapsing: every request reaches your origin. Under a burst, the waiting requests may also reach it one after another rather than all at once, so an uncacheable hot operation can take longer under load than the same operation with a freshness lifetime would. If an operation should be cached, check its `gp-cache-reason`.

> **Note**
>
> Collapsing happens **per edge location**, like the cache itself. Ten simultaneous requests at one location become one origin request; the same ten spread across ten locations become ten. See edge locations.
>
> With shielding switched on, the misses from those ten locations meet again at the shield and collapse there, so your origin receives one request.

Why it matters

- **Thundering herd protection.** When a popular entry expires, the next thousand requests for it do not all become origin requests.
- **Cold start protection.** A newly deployed service, or a location that has not cached an operation yet, takes one origin request per operation rather than one per client.
- **It bounds the worst case.** Origin load from cacheable traffic is bounded by the number of distinct operations in flight, not by request volume.

To keep even that one request from holding clients up at expiry, give the entry a stale window; see stale windows and request collapsing.

Related

- When a response is cached
- Cache statuses
- Edge locations

Invalidation

INVALIDATION

Cache invalidation

A cached response is only useful while it is still true. GraphPilot removes an entry when the data behind it changes, so you can give responses a long freshness lifetime without serving old data after a write.

Invalidation works in two steps, the same for a plain HTTP API and a GraphQL API: **tag, then purge**. Every cached response is tagged with surrogate keys, labels that name what it contains. A purge names keys, and every entry carrying one of them is removed, whichever request produced it.

Ways to control this

Through	What it controls
Your origin	The `Surrogate-Key` header: tags a read, purges on a write.
Your GraphQL schema	`@surrogateKey` on types and fields; a mutation purges what it returned automatically.
`gpilot.toml`	Which fields count as keys and how far a mutation reaches (`[surrogate key]`).
The API, the CLI and the portal	Purge on demand: `gpilot purge` , the purge mutations and `POST /purge` , and the Purges page.

Tag

Every entry carries its own cache key (`ck:…`), so one response can always be purged on its own. On top of that:

- **Your origin names keys** in a `Surrogate-Key` header on the response to a read (a `GET` or `HEAD` , or a GraphQL query). Use it for anything that identifies what the response holds: a product, a customer, a content version.
- **For GraphQL, GraphPilot derives keys from your schema and the response data**: one key per object it returned (`Product:id:p01`), one per type and operation, and any you declare with `@surrogateKey` . Clients do not have to select the fields a key needs; selection set expansion adds them.

Purge

A purge happens in one of three ways.

When a write comes back

Your origin answers	GraphPilot purges
A plain HTTP `POST` , `PUT` , `PATCH` or `DELETE`	Every key in the `Surrogate-Key` header on the response.

Your origin answers	GraphPilot purges
A GraphQL mutation	Every object the mutation returned, automatically, plus every key in the `Surrogate-Key` header on its response.

For a plain HTTP API this is the whole mechanism: tag the `GET` with a key, name the same key on the response to the write, and the next `GET` is a `MISS`. See the Surrogate-Key header.

For GraphQL most of it needs no code: a mutation that returns the object it changed clears every cached response holding that object. It leaves some cases uncovered, most importantly a new item in a cached list that was not empty. Read automatic invalidation before you rely on it.

When you send a purge

For a change that never passes through GraphPilot (a database migration, a nightly import, an admin tool), purge through the GraphPilot API, with `gpilot purge`, or from the portal. You can address surrogate keys, one cached response by its cache key, a request from your telemetry, a GraphQL query by its hash, or the whole cache. See purging through the API and CLI, and verifying a purge to confirm what it removed.

When the lifetime runs out

Every entry has a freshness lifetime from your origin's `Cache-Control` header or, for GraphQL, your schema's `@cacheControl` policy. When it runs out, the entry stops being served whether or not anything purged it. That bounds how long an entry can be wrong in the worst case, but it is not your invalidation strategy.

Example: one product page

The pages in this section use one running example: a product catalog with `Product`, `Category` and `Review` types and a `setPrice` mutation, served by a service called `shop-api`. `Product` carries `@surrogateKey(of: "id")`, `@surrogateKey(of: "category.id")` and `@surrogateKey(static: "catalogue")`, so every product can be purged on its own, with its whole category, or with the whole catalog.

Tag. A product page asks for one product, its category and its reviews. With the `gp-surrogate-key` header switched on, the response shows every key the entry carries:

```
curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"query ProductPage { product(id: \"p01\") { id name price category { id name } reviews { id rating } } }"}'
```

```
gp-cache: MISS
gp-surrogate-key: ck:95f9f0f4…72b0 q:product Category Product Query Review catalogue Category:id:audio Product:category.id:audio Product:id:p01 Review:id:r01 Review:productId:p01
```

Purge on a write. A `setPrice(id: "p01", price: 2450)` mutation returns the product it changed, so its response carries `Product:id:p01` and GraphPilot purges it on the way back. The next `ProductPage` request is a `MISS` with the new price. The full walk-through, with what Insights and the Purges page show, is on automatic invalidation.

Purge you send. A nightly price import writes to the database directly, so no response passes through GraphPilot. The import job purges what it changed itself:

```
gpilot purge --yes --service shop-api Product:id:p01 Product:id:p07
```

The same purge from a webhook, without the CLI, is under purging over REST.

Related

- Surrogate keys: every kind of key and what a key may contain
- The Surrogate-Key header: tagging and purging from your origin
- Automatic invalidation: exactly what a GraphQL mutation purges
- Verifying a purge: where purges are recorded

INVALIDATION

Automatic invalidation

> **GraphQL only** A plain HTTP write purges the keys named in the Surrogate-Key header on its response.

Every mutation that goes through GraphPilot invalidates the cached responses that contained the objects it returned. You write no purge call, annotate nothing, and configure nothing: it follows from the surrogate keys both the query and the mutation were tagged with.

This page states exactly which keys a purge follows, so you can predict what a given mutation clears and, just as importantly, what it does not.

Which keys a purge follows

Every response GraphPilot handles is tagged with a set of surrogate keys, and the kind of key decides what a purge can do with it later. Automatic invalidation follows exactly two kinds: **entity keys** such as `Product:id:p01`, found in the response body, and **origin keys** your origin names in a `Surrogate-Key` header. Empty-list keys (`!Product`) are purged as targets. Type keys, operation keys, static keys and cache keys are never purged automatically.

The distinction that matters: an **entity key names something that exists**, so purging it removes exactly the responses that contained that object. A type key or an operation key names a shape, so purging it would remove every response of that shape at once. GraphPilot never does the second one on your behalf.

Composite and nested keys follow the same rule. A type keyed on two fields produces `PriceList:currency:EUR:region:de`, and a key field reached through a parent field produces `Product:category.id:audio`. Both are entity keys and both purge automatically.

So do keys an object inherits from an interface or a union. With `interface Medium @surrogateKey(of: "isbn")`, a `Book` carries `Medium:isbn:111` and `Book:isbn:111` wherever it appears, whether the query asked for `media` or for `book`, and so does a mutation that returns it.

> **Note**
> You do not have to select the key fields for this to work. Selection set expansion injects them into the query, reads them, and strips them again before the response reaches the client.

What a mutation purges

A mutation response is tagged exactly like a query response, from the objects it returns. Whenever a mutation response comes back, GraphPilot purges:

1. **Every entity key on it.** `setPrice` returning `Product:id:p01` purges every cached response anywhere that contained product `p01`, whatever operation produced it. A mutation returning a `Book` whose key is declared on its interface purges `Medium:isbn:111` and `Book:isbn:111`, which reaches the cached `media` lists and the cached `book` answers that showed it.
2. **Every origin key on it**, so your origin can name additional things to clear by setting a `Surrogate-Key` header on the mutation response.
3. `![Typename]` **for every type the mutation touched**, derived both from the entity keys it returned and from the types its selection names. The root type counts too, so every mutation purges `![Mutation]` as well, which no cached response carries.

Everything else on the response rides along to describe it and purges nothing. A failed mutation usually returns no objects, so it usually carries nothing to purge.

Worked example

Take a `setPrice` response tagged `m:setPrice`, `Product:id:p01`, and the type keys `Mutation`, `Product` and `Review`, because the mutation selected the product's reviews and the product has none yet. It purges exactly four keys:

```
Product:id:p01   the product that changed
![Mutation]      derived from the root type; no cached response carries it
![Product]       cached responses whose product list was empty
![Review]        cached responses whose review list was empty
```

`m:setPrice` is not purged, because nothing is cached under a mutation name. `Review` is not purged either: the mutation selected reviews, and clearing every cached response that so much as mentions reviews would empty most of your cache on a single write. What it purges instead is `![Review]`, the entry that claims there are none.

Why the empty-list key exists

An entity key can only invalidate a response that already contained that object, which leaves one case uncovered: a response that is correct only because the object did not exist yet.

```
query { product(id: "p01") { reviews { id body } } }   # → { "reviews": [] }
```

That response is tagged `![Review]`. When a mutation later creates a review, the new review's entity key clears nothing (no cached response contained it), but `![Review]` clears the cached "there are none" answer. Without it, an empty list would stay empty for its whole TTL and the first item your users create would appear to vanish.

Mutations that return a payload type

Many schemas wrap the changed object in a payload type, in the Relay style:

```
type Mutation {
  setPrice(input: SetPriceInput!): SetPricePayload
}

type SetPricePayload {
  product: Product
  clientMutationId: String
}
```

A client that selects only `clientMutationId` never receives the `Product`, so its response would carry no `Product:id:…` key and purge nothing. To prevent that, GraphPilot looks one level below a mutation's return type for types that carry surrogate keys, their own or ones inherited from an interface or a union. When it finds one, it adds that field and its key fields to the query it sends your origin, reads the keys from the answer, and purges them, even if the client did not select the field. This works the same way as selection set expansion.

`surrogate_key.mutation_payload_depth` sets how far down GraphPilot looks:

Value	Effect
`0`	Off. A mutation purges only what the client selected.
`1` (default)	The payload's own fields: `SetPricePayload.product`. This reaches the object the mutation changed.
`2`	One level further: the changed object's single-valued fields, such as its category.
`3` or more	Each further level adds the single-valued fields of the level above it.

gpilot.toml

```
[surrogate_key]
mutation_payload_depth = 1
```

Raising it to `2` is a broad tool. It purges the parent itself, so every cached response that mentions that category goes, not only the ones that listed the changed product. If a cached response reads your object through its parent, the narrower fix is a key on the object's own type that names the parent, for example `@surrogateKey(of: "category.id")` on `Product`. See keys through a relation.

The descent has limits:

- **A mutation that returns a keyed type directly gets no descent.** `setPrice( … ): Product` is treated as returning the entity itself. Its own keys are purged as usual, but field-level rules on it (such as `category: Category @surrogateKey(of: "id")`) add nothing the client did not select.
- **A payload type with keys of its own is treated the same way.** A payload that has its own `id` field is taken for an entity, and the object nested inside it is not reached. Select the nested object in your mutation, or return it directly.
- **Fields that require arguments are never added**, because GraphPilot cannot know which values to send.

What is not invalidated automatically

This is the part to read closely.

A create does not appear in a cached list that was not empty

A cached list of a product's reviews holding three reviews is tagged with those three entity keys, the type key `Review`, and `q:product`. An `addReview` mutation creating a fourth review purges the new review's entity key, which no cached response contains, and `![Review]`, which does not match a non-empty list. The cached list keeps its three entries until its TTL expires.

Updates and deletes do not have this problem: they operate on an object that is already in the cached response, so its entity key is on both sides.

> ⚠ **This is the one to plan for**
>
> If a list has to show new items immediately, automatic invalidation alone will not do it. Give the mutation a key that reaches the list, either by returning the affected parent object so its entity key is purged, or by setting a `Surrogate-Key` header on the mutation response naming a key the list also carries. Both purge automatically. Failing that, purge it explicitly through the API or the CLI. How to tag a list so a create reaches it is covered in the guide lists and creates.

A mutation that returns nothing purges nothing

The purge is derived from the mutation's response body. A mutation declared as `deleteReview(id: ID!): Boolean!` returns no object, carries no entity key, and therefore invalidates nothing at all. Return the affected object, or its parent, and the key comes with it.

Static keys are never purged automatically

A key from `@surrogateKey(static: "catalogue")` groups cached responses deliberately and by hand, so it is cleared deliberately and by hand, through the purge API. A mutation carrying the same static key does not sweep it, because a static key usually spans far more than the mutation touched.

There is no purge by type or by operation

Neither `Product` nor `q:products` is ever purged automatically. Both are available as purge targets through the API when you decide you want that reach.

Hard, and always

Automatic purges are **hard**: the entry is removed, and the next request for it goes to your origin as a `MISS`. There is no soft-purge (serve-stale-while-refreshing) option on the automatic path.

The purge also runs when your billing account is over its request quota and the cache has stopped serving. That is deliberate. A mutation during such a window still has to invalidate entries written before it, or they would come back live and stale the moment normal service resumes. It is not recorded in the purge history while the service runs lean.

Seeing what was purged

When request telemetry is stored (on every plan, within its logged-request allowance, and not in lean mode), every automatic purge is recorded in the service's purge history with the cleartext keys it covered and the mutation request that caused it. You find it on the **Purges** page of the service in the portal, marked **Automatic**. The mutation request's **Edge codes**, in the **Overview** tab of its Insights request detail, show a `CACHE_PURGED` entry for it with the number of keys, not the keys themselves.

A mutation that carried no purgeable key records no purge at all, rather than a purge of zero keys. So if a mutation has no entry in the history, it had nothing to clear, and that is usually the answer to "why is my list still stale". See verifying a purge.

End to end: a price change

A product catalog, served by the service `shop-api`. Its schema keys each product three ways and returns the changed product from the mutation:

schema.graphql

```
type Product
  @cacheControl(maxAge: 3600, scope: PUBLIC, swr: 3600)
  @surrogateKey(of: "id")
  @surrogateKey(of: "category.id")
  @surrogateKey(static: "catalogue") {
  id: ID!
  name: String!
  price: Int!
```

```
  category: Category!
}

type Mutation {
  setPrice(id: ID!, price: Int!): Product!
}
```

gpilot.toml

```
[response]
surrogate_key_header = true   # show the keys on the response while you try this
```

1. A reader caches the product. A client asks for the price without selecting `id` or the category:

```
curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"query Price { product(id: \"p02\") { name price } }"}'
```

```
gp-cache: MISS
gp-surrogate-key: ck:0ed2ffb7…7884 q:product Product Query catalogue
Product:category.id:audio Product:id:p02
```

Both entity keys are there although the client selected neither field: selection set expansion fetched `id` and `category { id }` for them. The same request again answers `gp-cache: HIT`.

2. An editor changes the price.

```
curl -sS "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"mutation { setPrice(id: \"p02\", price: 2450) { id price } }"}'
```

The response carries the same entity keys, `Product:id:p02` and, through the expanded `category { id }`, `Product:category.id:audio`, beside `m:setPrice`, the type keys `Mutation` and `Product`, and the static key `catalogue`. Before the response reaches the editor, GraphPilot purges exactly four keys:

```
Product:category.id:audio
Product:id:p02
![Mutation]
![Product]
```

The type keys, `m:setPrice` and `catalogue` are not purged.

3. The next reader gets the new price. The `Price` query from step 1 now answers `gp-cache: MISS` with `"price": 2450`, and is cached again.

Note the reach of `Product:category.id:audio`: every cached response holding any product of the audio category is purged too, not only those holding `p02`. That is what the category key on `Product` is for (one purge for a whole category), and it makes every product write a category purge. If a price change should not reach the rest of its category, leave the category key off `Product` and give the pages that list a category a key of their own; see what purges together.

4. What the portal shows. On the service's **Purges** page the purge is listed with the trigger **Automatic**, and its detail panel lists the keys and links to the `setPrice` request under **Triggered by**. In Insights, that request's **Edge codes** on the **Overview** tab show `CACHE_PURGED` with the number of keys. Both need the request to be recorded; see seeing what was purged.

Related

- Surrogate keys: the key kinds and where they come from
- The Surrogate-Key header: keys for what a mutation affects but does not return
- Purging through the API and CLI: for changes that never go through a mutation
- Verifying a purge: the purge history and how to check a purge worked

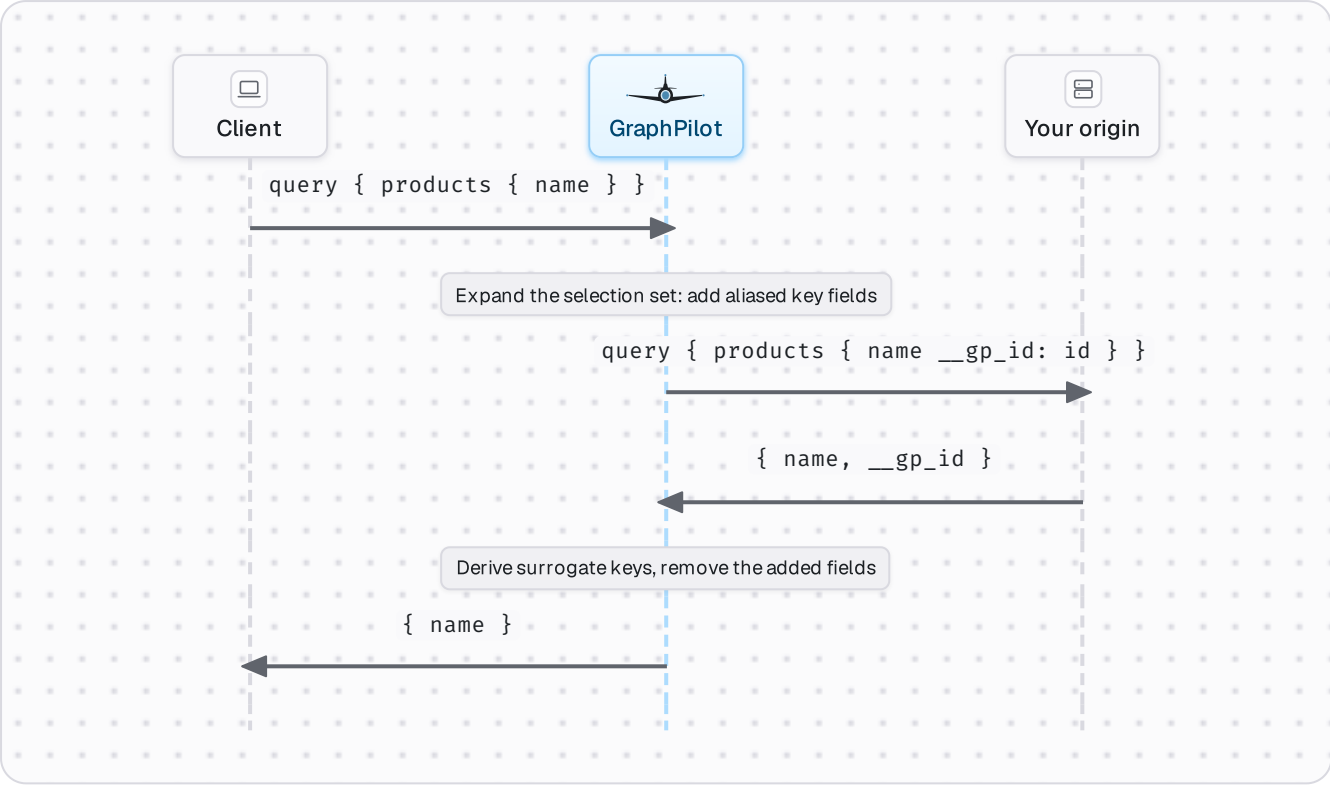
---

INVALIDATION

Selection set expansion

> **GraphQL only**

Selection set expansion makes sure every field a surrogate key needs is in the query your origin receives, even when the client did not select it. Your clients write the queries they need, and GraphPilot still tags every cached response with the keys that let a write purge it.



Your client gets exactly what it asked for; the entry gets its keys.

Why it matters

A surrogate key is built from field values in the response, such as a product's `id`. If the client did not select that field, the value is not in the response, and without expansion the cached entry would carry no key for it. A later write to that product could not purge it.

Expansion removes that dependency on the client:

- **Keys do not depend on what the client selected.** The same operation is tagged with the same keys whether or not the client asked for `id`, so a purge reaches it either way.
- **Clients need no cache-specific fields.** Nobody has to remember to select `id` "for the cache", and adding such a field to one client does not split the cache from the clients that do not.
- **Composite and related keys work.** A key built from a relation, such as `category.id`, gets the relation's field added too.

How it works

When you deploy, the deployment build reads your schema and every configured source of surrogate keys: the `@surrogateKey` directive, a directive of your own, and the global rules for scalars, enums and field names. From these it knows which fields each key needs.

On every query or mutation that reaches your origin, GraphPilot compares the client's selection set with those fields and adds the missing ones. Each added field gets an alias with the prefix set by `origin.field_prefix` (default `__gp_`), so it cannot collide with a field the client selected. The prefix is reserved for this: a query with a

response key that starts with it is refused with `400` and `RESERVED_FIELD_PREFIX` before it reaches your origin, since its answer would otherwise be removed along with the added fields. The response key is a field's alias when it has one and its name otherwise, so an unaliased schema field whose name starts with the prefix is refused as well. Keep that in mind before you set a short prefix such as `data_`: every query that selects such a field without an alias is then refused, and the fix the edge's message asks for ("Choose a different alias") means aliasing that field. When the response comes back, GraphPilot reads the key values from the added fields, derives the surrogate keys, and removes the added fields from the response, and from any `errors` entry that mentions them, before the client receives it.

> **Only where the client already selects data**
>
> GraphPilot adds key fields only under fields the client selected. It does not walk into parts of the schema the query never reached. The one exception is a key that reads a relation, such as `category.id`: then the relation field is added as well, with only the fields the key needs inside it. An added relation contributes only to the key that asked for it. It adds no keys of the related type, whether they come from a `@surrogateKey` on that type, on one of its fields, or from the automatic rules: those appear only where the client selected the relation itself.

Example

Consider this schema, from a product catalog:

schema.graphql

```graphql
type Query {
  products(category: ID): [Product!]!
}

type Product @surrogateKey(of: "category.id") {
  id: ID!
  name: String!
  price: Int!
  category: Category!
}

type Category {
  id: ID!
  name: String!
}
```

A client sends:

Query from the client

```graphql
query {
  products {
    name
```

```
    price
  }
}
```

GraphPilot adds what the keys on `Product` need: `id` for the automatic `Product:id` key (an `ID` field), and `category.id` for the key the directive declares.

Expanded query sent to your origin

```
query {
  products {
    name
    price
    __gp_id: id
    __gp_category: category {
      id
    }
  }
}
```

The alias sits on the field GraphPilot added. Fields inside an added relation, here `id` inside `__gp_category`, keep their names, because the whole relation is already set apart by its alias. No `__typename` is added here: `Product` and `Category` are object types, so the type of every position is known from the query. GraphPilot adds `__typename` only where a key sits under an interface or a union, to tell the concrete types apart.

Your origin responds:

Response from your origin

```
{
  "data": {
    "products": [
      {
        "name": "Studio headphones",
        "price": 1999,
        "__gp_id": "p01",
        "__gp_category": { "id": "audio" }
      },
      {
        "name": "Standing desk",
        "price": 39900,
        "__gp_id": "p06",
        "__gp_category": { "id": "desks" }
      }
    ]
  }
}
```

From it, GraphPilot derives these surrogate keys for the cached entry:

What a purge by this key reaches	Surrogate keys
Products with a specific id	`Product:id:p01` `Product:id:p06`
Products of a specific category	`Product:category.id:audio` `Product:category.id:desks`

There is no `Category:id:audio` among them, although `Category.id` is a key field: the `category` GraphPilot added is there for the `Product` key alone. A client that selects `category { id name }` itself gets `Category:id:audio` on its entry as well.

Before the response leaves, GraphPilot removes every added field, so the client receives exactly what it asked for:

Response sent to the client

```
{
  "data": {
    "products": [
      { "name": "Studio headphones", "price": 1999 },
      { "name": "Standing desk", "price": 39900 }
    ]
  }
}
```

The `gp-surrogate-key` header shows the same thing on a real response. A client asking only for `product(id: "p02") { name price }` gets back an entry tagged with both keys:

```
gp-surrogate-key: ck:0ed2ffb7…7884 q:product Product Query Product:category.id:audio
Product:id:p02
```

Purging a category's products

`Product:category.id:audio` is on every entry that contains a product of the audio category, whichever query produced it and whether or not the client selected the category. One purge reaches all of them:

```
gpilot purge --service shop-api Product:category.id:audio
```

It is a different key from `Category:id:audio`, which is on entries that contain the category object itself, such as a page that selected `category { name }`. A `renameCategory` mutation returns the category, so it purges `Category:id:audio` and reaches the pages that show the old name. A price change in the category is the job of `Product:category.id:audio`. Pick the key that matches what a write actually changes.

Fragments

GraphPilot never adds fields to a fragment definition. A fragment can be spread in several places, so a field added to its definition would change every place it is used. Instead, the missing fields are added to the selection set at the place where the fragment is used, beside the spread. This is the same whether the `@surrogateKey` sits on the type or on the field that returns it.

Query from the client

```
fragment ProductFields on Product {
  name
}

query {
  products {
    ... ProductFields
  }
}
```

Expanded query sent to your origin

```
fragment ProductFields on Product {
  name
}

query {
  products {
    __gp_id: id
    __gp_category: category {
      id
    }
    ... ProductFields
  }
}
```

A fragment spread in two places gets the added fields at both places. A field that is already selected where the fragment is used, directly or through an inline fragment, is reused rather than added a second time.

A fragment cannot spread itself, directly or through another fragment: the GraphQL specification forbids fragment cycles, and your origin rejects such a document.

Limitations

- **A response GraphPilot cannot read keeps the added fields.** Removing them needs the response parsed as JSON. When that fails, the client receives the body exactly as your origin sent it, including the aliased key fields (`__gp_id` and the like), and the response is not stored (`CACHE_SKIPPED_UNPARSEABLE_GRAPHQL_RESPONSE`). The causes are a body larger than 133 MiB and a JSON

integer larger than 18446744073709551615. A body that is not JSON at all, such as an HTML error page, carries no added fields in the first place. The added fields are only ever key fields of the objects the client selected, never data from anywhere else, and because the prefix is reserved a client cannot mistake one for a field of its own. To avoid it, return very large integers as strings (an `ID` or a custom scalar) and keep responses well below the size limit. See refusals that depend only on what the response carries.

Related

- Surrogate keys
- The `@surrogateKey` directive
- Automatic invalidation
- How a request flows

INVALIDATION · SURROGATE KEYS

Surrogate keys

A surrogate key is a label on a cached response. The cache key identifies exactly one cached response; a surrogate key can sit on many of them. Purging a surrogate key removes every response that carries it, so the keys decide what an invalidation can reach.

Ways to control this

Through	What it controls
Your origin	Keys of your own in the `Surrogate-Key` header.
Your GraphQL schema	`@surrogateKey`: key fields (`of`), fixed keys (`static`), and fields left out (`ignore`).
`gpilot.toml`	Which scalars and field names count as key fields, and which types and fields are left out (automatic key fields).
The API, the CLI and the portal	Purging by key: `gpilot purge <key>`, `purgeBySurrogateKey`, or `POST /purge` with `keys` (purge on demand).

Kinds of keys

Key	Example	Comes from	Plain HTTP	Purged by a write's response
Cache key	`ck:GET:/products/42`, `ck:9f86d0`	The key the entry is stored under	Yes	No
Origin key	`catalog-v3`	The `Surrogate-Key` header on your origin's response	Yes	**Yes**, on a plain HTTP write and on a GraphQL mutation

Key	Example	Comes from	Plain HTTP	Purged by a write's response
Entity key	`Product:id:p01`	A key field found in a GraphQL response	No	**Yes**, on a mutation
Empty-list key	`![Review]`	A list in a GraphQL response that came back empty	No	Only as a target
Type key	`Product`	The types a GraphQL operation selects, whether or not any came back	No	No
Operation key	`q:product`, `m:setPrice`	The root field a GraphQL operation names	No	No
Static key	`catalogue`	`@surrogateKey(static:)` in your schema	No	No

A key that is not purged by a write's response is still a purge target: name it through the purge API or the CLI when you want that reach. What a GraphQL mutation purges on its own is the subject of automatic invalidation.

> ⚠ **Caution**
>
> Keys share one namespace, whoever adds them. A key your origin sends that looks like `Product:id:1` or `q:products` is indistinguishable from the generated one, and a purge of either reaches both. Give your own keys a prefix if you want them kept apart.

Reading the keys of one response

With `response.surrogate_key_header` switched on, every response lists its keys in `gp-surrogate-key`. A product page that asks for one product, its category and its reviews comes back with:

```
gp-surrogate-key: ck:95f9f0f4…72b0 q:product Category Product Query Review catalogue
Category:id:audio Product:category.id:audio Product:id:p01 Review:id:r01
Review:productId:p01
```

Key	Kind	A purge of it reaches
`ck:95f9f0f4…72b0`	Cache key	This query, with every variable set.
`q:product`	Operation key	Every cached `product( … )` answer, whatever the id.
`Category`, `Product`, `Query`, `Review`	Type keys	Every cached response that selected that type.

Key	Kind	A purge of it reaches
`catalogue`	Static key, from `@surrogateKey(static: "catalogue")`	Everything the schema groups under it: every product and category.
`Category:id:audio`, `Product:id:p01`, `Review:id:r01`	Entity keys	Every cached response that contains that object.
`Product:category.id:audio`	Entity key, from `@surrogateKey(of: "category.id")`	Every cached response that contains a product of the audio category.
`Review:productId:p01`	Entity key, from `@surrogateKey(of: "productId")` on `Review`	Every cached response that contains a review of `p01`.

Purging `Product:id:p01` does not reach `Review:productId:p01`: a purge matches keys exactly, type name included. To clear a product and its reviews in one call, name both.

Keys on every entry

`purgeAll` and `gpilot purge --all` remove every stored entry; there is no key to name for that.

The cache key

Every stored response carries its own cache key as a surrogate key, with a `ck:` prefix, so one response can be purged on its own:

- **Plain HTTP:** `ck:<METHOD>:<path>`, the method and the path the request was sent to, query string included, for example `ck:GET:/products/42?color=blue`. A request that carries a body adds `#` and a SHA-256 of the body. Scheme and hostname are not part of the key, so one path is one entry whether the client used the service hostname or a custom domain. An origin that answers the same path differently per hostname can put them back with `host = "include"`: the key is then the full URL, such as `ck:GET:https://catalog-api.k7m2qp.graphpilot.cloud/products/42?color=blue`, and each hostname keeps its own entry. This is how you purge one plain HTTP response without a key of your own. There is no purge-by-URL call; you name the cache key, and the surest way to get its spelling is to copy it from the request detail in the portal or from the `gp-surrogate-key` header.
- **GraphQL:** `ck:<queryHash>` covers every variable set of a query, and a longer form that adds the operation name and the variables hash (`ck:<queryHash>#<operationName>:<variablesHash>`, the variables hash being 32 hex digits) addresses exactly one cached response. The variables hash here is a keyed digest that differs per service (see what the entry is keyed by), so copy it from the request detail in the portal or from request telemetry; it cannot be computed from the variables. Neither form carries the path or the query string the response was stored under, so a purge reaches it on every route: one purge of `ck:<queryHash>` clears that query on `/graphql`, `/graphql/acme` and `/graphql?locale=de` alike.

A cache key becomes a surrogate key only when it is printable ASCII and at most 1024 bytes. A normal URL is both; an entry stored under a longer one carries no `ck:` key, and only a key your origin named or a purge-all reaches it. See purging one cached response by its cache key.

Keys from your origin

Your origin adds keys with a `Surrogate-Key` header on its response. On a read they tag the entry, on a write they purge. The header and its format are on the Surrogate-Key header.

Keys from your schema and data

> **GraphQL only** For plain HTTP, name keys in the Surrogate-Key header on your origin's response.

GraphPilot generates these keys from your schema and from the response data, without any annotation. You can declare more with the `@surrogateKey` directive.

Operation keys

Every query response carries `q:<field>` for its root field, for example `q:product`. It lets you purge a query **regardless of which arguments were used**. It is the field name the schema defines, not an alias the client chose.

A mutation response carries `m:<field>`, for example `m:setPrice`. It describes the response, so its root field is visible in telemetry and on the response. Nothing is ever stored under it, because a mutation is never cached.

Type keys

Every type an operation selects adds a key with its name, such as `Product`, whether or not any object of that type came back. **An operation that returns an empty list still carries it.** Under a field of an interface or union type, the type selected is the interface or union, plus each member the query names in a fragment: `{ media { title } }` carries `Medium` and no `Book`, whatever the response holds.

Entity keys

`<typename>:<keyField>:<keyValue>` names one object. For a type `Product` whose `id` field is `p01`, the key is `Product:id:p01`.

Without any directive, two rules decide which fields are key fields: a field whose type is a key scalar (`ID` by default) and a field whose name is a key name (`id`, `_id` and `key` by default). Both are configurable, and both can be switched off where they pick a field that identifies nothing; see `ignore`.

A key can combine several fields (`PriceList:currency:EUR:region:de`) or reach through a related object (`Product:category.id:audio`). Both are entity keys. A composite key is only reached by a purge naming it in full; see composite keys.

Empty-list keys

A list that came back empty adds `![<typename>]`, so creating the first object of that type can clear the cached “there are none” answer. See why the empty-list key exists.

What a key may contain

Every key GraphPilot stores is **printable ASCII** (`0×21` to `0×7E`). How a key gets there depends on where it comes from.

Keys from your schema and your data are folded

This covers GraphQL keys built from field values, `static` keys from the directive, and keys from your origin's `Surrogate-Key` header. Every character is folded into printable ASCII:

- **Printable ASCII** (`!` through `~`) is kept exactly as written, casing included. `:` , `{` , `-` , `_` and every other printable symbol pass through unchanged.
- **Latin letters with a diacritic** fold to their base letter: `Müller` becomes `Muller` , `Łódź` becomes `Lodz` , `Ångström` becomes `Angstrom` . The decomposed spelling (a base letter followed by a combining mark) reaches the same key, because the combining mark is dropped.
- **Ligatures and similar letters** take their conventional ASCII spelling: `ß` becomes `ss` , `æ` becomes `ae` , `œ` becomes `oe` , `þ` becomes `th` .
- **Every other character** becomes one underscore: a space, a tab, a non-breaking space, `×` , Cyrillic, CJK, emoji. A product titled `Blue Shirt` is tagged `Product:title:Blue_Shirt` .

The folding is deliberately lossy. `Blue Shirt` and `Blue_Shirt` share a key, as do `Müller` and `Muller` , and two Cyrillic words of the same length collapse onto the same run of underscores. A collision means a purge clears a little more than you named, never less.

Casing is never changed, neither in your schema's field names nor in the response data, so `Product:sku:AB1` and `Product:sku:ab1` are two different keys.

Keys from your origin are folded the same way

Keys in the `Surrogate-Key` header go through the same folding, so `Brand:Müller` from your origin and `Brand:Müller` from your data reach the same key. ASCII spaces, tabs and commas separate keys in the header, so none of them can be part of one. See the header's format.

A purge is folded the same way

The purge API accepts any characters and folds them before it runs, so you can purge with a value exactly as it appears in your data: `Product:title:Blue Shirt` reaches the entry tagged `Product:title:Blue_Shirt` .

Keep keys to 1024 characters

A key your schema derives is never shortened. One that is longer than 1024 bytes after folding is still attached to the response, but a purge naming it is refused. (A cache key works differently: one that long is not attached at all, as above.)

Keep a key under 1024 bytes after folding: a few characters fold to two (`ß` becomes `ss`). Free text such as titles makes a poor key in any case; prefer an identifier.

How many keys fit on one response

The keys of one response share a budget of 16,000 bytes. GraphPilot stores them as short hashes to fit, and a response whose keys do not fit even then is not cached rather than cached with keys missing. The hash widths, how many keys each holds and what to do about a response that does not fit are on limits.

Related

- The Surrogate-Key header: keys from your origin
- Keys from the schema: declaring keys with `@surrogateKey`
- Debugging: reading the keys a response carries
- Purging through the API and CLI: naming keys in a purge

INVALIDATION · SURROGATE KEYS

Keys from the schema

> **GraphQL only** For plain HTTP, name keys in the Surrogate-Key header on your origin's response.

Your schema already describes which fields identify an object. The `@surrogateKey` directive lets you state the rest there too: which fields form a key, which fixed keys a type or field carries, and where the automatic key rules should stay out. GraphPilot reads it when it builds a deployment, so the keys change with the schema and there is no second place to keep in sync.

If your schema already marks keys with another directive, such as `@key` from Apollo Federation, you can configure GraphPilot to read that one instead.

What purges together

A surrogate key decides which cached responses one purge removes together. Start from that question, and the place to write the directive follows:

A purge should reach	Declare	Key
One object, wherever it appears	Nothing, or a key field: `id: ID! @surrogateKey`	`Product:id:p01`
Objects that share values	`@surrogateKey(of: "category.id")` on the type	`Product:category.id:audio`
Every implementor or member at once	The key on the interface or union	`Medium:genre:fiction`
What one field of a parent returns	`@surrogateKey(of: "id")` on the field	`Category:featured.id:p01`
A group you name by hand	`@surrogateKey(static: "catalogue")`	`catalogue`

Each directive defines exactly one key. That key can be based on a single field or on a combination of fields. To give the same type more than one key, repeat the directive; it is `repeatable`.

The first four are built from values in the response, so they are entity keys: a mutation whose response carries the same key purges them automatically. A `static` key is only ever purged when you name it.

One object

Automatic key fields

A type with an `id: ID!` field is keyed `Type:id:<id>` without anything written in the schema, by the two automatic rules for key scalars and key names. Where one of them picks a field that identifies nothing, switch it off with `ignore`.

A field as a key field

On a field that returns a scalar or an enum, the directive makes that field a key field of the enclosing type. `static` is allowed there and attaches a fixed key that is included only when the field is queried; `of` has no effect on such a field and must not be used.

schema.graphql

```
type Product {
    id: ID! @surrogateKey # → Product:id:{id}
    name: String
    brand: String! @surrogateKey # → Product:brand:{brand}
}
```

Field or type

Annotating the field is equivalent to a key with `of` on the enclosing type, and it reads more clearly:

schema.graphql

```
type Product @surrogateKey(of: "id") {      # ← Equivalent but less clear
  id: String! @surrogateKey                 # ← Recommended for scalars/enums
  name: String
}
```

Every field of a scalar or enum type

Annotating a scalar makes every field of that type a key field of the object it sits on. It does the same as listing the scalar in the configuration, but keeps it in your schema. An enum type works the same way.

schema.graphql

```
scalar SKU @surrogateKey # Every field returning this type becomes a key field
```

```
type Product {
    sku: SKU! # → Product:sku:{sku}
    name: String
    category: Category!
}

type Variant {
    parentSku: SKU! # → Variant:parentSku:{parentSku}
    color: String
}
```

schema.graphql

```
enum StockStatus @surrogateKey {
    # Every field returning this type becomes a key field
    IN_STOCK
    SOLD_OUT
}

type Product {
    stockStatus: StockStatus! # → Product:stockStatus:{stockStatus}
    name: String
}

type Inventory {
    status: StockStatus! # → Inventory:status:{status}
    available: Int
}
```

ENUM_VALUE is not supported

The directive applies to an enum type, never to a single enum value, so a key that appears only when one particular value comes back is not something GraphPilot does. Declaring `ENUM_VALUE` among the directive's locations reports a schema error in the deployment's diagnostics (the deployment still succeeds, and the location contributes nothing). Writing the directive on an enum value without declaring that location is ignored with no diagnostic at all.

Objects that share values

Composite keys

A key built from several fields is a composite key: one key per combination of values. A type can carry single-field keys and composite keys side by side, one directive each:

schema.graphql

```
# Single-field keys
```

```
type Product {
    id: String! @surrogateKey               # → Product:id:{id}
    sku: String! @surrogateKey              # → Product:sku:{sku}
}

# Composite keys (multiple fields)
type PriceList
  @surrogateKey(of: "region currency")      # → PriceList:currency:{currency}:region:
{region}
  @surrogateKey(of: "channel region")       # → PriceList:channel:{channel}:region:{region}
{
    region: String!
    currency: String!
    channel: String!
}
```

The fields of a composite key are sorted alphabetically by name, so `region currency` and `currency region` produce the same key. When one of them is a list, the key multiplies out into one key per combination: `@surrogateKey(of: "brand tags")` with `brand: "acme"` and `tags: ["new", "bestseller"]` yields `Product:brand:acme:tags:new` and `Product:brand:acme:tags:bestseller`.

> **Purge a composite key in its full form**
>
> `PriceList:currency:{currency}:region:{region}` is only reached by a purge naming the entire key. `PriceList:region:{region}` alone reaches nothing; to purge by one field, declare a key for that field as well.

Keys through a relation

On an object type the directive needs at least one argument, `of`, `static` or both. A bare `@surrogateKey` on an object does not know how to build a key, and none is created. `of` can reach into a related type; selection set expansion fetches those fields when the client did not select them.

schema.graphql

```
type Product
    @surrogateKey(
        of: "brand category_id" # → Product:brand:{brand}:category_id:{category_id}
    )
{
    brand: String!
    category_id: ID! # → Product:category_id:{category_id}, automatic: an ID field is a key
field
}
```

This key reaches products that share both brand and category, never products that share only one of them. `category_id` is an `ID`, so the automatic rule for key scalars also keys `Product:category_id:{category_id}` on its

own; a foreign key of another scalar type gets no such key. For a purge by brand alone or by category alone, declare those keys as well:

schema.graphql

```
type Product
    @surrogateKey(
        of: "brand category_id" # → Product:brand:{brand}:category_id:{category_id}
    )
{
    brand: String! @surrogateKey # → Product:brand:{brand}
    category_id: ID! @surrogateKey # → Product:category_id:{category_id}
}
```

Use case	Surrogate key
Products of a specific **brand** and **category**	`Product:brand:{brand}:category_id:{category_id}`
Products of a specific **category**	`Product:category_id:{category_id}`
Products of a specific **brand**	`Product:brand:{brand}`

These examples use a foreign key field, `category_id`. Traversing the relation instead (`category.id`) makes it clear that the key follows the relationship rather than a raw identifier, and it needs no extra field on the type:

schema.graphql

```
type Product
    @surrogateKey(
        of: "brand category.id" # → Product:brand:{brand}:category.id:{category.id}
    )
{
    brand: String!
    category: Category!
}

type Category {
    id: String!
    name: String
}
```

The same with separate keys for the brand and the category:

schema.graphql

```
type Product
    @surrogateKey(
        of: "brand category.id" # → Product:brand:{brand}:category.id:{category.id}
    )
```

```
{
    brand: String! @surrogateKey # → Product:brand:{brand}
    category: Category!
}

type Category {
    id: String! @surrogateKey # → Category:id:{id}
    name: String
}
```

Every implementor or member at once

A product catalog rarely has an interface or a union, so this section, and the cases further down that read a field returning one, use a media library instead: books, magazines and movies.

Interface

On an interface the directive works as on an object type, and every implementor inherits the keys, so you declare a key once for all of them. An implementor's own directives add to the inherited keys, and `static` on an interface gives all implementors a shared key.

An inherited key is named after the interface, with the implementor's own name beside it: `Medium:genre:fiction` and `Book:genre:fiction` for a book. Both are attached wherever the query reaches the implementor: through a field of the interface type (`media: [Medium!]`), through a field that returns the implementor itself (`book: Book`), or further down a selection. A purge by either key therefore reaches every cached response that shows the object, whichever field the query used.

Under a field of the interface type, a query that names no implementor in a fragment (`{ media { title } }`) gets the interface's keys and their implementor twins. The keys an implementor declares itself, or inherits from another interface or a union, are added where the query names it: `{ media { ... on Book { title } } }`.

The automatic keys are not inherited. An interface with an `id: ID!` field is keyed `Node:id:<id>` where it is selected, and every implementor declares the same field and carries `User:id:<id>` under its own name, so a `User` reached directly carries no `Node:id` key.

schema.graphql

```
interface Medium
    @surrogateKey(
        of: "author.id genre" # → Medium:author.id:{author.id}:genre:{genre}
    )
{
    genre: String!
    author: Author!
}

# The keys below apply to a Book or Magazine wherever the query reaches it: through a field
# of type Medium, or through a field returning Book or Magazine itself.
```

```
type Book implements Medium {
    # → Book:author.id:{author.id}:genre:{genre}
    genre: String!
    author: Author!
}

type Magazine implements Medium {
    # → Magazine:author.id:{author.id}:genre:{genre}
    genre: String!
    author: Author!
}

type Author {
    id: String!
    name: String
}
```

Keys on the interface's fields are inherited too, which gives a purge by genre, by author, or by both, across every implementor and on the interface itself:

schema.graphql

```
interface Medium
    @surrogateKey(
        of: "author.id genre" # → Medium:author.id:{author.id}:genre:{genre}
    )
{
    genre: String! @surrogateKey # → Medium:genre:{genre}
    author: Author! @surrogateKey(of: "id") # → Medium:author.id:{author.id}
}

# The keys below apply to a Book or Magazine wherever the query reaches it: through a field
# of type Medium, or through a field returning Book or Magazine itself.
type Book implements Medium {
    # → Book:author.id:{author.id}:genre:{genre}
    genre: String! # → Book:genre:{genre}
    author: Author! # → Book:author.id:{author.id}
}

type Magazine implements Medium {
    # → Magazine:author.id:{author.id}:genre:{genre}
    genre: String! # → Magazine:genre:{genre}
    author: Author! # → Magazine:author.id:{author.id}
}

type Author {
    id: String!
    name: String
```

```
}
```

Use case	Surrogate key
Media of a specific **author** and **genre**	`Medium:author.id:{author.id}:genre:{genre}`
Media of a specific **author**	`Medium:author.id:{author.id}`
Media of a specific **genre**	`Medium:genre:{genre}`
Books of a specific **author** and **genre**	`Book:author.id:{author.id}:genre:{genre}`
Books of a specific **author**	`Book:author.id:{author.id}`
Books of a specific **genre**	`Book:genre:{genre}`
Magazines of a specific **author** and **genre**	`Magazine:author.id:{author.id}:genre:{genre}`
Magazines of a specific **author**	`Magazine:author.id:{author.id}`
Magazines of a specific **genre**	`Magazine:genre:{genre}`

Through a field returning the implementor the field keys apply as well: `Medium.author @surrogateKey(of: "id")` keys `book { author }` with `Medium:author.id:{author.id}` and `Book:author.id:{author.id}`, exactly as it keys `media { author }`.

A type that also defines its own `@surrogateKey` gets both the inherited and its local keys. The same holds for a union member.

Union

On a union the directive is a shared contract across the members, with `of` checked per member. Every member that has the fields the key names gets the key, each resolving to a scalar, an enum, or a list of either; a member that lacks them gets none. The keys are named after the union, with the member's own name beside them, and they are attached wherever the query reaches the member. Under a field of the union type that is the members the query names in a fragment: `{ contents { __typename } }` carries none of them. A field that returns a member type itself (`book: Book`) carries the union's keys as well as the member's own.

schema.graphql

```
union Content
    @surrogateKey(of: "author.id genre") # → Content:author.id:{author.id}:genre:{genre}
=
    | Book
    | Magazine
    | Movie

# The keys below apply to a member reached through a field of type Content and
# named in a fragment of the query, or through a field returning the member itself.
type Book {
```

```
    # → Book:author.id:{author.id}:genre:{genre}
    genre: String!
    author: Author!
}

type Magazine {
    # → Magazine:author.id:{author.id}:genre:{genre}
    genre: String!
    author: Author!
}

# No key is generated for this type
# because the path `author.id` does not exist on `Movie`
type Movie {
    genre: String!
    director: Author!
}

type Author {
    id: String!
    name: String
}
```

A union has no fields of its own, so each further dimension, such as the director, is another directive with its own selection:

schema.graphql

```
union Content
    @surrogateKey(of: "author.id genre") # → Content:author.id:{author.id}:genre:{genre}
    @surrogateKey(of: "director.id genre") # → Content:director.id:{director.id}:genre:
{genre}
=
    | Book
    | Magazine
    | Movie

# The keys below apply to a member reached through a field of type Content and
# named in a fragment of the query, or through a field returning the member itself.
type Book {
    # → Book:author.id:{author.id}:genre:{genre}
    genre: String!
    author: Author!
}

type Magazine {
    # → Magazine:author.id:{author.id}:genre:{genre}
    genre: String!
```

```
    author: Author!
}

type Movie {
    # → Movie:director.id:{director.id}:genre:{genre}
    genre: String!
    director: Author!
}

type Author {
    id: String!
    name: String
}
```

One directive per selection gives a purge by author, by director, by genre, or a combination, across every member and on the union itself:

schema.graphql

```
union Content
    @surrogateKey(of: "author.id genre") # → Content:author.id:{author.id}:genre:{genre}
    @surrogateKey(of: "author.id") # → Content:author.id:{author.id}
    @surrogateKey(of: "director.id genre") # → Content:director.id:{director.id}:genre:
{genre}
    @surrogateKey(of: "director.id") # → Content:director.id:{director.id}
    @surrogateKey(of: "genre") # → Content:genre:{genre}
=
    | Book
    | Magazine
    | Movie

# The keys below apply to a member reached through a field of type Content and
# named in a fragment of the query, or through a field returning the member itself.
type Book {
    # → Book:author.id:{author.id}:genre:{genre}
    genre: String! # → Book:genre:{genre}
    author: Author! # → Book:author.id:{author.id}
}

type Magazine {
    # → Magazine:author.id:{author.id}:genre:{genre}
    genre: String! # → Magazine:genre:{genre}
    author: Author! # → Magazine:author.id:{author.id}
}

type Movie {
    # → Movie:director.id:{director.id}:genre:{genre}
    genre: String! # → Movie:genre:{genre}
```

```
    director: Author! # → Movie:director.id:{director.id}
}

type Author {
    id: String!
    name: String
}
```

Use case	Surrogate key
Contents of a specific **author** and **genre**	`Content:author.id:{author.id}:genre:{genre}`
Contents of a specific **author**	`Content:author.id:{author.id}`
Contents of a specific **director** and **genre**	`Content:director.id:{director.id}:genre:{genre}`
Contents of a specific **director**	`Content:director.id:{director.id}`
Contents of a specific **genre**	`Content:genre:{genre}`
Books of a specific **author** and **genre**	`Book:author.id:{author.id}:genre:{genre}`
Books of a specific **author**	`Book:author.id:{author.id}`
Books of a specific **genre**	`Book:genre:{genre}`
Magazines of a specific **author** and **genre**	`Magazine:author.id:{author.id}:genre:{genre}`
Magazines of a specific **author**	`Magazine:author.id:{author.id}`
Magazines of a specific **genre**	`Magazine:genre:{genre}`
Movies of a specific **director** and **genre**	`Movie:director.id:{director.id}:genre:{genre}`
Movies of a specific **genre**	`Movie:genre:{genre}`
Movies of a specific **director**	`Movie:director.id:{director.id}`

If `of` resolves for no member at all, the deployment build reports one schema error naming the union.

Objects reached through a field

A key on a field that returns an object, an interface or a union names what that field returns, from the parent's side: `@surrogateKey(of: "id")` on `Category.featured` puts `Category:featured.id:p01` on every response that selected a category's featured product `p01`. The field name is part of the key, so keys from two fields of the same type never reach each other's responses. Such a field needs `of`, `static` or both; `static` attaches a fixed key that is included only when the field is queried.

schema.graphql

```
type Category {
    id: ID!
```

```
    name: String
    featured: Product @surrogateKey(of: "id") # → Category:featured.id:{featured.id}
}

type Product {
    id: ID!
    name: String
}
```

> ⚠ **Only when the field is selected**
>
> A key on a field that returns an object, an interface or a union is created only when that field is in the client's selection set. A key field of scalar or enum type, by contrast, keys the enclosing type whatever the client selected (see selection set expansion).

A composite key on such a field combines several subfields of the related type:

schema.graphql

```
type Library {
    books: [Book!] @surrogateKey(of: "author.id genre") # → Library:books.author.id:
{books.author.id}:books.genre:{books.genre}
}

type Book {
    genre: String!
    author: Author!
}

type Author {
    id: ID!
    name: String
}
```

On a field that returns an interface or a union, `of` is checked per member or implementor, as on a union. Where members expose different leaf fields, repeat the directive with a selection for each:

schema.graphql

```
type Library {
    media: [Medium!] @surrogateKey(of: "author.id genre") # → Library:media.author.id:
{media.author.id}:media.genre:{media.genre}
    contents: [Content!]
        @surrogateKey(of: "author.id genre") # → Library:contents.author.id:
{contents.author.id}:contents.genre:{contents.genre}
        @surrogateKey(of: "director.id genre")
    # → Library:contents.director.id:{contents.director.id}:contents.genre:{contents.genre}
}
```

```
union Content = Book | Magazine | Movie

interface Medium {
    genre: String!
    author: Author!
}

type Book implements Medium {
    genre: String!
    author: Author!
}

type Magazine implements Medium {
    genre: String!
    author: Author!
}

type Movie {
    genre: String! @surrogateKey # → Movie:genre:{genre}
    director: Author!
}

type Author {
    id: String!
    name: String
}
```

Use case	Surrogate key
Library media of a specific **author** and **genre**	`Library:media.author.id:{author.id}:media.genre:{genre}`
Library contents of a specific **author** and **genre**	`Library:contents.author.id:{author.id}:contents.genre:{genre}`
Library contents of a specific **director** and **genre**	`Library:contents.director.id:{director.id}:contents.genre:{genre}`
Movies of a specific **genre**	`Movie:genre:{genre}`

A key on a field belongs to the enclosing type only. It gives no key to the field's return types (`Book` , `Magazine` , `Movie`). A key a member declares itself, such as `Movie.genre` , applies to that member alongside the field's keys, and to no other member.

A group you purge by hand

The `static` argument defines a fixed key. Use it to give a group of responses a shared handle you can purge by hand, without introducing a new type. **Whitespace separates keys**, so you can declare several keys on one

directive or repeat the directive:

schema.graphql

```
type Product @surrogateKey(static: "catalogue") {
    # → catalogue
    name: String!
}
type Category @surrogateKey(static: "catalogue") {
    # → catalogue
    name: String!
}

# Define 3 static keys on the same type:
type Promotion
    @surrogateKey(static: "catalogue") # → catalogue
    @surrogateKey(static: "promo banner")
{
    # → promo, banner (whitespace separates keys)
    title: String!
}
```

So `static: "promo banner"` declares two separate keys, `promo` and `banner`; there is no way to put a space inside a single static key. Each key is then folded into printable ASCII. Often a union or an interface is the more natural handle.

> **The selection decides, not the response**
>
> A `static` key is included whenever the query selected the thing it is declared on: the field, the type, the interface or union, or a member named in a fragment. The response is never consulted, so an empty list or a `null` field carries the same `static` keys as a full response.
>
> That is deliberate: a `static` key is only ever purged by name, and the responses that most need that handle are the ones saying something is not there yet, such as a cached “no products”, which holds no object key for an automatic purge to catch.
>
> Where you declare it decides which queries carry it: on `type Product`, every selection touching a product; on the `products` field, that one query.

On a scalar or enum

A `static` key on a leaf type attaches to any response that selected a field returning that type, wherever the field sits.

```
scalar Money @surrogateKey(static: "ledger")

type Invoice {
```

```
  id: ID!
  total: Money!
}
```

A query for `invoice { total }` carries `ledger`; a query for `invoice { id }` does not.

> ⚠ **`static` replaces the key-field behavior**
>
> On a scalar or enum, `static` switches off the key-field behavior for that usage of the directive: `scalar Money @surrogateKey(static: "ledger")` gives `ledger` and no `Invoice:total:<value>`. To have both, apply the directive twice, which is what `repeatable` is for:
>
> ```
> scalar Points @surrogateKey @surrogateKey(static: "loyalty")
> ```

On an `OBJECT`, `INTERFACE` or `UNION`, by contrast, `static` and `of` on the same directive contribute independently. With the directive applied twice as above, a selection of `Profile.points` carries `loyalty` and `Profile:points:<value>`.

On a union or interface member

A `static` key on the union itself is added whenever the union is selected, and whenever a field returning one of its members is. A `static` key on one of its **members** is added whenever the query **names that member** in a fragment, or selects a field returning the member.

```
union Content @surrogateKey(static: "all-content") = NewsArticle | BlogArticle
type BlogArticle @surrogateKey(static: "blogroll") { title: String! }
```

A query naming both members carries `all-content` and `blogroll`, whether or not a blog article came back. A query naming only `... on NewsArticle` carries `all-content` and not `blogroll`, whatever the response holds. The same rule applies to an interface and its implementors.

How values become keys

How lists, `null` values and aliases turn into key values, and which directives the deployment build refuses, is on how keys are built.

Switching the automatic rules off

The two automatic rules can pick a field that merely shares a key name or type, and produce a key that identifies nothing: a `key` field holding rendered text, say. Such a key still takes room in the key budget and no purge will ever name it. `ignore: true` switches both rules off where the directive sits:

schema.graphql

```
# No automatic key for any field of this type
type RenderedSnippet @surrogateKey(ignore: true) {
```

```
  id: ID!
  key: String
}

# Product keeps Product:id:<id>, but not Product:key:<key>
type Product {
  id: ID!
  key: String @surrogateKey(ignore: true)
}
```

An exclusion only switches off the automatic rules; it never removes a key you wrote. Where it applies and how it meets the configuration's `ignored_types` and `ignored_fields` is on the directive reference.

Reading another directive

If your schema already marks keys with another directive, such as Apollo Federation's `@key`, GraphPilot can read that one instead:

gpilot.toml

```
[[surrogate_key.directives]]
directive_name = "key"      # the directive as it appears in your schema, without the @
of_arg         = "fields"   # its argument that lists the key fields
```

An entry replaces the built-in `@surrogateKey` rather than adding to it. Every key of an entry, including a marker directive with no arguments, is in the configuration reference. The exact definition and arguments of `@surrogateKey` are on the directive reference.

INVALIDATION · SURROGATE KEYS

The Surrogate-Key header

Your origin can name surrogate keys of its own in a `Surrogate-Key` response header. The same header does both halves of invalidation: on the response to a read it tags the cached entry, and on the response to a write it purges every entry carrying those keys. There is no API call to make, no credential to manage, and no second system that can be down when a write succeeds.

Tag on a read, purge on a write

Which job the header does depends on the request that produced the response:

The response answers	The header
A plain HTTP `GET` or `HEAD`	**Tags** the entry. Its keys are added to the ones every entry carries.

The response answers	The header
A GraphQL query	**Tags** the entry. Its keys are added to the ones GraphPilot derives from your schema.
A plain HTTP `POST`, `PUT`, `PATCH` or `DELETE`	**Purges** every cached entry carrying one of its keys.
A GraphQL mutation	**Purges** every cached entry carrying one of its keys, together with what automatic invalidation purges.

The two halves fit together: tag a read with a key your origin chooses, then name that same key on the response to the write that changes it.

For a plain HTTP API:

```
# The response to GET /products/42: tag it
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Surrogate-Key: product-42 catalog

# The response to PUT /products/42: clear it
HTTP/1.1 200 OK
Surrogate-Key: product-42
```

When the response to the `PUT` comes back through GraphPilot, every cached response tagged `product-42` is purged, and the next `GET /products/42` is a `MISS`.

For a GraphQL API it is the same, with a query and a mutation:

```
# The query response: tag it
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Surrogate-Key: catalog-v3 promo-banner

# The mutation response: clear it
HTTP/1.1 200 OK
Surrogate-Key: catalog-v3
```

Every cached response tagged `catalog-v3` is purged, whatever operation produced it and whichever fields it selected.

A purge from this header is always **hard**: the entry is removed, and the next request for it goes to your origin. It is recorded in the service's purge history like any automatic purge; see verifying a purge.

The same holds for a purge named on the response to a plain HTTP write.

Format

- **ASCII spaces, tabs and commas separate keys.** A key cannot contain any of them.
- **Keys are folded into printable ASCII**, the same way as keys from your schema: `Müller` becomes `Muller`, and a non-breaking space inside a key becomes `_` rather than splitting it. A line that is not valid UTF-8 is read as Latin-1, so both spellings of `Müller` reach the same key. At most 1024 characters per key. See what a key may contain.
- **The header name is fixed** at `Surrogate-Key` and cannot be changed. An origin that already uses that header for a CDN in front of GraphPilot sends the same keys to both.
- **Keys are not namespaced.** A key you invent shares a space with the ones GraphPilot adds (`ck:…`, and for GraphQL `Product:id:1`, `q:products`, `![Product]`). Prefix yours if you want to be sure they cannot collide.
- **Multiple headers merge.** Per HTTP, several headers with the same name form one logical header. These two forms are equivalent, and GraphPilot accepts both:

```
Surrogate-Key: product-123
Surrogate-Key: category-456
```

```
Surrogate-Key: product-123 category-456
```

GraphPilot removes your origin's `Surrogate-Key` header before the response reaches the client. To see the keys a response carries, switch on the `gp-surrogate-key` header; see debugging.

When this is the right tool

The keys GraphPilot adds on its own cover what a response *contains*. This header covers what a write *affects*.

- **Any invalidation on a plain HTTP API.** Without a schema, the only keys on an entry are its cache key and the ones your origin names. This header is how a write reaches the reads it changes.
- **A create that has to appear in a list.** The new object's key clears nothing, because no cached response contained it. A key naming the list does. See automatic invalidation.
- **A write that touches more than it returns.** Recalculating an aggregate, expiring a sibling, cascading through a relation your response does not include.
- **Keys that only exist at runtime.** A customer, a locale, a feature-flag cohort, a content version your CMS mints. Your origin knows them when it answers.
- **Keys from somewhere else entirely.** A third-party API that returns its own cache tags can have them passed straight through.

> ⚠ **A key on a write purges everything carrying it**
>
> A key on a write's response purges every entry carrying it, with no way to scope it further. A key that names something broad ("catalog") invalidates broadly. That is often what you want, but it is worth choosing the width deliberately rather than discovering it from a hit rate.

What it does not do

- **It does not reach a response that was never tagged.** Invalidation is a lookup by key, so a key that was never written on an entry matches nothing. If a purge appears to do nothing, check the entry's keys first with the `gp-surrogate-key` response header.
- **It only runs on responses that pass through GraphPilot.** A write made by a background job, an admin tool or a direct database change never produces a response here. Use the purge API or the CLI for those.
- **It does not confirm anything to your origin.** The write's response is the trigger; there is no acknowledgement of how many objects were removed. When request telemetry is stored (on every plan, within its logged-request allowance, and not in lean mode), the purge is recorded in the service's purge history with the keys it covered.

Related

- Surrogate keys: every kind of key and what a key may contain
- Automatic invalidation: what a GraphQL mutation purges on its own
- Verifying a purge: where the purge is recorded

INVALIDATION · SURROGATE KEYS

How keys are built

> **GraphQL only** For plain HTTP, name keys in the Surrogate-Key header on your origin's response.

The directive says which fields form a key. This page is what GraphPilot does with the values it finds in a response, and what happens when a directive cannot produce a key at all.

How values become keys

- **A deep key includes its full path.** `@surrogateKey(of: "category.id")` produces `Product:category.id:<id>`. Each segment must be a valid field, and the last one must not be an object, interface or union.
- **An enum contributes its value** as it appears in the response, for example `Product:stockStatus:SOLD_OUT`.
- **A list of scalars or enums contributes one key per distinct element**, not one key holding the whole list: `tags: ["new", "bestseller"]` yields `Product:tags:new` and `Product:tags:bestseller`. The order of the elements does not matter, and an element that appears twice yields its key once. In a composite key, a list multiplies out into one key per combination, as described under composite keys.

- **A list of objects contributes a key for each object**, based on the key fields.
- **Keys are collected in a set.** A key that two directives, or two objects, both produce is attached once, so defining the same key twice costs nothing at runtime.
- **Field names in a composite key are sorted alphabetically**, and casing is never changed. The only other normalization is the character folding, which leaves ordinary ASCII values untouched.
- **The key does not depend on the client's aliases.** See aliases.

When a field is `null`

A key is generated even when a field it names is `null`, with an empty value: `@surrogateKey(of: "brand.id")` on a product without a brand produces `Product:brand.id:`. That keeps caching and purging symmetric: a cached product without a brand can still be purged, an update from `null` to a value can purge both the old key (`Product:brand.id:`) and the new one (`Product:brand.id:acme`), and a purge of `Product:brand.id:` reaches every product without a brand.

A key on a field that returns an object is the exception: `@surrogateKey(of: "id")` on `Category.featured` produces no key at all when `featured` is `null`, because there is no object to key. A `static` key on that field is still attached.

A union member or implementor on which the path does not exist at all contributes no key; the empty value only appears when the field exists and is `null`. With `@surrogateKey(of: "author.id")` on a `search` field returning `Book | Movie`:

```
... on Book { author { id } }       # author.id = "A1"
... on Movie { director { name } }  # Movie has no author field
```

A book produces `Query:search.author.id:A1`; a movie produces no key from this directive. Add a second directive, such as `@surrogateKey(of: "director.id")`, to key movies as well.

Aliases

When the fragments of a union or interface field use different aliases, GraphPilot reads the response path that matches each object:

```
query {
    search {  # Returns Union type
        ... on Book {
            author { _gp_id: id }     # Uses alias "_gp_id"
        }
        ... on Magazine {
            _gp_author: author { id } # Uses alias "_gp_author" for entire field
        }
    }
}
```

For `@surrogateKey(of: "author.id")`, a book is read at `author._gp_id` and yields `Query:search.author.id:BA1`; a magazine is read at `_gp_author.id` and yields `Query:search.author.id:MA1`. The key is the same whichever alias the client used, so a purge reaches the response either way.

When a directive produces no key

A directive the deployment build refuses contributes no key, and the deployment still succeeds. Every other key in the schema still applies, including a `static` value on the same directive, so the first thing you notice without reading the diagnostics is a response that never invalidates.

The build refuses, with the type, the directive, and its line and column:

- an `of` naming a field that does not exist, or a path that does not end in a scalar, enum, or list thereof;
- an argument of the wrong type;
- a bare `@surrogateKey` on a type that needs a selection;
- a directive written where no key can come from, such as inside `extend type` or on a field argument.

Read the diagnostics of your deployment. The portal shows them under **Schema → Findings** for the live deployment and on each deployment's page under **Deployments**. `gpilot deploy` prints them when the deployment finishes, and the GraphPilot API returns them too. Only the deployment build validates your schema, so read its diagnostics.

Check them whenever you change a type that carries a key. If a field used in an `of` argument is removed or renamed, the next deployment reports it and the directive produces no key from then on. Keys already in the cache are not affected until they are invalidated or expire. To see the keys your directives produced for real data, read the `gp-surrogate-key` header.

INVALIDATION

Purging through the API and CLI

Plans Every plan **Permission** `cache:purge`

Most invalidation happens on its own, from a key your origin names on the response to a write or from what a GraphQL mutation returned. A purge you send is for everything else: a change that never went through GraphPilot, an admin tool, a support runbook, a nightly import, a migration.

You send it with `gpilot purge`, through the GraphPilot API (as a GraphQL mutation or with `POST /purge`), or with the **Purge** button on the service's **Purges** page in the portal. The CLI is the quickest for a runbook or a CI job; the API is for your own tools and for the addressing modes the CLI does not offer.

Which one to use

Addresses	CLI	API	Reach for it when
One or more surrogate keys	`gpilot purge <key> ...`	`purgeBySurrogateKey`, or `POST /purge` with `keys`	You know what changed. This is the normal one.
One cached response, by its cache key	`gpilot purge ck:<cacheKey>`	`purgeBySurrogateKey`, or `POST /purge` with `keys`	One response is wrong and it carries no key of your own.
The entry one telemetry request read or wrote	Not available	`purgeByRequest`	You are looking at a bad response in the portal's request view and want exactly that one gone.
A GraphQL query, with or without one variable set	Not available	`purgeByQueryHash`	You know the operation is wrong but not which objects.
Everything	`gpilot purge --all`	`purgeAll`, or `POST /purge` with `all`	A schema or config change made the whole cache untrustworthy.

Pick the narrowest one that reaches what you need. Everything you purge is an origin request somebody pays for on the way back in.

There is no purge by URL. To clear one plain HTTP response, purge its cache key, `ck:<METHOD>:<path>`, for example `ck:GET:/products/42`. The path carries its query string but no scheme or hostname, unless the service sets `cache.http_key.host = "include"`, in which case the key is the full URL.

From the CLI

`gpilot purge` takes surrogate keys, cache keys included, and purges them in the current service:

```
gpilot purge Product:id:p01 Category:id:audio
gpilot purge ck:GET:/products/42
gpilot purge --soft product-42
gpilot purge --all
```

It shows what it is about to purge and asks you to confirm. In a script or CI job there is no one to ask, so the purge is refused unless you pass `--yes` (`-y`):

```
gpilot purge --yes --all
```

With no keys on the command line, it reads keys from stdin, one per line. That lets you keep a list of keys in a file, or purge what another tool produced:

```
gpilot purge --yes < keys.txt
```

`--soft` cannot be combined with `--all`. Use `--service <subdomain>` to purge a service other than the current one. The CLI sends the same purge as the API, with the credential of the active profile, so the same permission and limits apply. It has no form for a request id or a query hash; use the API or the portal for those. Every flag is in the command reference.

Through the GraphPilot API

The API exposes purging as four mutations, each scoped to one service. Two of them, by key and everything, can also be sent over REST without writing GraphQL.

Calling it

The GraphPilot API is one GraphQL endpoint:

```
https://api.graphpilot.io/graphql
```

Authenticate with a bearer credential. For a script or a CI job that is an API key you created in the portal; the CLI sends the same header with whatever credential the active profile holds.

```
curl -sS https://api.graphpilot.io/graphql \
  -H "authorization: Bearer $GRAPHPILOT_API_KEY" \
  -H 'content-type: application/json' \
  -d '{"query":"mutation($i:PurgeBySurrogateKeyInput!){purgeBySurrogateKey(input:$i)
{purgedKeyCount}}","variables":{"i":{"serviceId":"shop-api","keys":["Product:id:p01"]}}}'
```

The organization comes from the API key or the login session. `serviceId` takes either the service's id or its subdomain (the `<subdomain>` in `https://<subdomain>.<label>.graphpilot.cloud`, the value `--service` takes), and both are resolved within your organization. A service belonging to another organization is reported as not found rather than as forbidden, so an id cannot be used to probe for services you cannot see.

The shared payload

All four return the same type:

```
type PurgePayload {
  success: Boolean!
  purgedKeyCount: Int!
  purgedKeys: [String!]!
}
```

`purgedKeyCount` counts **keys, not cache objects**. A key that matched nothing still counts, and no object count is available, because purges propagate between edge locations asynchronously. It can also be larger than the number of keys you sent: a GraphQL cache key sent without its `ck:` prefix is purged in both forms and counted twice, and `purgedKeys` lists both. See purging one cached response by its cache key.

It can be smaller, too. Duplicate keys are purged once: the API drops exact duplicates, and the edge then drops keys that are the same once folded, so `Product:title:Blue Shirt` and `Product:title:Blue_Shirt` in one call count as one, though `purgedKeys` lists both.

`purgedKeys` matters most for the indirect addressing modes: without it you cannot verify that a request id resolved to the entry you meant.

`success` is never `false` in a payload you receive. Every failure path returns a GraphQL error instead, so treat the presence of a payload as the success signal and read `purgedKeys` for what it covered.

purgeBySurrogateKey

```
mutation {
  purgeBySurrogateKey(input: {
    serviceId: "3fa3b8c2-9e41-4b7a-9c2e-1a6f4d8b2c10"
    keys: ["Product:id:p01", "Category:id:audio"]
    soft: true
  }) {
    success
    purgedKeyCount
    purgedKeys
  }
}
```

Field	Type	Notes
`serviceId`	`ID!`	The service to purge in: its id or its subdomain.
`keys`	`[String!]!`	Cleartext surrogate keys. At most **1000 per request**, each at most **1024 characters**.
`soft`	`Boolean`	Mark entries stale rather than removing them.

Write each key as it appears in your data. GraphPilot folds a key into printable ASCII when it tags an entry, and folds a purge the same way, so `Product:title:Blue Shirt` reaches the entry tagged `Product:title:Blue_Shirt`. The folding rules are under what a key may contain.

Purging one cached response by its cache key

A cache key counts as a surrogate key here, written with a `ck:` prefix. For plain HTTP it is `ck:<METHOD>:<path>`, the path the request was sent to with its query string, for example `ck:GET:/products/42?color=blue`. Scheme and hostname are left out, unless the service sets `cache.http_key.host = "include"`: then the key is the full URL, `ck:GET:https://catalog-api.k7m2qp.graphpilot.cloud/products/42?color=blue`, and a path on its own names no

entry. For GraphQL, `ck:<queryHash>` covers every variable set of a query, and the longer form in the portal's request detail addresses one operation with one variable set. Neither names a path or a query string, so both reach the response on every route it was cached for. The portal shows the cache key with the prefix already in place, so you can copy it as it is. See the cache key.

> ⚠ **Add the ck: prefix to a cache key from telemetry**
>
> Request telemetry read through the GraphPilot API reports a request's `cache.key` without the prefix. Put `ck:` in front of it before you purge.
>
> A **GraphQL** cache key sent without the prefix still works: its shape (a 64-character query hash, optionally followed by `#<operationName>` and `:<variablesHash>`) is unmistakable, so the API sends it in both forms, as you wrote it and with `ck:` added. A **plain HTTP** cache key gets no such help, because its shape cannot be told apart from a key of your own. Without its prefix it matches nothing, and the purge still reports success.

purgeByQueryHash

GraphQL only

```
mutation {
  purgeByQueryHash(input: {
    serviceId: "3fa3b8c2-9e41-4b7a-9c2e-1a6f4d8b2c10"
    queries: [
      { queryHash: "9f86d081884c7d65 … " }
      {
        queryHash: "b1946ac92492d234 … "
        variablesHash: "4e07408562bed … "
        operationName: "ProductPage"
      }
    ]
  }) {
    success
    purgedKeyCount
    purgedKeys
  }
}
```

Field	Type	Notes
`queries[].queryHash`	`String!`	Hash of the query document.
`queries[].variablesHash`	`String`	With it, exactly one cache object is purged. **Without it, every variable set of that query is purged.**

Field	Type	Notes
`queries[].operationName`	`String`	The `operationName` the request sent, which is the name in its cache key. With `variablesHash`, a name purges that operation's entry and `null` purges the entry of requests that sent no name. Omitted, GraphPilot looks the names up in your request telemetry (see below). Ignored without `variablesHash`.
`soft`	`Boolean`	Mark entries stale rather than removing them.

Omitting `variablesHash` is the useful case: one call removes every cached page, filter and sort order of the same query. Both hashes appear in your request telemetry, and the query hash is the same value a persisted query is registered under.

> ⚠ **Name the operation when you pass a variables hash**
>
> The operation name is part of the cache key. With `variablesHash` but without `operationName`, GraphPilot looks up in your request telemetry which cache keys that query was recorded with at that variable set, and purges the anonymous entry together with every named one it finds. The names are read from the cache keys, so a request listed under its document's name but sent without `operationName` purges the anonymous entry, which is the one it was stored under.
>
> When telemetry holds no cache key and no operation name for it, the purge is refused with `BAD_REQUEST` and nothing is purged. That is the case for requests that were never stored: a service whose `telemetry.record` left them out, a service in lean serving mode, or requests older than your plan's telemetry retention. If the lookup itself fails, the purge is refused with `INTERNAL_ERROR`. In both cases, pass `operationName` and try again: the name the request sent, or `null` for a request that sent none. A purge that states `operationName` never needs the lookup.
>
> When you state `operationName`, use the name in the request's cache key. Request telemetry shows the name the request sent or, for a request that sent no `operationName`, the name its document gives its only operation. The cache key holds only a name that was sent, so the entry of such a request is the anonymous one: purge it with `null`. The **Cache key** on the request's **Cache** tab in Insights tells the two apart: a sent name follows the query hash after a `#`.

`purgeByRequest`

```
mutation {
  purgeByRequest(input: {
    serviceId: "3fa3b8c2-9e41-4b7a-9c2e-1a6f4d8b2c10"
    requestIds: ["01J8XN3M7QK9V2ZC4T0GY5B6RD"]
  }) {
    success
    purgedKeyCount
    purgedKeys
  }
}
```

Field	Type	Notes
`serviceId`	`ID!`	The service to purge in: its id or its subdomain.
`requestIds`	`[ID!]!`	Telemetry request ids. Each resolves to the cache key that request read or wrote.
`soft`	`Boolean`	Mark entries stale rather than removing them.

This one reads your stored request telemetry (see what Insights needs). A request that was not stored gives a request id nothing to resolve to.

> ⚠ **Caution**
>
> This one is bound to **your plan's telemetry retention** (1 to 90 days), and it behaves differently depending on how much resolves:
>
> - **Some ids resolve, some do not.** The ones that do not are silently dropped, the purge proceeds with the rest, and the result reports `success`. Read `purgedKeys` to see what was actually covered.
> - **None of them resolve.** The call **fails** with `NOT_FOUND`, and the message begins with *"Nothing to purge: no telemetry row for those request ids carries a cache key"*.
>
> An id fails to resolve because retention expired, because the request was never cached, or because no telemetry was stored for it. If you need certainty, purge by surrogate key instead.

`purgeAll`

```
mutation {
  purgeAll(input: { serviceId: "3fa3b8c2-9e41-4b7a-9c2e-1a6f4d8b2c10" }) {
    success
    purgedKeyCount
    purgedKeys
  }
}
```

It takes no `soft`: a purge-all is always hard.

It addresses no surrogate key, so it returns `purgedKeyCount: 0` and an empty `purgedKeys`. That is not a failure; it is what "everything" looks like when there are no keys to enumerate.

Over REST

`POST https://api.graphpilot.io/purge` sends a purge without GraphQL. It runs `purgeBySurrogateKey` or `purgeAll` on your behalf, with the same credential, the same permission, the same limits and the same error codes, so everything above about keys, `serviceId` and the payload holds for it as well.

```
curl -sS https://api.graphpilot.io/purge \
```

```
  -H "authorization: Bearer $GRAPHPILOT_API_KEY" \
  -H 'content-type: application/json' \
  -d '{"serviceId":"shop-api","keys":["Product:id:p01","Category:id:audio"]}'
```

The body is a JSON object in one of two forms:

Body	Runs	Notes
`{"serviceId": "...", "keys": [...], "soft": true}`	`purgeBySurrogateKey`	`soft` is optional and defaults to `false`.
`{"serviceId": "...", "all": true}`	`purgeAll`	No `soft`: a purge-all is always hard.

`serviceId` is the service's id or its subdomain, as for the mutations. The body has to match one form exactly. Anything else is refused with `400` and `BAD_REQUEST` before anything is purged: no `serviceId`, `keys` that is not a list of strings, `keys` and `all` together, `soft` with `all`, `all` with any value but `true`, or a field neither form names. There is no REST form of `purgeByQueryHash` or `purgeByRequest`; send those as mutations.

A purge that ran answers `200` with the `PurgePayload` as the whole body:

```
{ "success": true, "purgedKeyCount": 2, "purgedKeys": ["Product:id:p01",
"Category:id:audio"] }
```

A refused one answers with a status that matches the code, and a body holding the same `errors` list the mutation returns, without `locations` and `path`:

```
{
  "errors": [
    {
      "message": "Access denied: Cache",
      "extensions": { "code": "FORBIDDEN" }
    }
  ]
}
```

Status	`extensions.code`
`400`	`BAD_REQUEST`: the body, or a key the purge refuses (an empty key, one over 1024 characters, more than 1000 keys, or one the edge refuses because it is over 1024 bytes once folded).
`401`	`UNAUTHENTICATED`: no `Authorization` header, an `sk_` value that is not a usable key (never issued, revoked, rotated or mistyped, not scoped to an organization, or holding only permissions an API key may not have), or a header that carries no `Bearer` credential. The response carries `WWW-Authenticate`. Any other bearer value that does not verify is refused by the edge in front of the API, also with `401`. See when a credential is refused.

Status	`extensions.code`
`403`	`FORBIDDEN` (the credential lacks `cache:purge`), `INSUFFICIENT_PERMISSIONS` (a permission check in the database refused the request), `ACCOUNT_LOCKED`, or `ACCOUNT_DELETION_IN_PROGRESS` (the account or the organization is being deleted, and accepts no changes until the deletion is complete). Also `FORBIDDEN` when the service's edge refused the purge: redeploy the service and try again.
`404`	`NOT_FOUND`: no such service in your organization, or `keys` was an empty list.
`409`	`PRECONDITION_FAILED`: the service has never been deployed or has no active hostname.
`429`	`TOO_MANY_REQUESTS`: an `sk_` key the API has not seen recently, sent over the validation budget for unrecognized keys. The key was not checked; the response carries `Retry-After`, so send the request again after that.
`503`	`SERVICE_UNAVAILABLE`: an `sk_` key could not be checked at that moment. It says nothing about the key; the response carries `Retry-After: 5`, so send the request again after that.
`500`	`INTERNAL_ERROR`, and any other code. A partial purge is possible, as for the mutations: see permissions and errors.

Read `extensions.code` rather than the status where you branch: it is the value the mutation reports too, and the codes are listed once for both. Every answer carries `Cache-Control: no-store`. Like the mutations, the endpoint has no rate limit of its own beyond the API key validation budget.

Example: a price-feed webhook

A supplier's price feed calls a small handler of yours whenever prices change. The handler writes the new prices to your database, so no request passes through GraphPilot and nothing is purged on its own. The handler then purges the products it changed:

```
curl -sS https://api.graphpilot.io/purge \
  -H "authorization: Bearer $GRAPHPILOT_API_KEY" \
  -H 'content-type: application/json' \
  -d '{"serviceId":"shop-api","keys":["Product:id:p01","Product:id:p07"],"soft":true}'
```

```
{ "success": true, "purgedKeyCount": 2, "purgedKeys": ["Product:id:p01",
"Product:id:p07"] }
```

- **Name the keys the entries carry.** `Product:id:p01` reaches every cached response that showed product `p01`, whatever query produced it. Read the spelling off the `gp-surrogate-key` header or the request detail in the portal rather than building it by hand: a key nothing was stored under still counts in `purgedKeyCount`.
- `soft` **suits a price feed.** With a stale window on the entries, readers keep getting the old price for the moment it takes to fetch the new one, and your origin sees one refresh per entry rather than a burst.

Without a stale window it purges like a hard one. See soft and hard.

- **Retry on `429`, `503` and `500`.** Purging is idempotent, so sending the same body again is safe. Read `Retry-After` where the response carries it.
- **Give the handler its own API key**, with the purge permission and nothing else. A purge sent with an API key shows no member under **Actor** on the Purges page; filter by **Trigger Manual** and the **Key** to find it.

Soft and hard

`soft: true` marks matched objects stale rather than removing them. A stale object can still be served while GraphPilot revalidates it against your origin in the background, using the serve-stale window the entry was stored with, which comes from `stale-while-revalidate` on your origin's `Cache-Control` header or from `swr` on `@cacheControl`.

An entry stored without such a window has none, so a soft purge on it behaves like a hard one. Soft purging is only meaningfully different if your origin sends `stale-while-revalidate` or your schema states an `swr`.

`soft` defaults to `false` on every mutation that accepts it, and `purgeAll` does not accept it.

Use a soft purge when a brief window of stale data is acceptable and you want to protect your origin from a stampede. Use a hard purge when the cached value is wrong rather than merely old.

With shielding

On a service with shielding active, a purge reaches the edge locations and the shield in no fixed order, so an edge location purged before the shield could fetch the shield's old copy again and keep it. To close that gap, GraphPilot sends every purge of a shielded service a second time with the same keys: a keyed purge about 2 seconds after the first, a purge of everything about 90 seconds after it. The payload you receive describes the first send; the second one runs on its own and never fails the call. Until it has run, an edge location can still answer with the old content.

A service counts as shielded when the configuration of its running deployment sets `origin.shield` and its plan includes shielding. Purges of any other service are sent once.

Limits

A call takes at most 1000 keys and each key at most 1024 characters; the mutations have no rate limit. An empty key is refused with `BAD_REQUEST` (`gpilot purge` drops blank lines from stdin before it sends anything). How the keys are counted is on limits.

Finding the values

The portal's request view is part of Insights, on every plan. Its request detail shows a request's surrogate keys and cache key on every plan; the cache key is also a column of the request list from Airliner (Business).

You need	Where it is
A surrogate key	The request detail in the portal. The `gp-surrogate-key` response header, when switched on. See debugging.
A cache key	The **Cache key** section of the request detail, prefix included. From telemetry through the API, the `cache.key` field, without the prefix: add `ck:` before you purge.
A query hash, variables hash or operation name	The request detail in the portal, and `gpilot pq list` for registered operations.
A request id	The request view in the portal, or the `gp-request-id` response header when switched on.
A service id	The service's subdomain works in its place. The id itself is in the portal.

Permissions and errors

Every purge mutation, and `POST /purge`, requires `cache:purge`, which `service:edit` does not include; see permissions. Failures come back as GraphQL errors with a code in `extensions.code`; every code is on codes. From a mutation they arrive in a `200` (from `POST /purge` with the status for the code), except a credential that is presented and refused, such as a revoked API key or one pasted without its prefix: that one is refused with `401` and `UNAUTHENTICATED` before the mutation runs. See when a credential is refused.

> ⚠ **A failed purge can be a partial purge**
>
> Keys are purged sequentially and the run stops at the first failure, so an `INTERNAL_ERROR` reporting a partial failure means an unknown prefix of your keys is already gone. Retrying is safe (purging an absent key removes nothing) but it runs against a half-invalidated cache, which is worth knowing if you were timing the purge against something else. The purge history marks such a purge as **Unclear what was invalidated**; see verifying a purge.

Related

- Verifying a purge: the purge history and how to confirm a purge worked
- The Surrogate-Key header: purging from your origin on a write
- Automatic invalidation: what a GraphQL mutation purges without a call
- Surrogate keys: what you can address
- API keys: creating a key with `cache:purge`

Security

SECURITY

Security and data protection

This page is the starting point for a security review. Each section says what GraphPilot does and links the page with the detail.

For personal data in particular, what is recorded, what your origin receives, how long it is kept and every setting that reduces it, see privacy and personal data.

Ways to control this

Through	What it controls
`gpilot.toml`	Who may call (authentication, client profiles), what a request may carry (request limits), which operations run (persisted queries), and what is recorded about a request (`[telemetry]`).
Your origin	What may be cached and for whom (authenticated requests), and whether a request really came from GraphPilot (verifying the signature).
The portal	Your password and multi-factor authentication, sessions, members and roles, API keys, and rotating a service's signing key.

Where your data lives

- **Cached responses** are held in the edge cache at the locations that served them, worldwide, with no region pinning. Data that must not leave a region should not be cacheable. See where data lives.
- **Request telemetry** is kept in the EU for at most 90 days. How far back Insights reaches depends on your plan, from 1 day on Glider (Free) to 90 days on Jumbo (Enterprise). See telemetry and logs and what Insights needs.

What is recorded about a request

- `authorization` **and** `cookie` **are not stored as sent.** By default they are replaced by a keyed digest, and only a handful of descriptive headers are kept. You choose which headers are recorded with `[telemetry.headers]` .
- **The client's IP address** can be kept in full, truncated to its network part, or not stored at all, per service: see client IP address.
- **Which requests are recorded** at all can be narrowed with `telemetry.record` .

Every category of recorded data and the setting that controls it is listed on privacy and personal data.

Protecting your origin

- **Request signing:** every request GraphPilot forwards is signed with a per-service key, so your origin can refuse anything that did not come through it.
- **Origin allowlist:** the IP ranges GraphPilot reaches your origin from, for a firewall rule.
- **Authentication:** credentials are verified at the edge on every request, a cache hit included.
- **Hardening:** request size, query depth, document size, selection and introspection limits, and what the edge does not check.
- **Persisted queries:** run only the operations your clients registered.

Accounts and access

- **Sign-in:** password and multi-factor authentication, and a list of active sessions you can revoke.
- **Access within an organization:** four roles and API keys, both made of the same permissions.
- **Deletion:** delete your account or delete an organization, and what each removes.
- **Not available yet:** single sign-on (SSO/SAML) and an audit log; see current limitations.

Service status

Incidents and maintenance on GraphPilot's side are posted on the status page.

Reporting a vulnerability

Report a suspected vulnerability privately to security@graphpilot.io, not in a public issue or forum. Include the steps to reproduce it, the service hostname or page it affects, and what you observed.

SECURITY

Privacy and personal data

This page is the overview for a data protection review: what GraphPilot records about a request, where and for how long, what your origin receives, and each setting that reduces it. Every section links the page with the detail.

Everything configurable here is per service, in `gpilot.toml`, and applies from the next deployment. None of it changes what is billed.

What is recorded

Every request a service serves is recorded in Insights, unless `telemetry.record` leaves it out. A recorded request keeps:

Category	What is kept	Can be reduced with
Client address	The IP address as the edge saw it, plus the location and network data looked up from it: country, region, city, latitude, longitude, postal code, ISP and ASN.	`telemetry.client_ip`
Client	The User-Agent and the device type derived from it, and the client profile the request matched.	`telemetry.record`, to leave such requests out
Request	Method, hostname, path, size, and the correlation ID the client sent.	Not configurable
Headers	Only the headers the `[telemetry.headers]` lists name. By default `authorization` and `cookie` are stored as a keyed digest and only a handful of descriptive headers in clear.	`[telemetry.headers]`
GraphQL	The operation name and type, the query document, a hash of the document, and the GraphQL errors your origin answered with (message, path and code). Variable values are not stored: the request keeps a keyed hash of them.	Not configurable; see the note below the table
Cache	The cache status and key, the surrogate keys of the response, and the segment values that partitioned it: a vary rule's value as a keyed pseudonym by default, the `private` and `public` values always as a keyed digest.	`telemetry.vary_values`, `pseudonymize`
Response	Status, size, latency and the recorded response headers. The response body is not stored.	`[telemetry.headers]`

The warnings and errors the edge writes for a request are kept as its log lines. A log line carries the request's id and the correlation ID the client sent, but no client address, location, User-Agent or client profile; those are kept with the request only. The full list of what each tab of a request shows is in the request detail panel.

> **Personal data in the document**
>
> The query document and the errors your origin returns are stored as they are. A value written into the document itself, such as an email address as a literal argument, is stored with it; the same value passed as a variable is not, because only the keyed hash of the variables is kept. An error message is stored as your origin wrote it.

Where it is stored: request telemetry and log lines are kept in the EU, for at most 90 days. Cached responses are a separate question: they are held in the edge cache wherever they were requested, with no region pinning. See where data lives.

Controlling telemetry

The `[telemetry]` section decides what is stored, without changing how a request is served or cached. The full reference is `[telemetry]`.

Key	Default	Does
`record`	unset (record everything)	Which requests are recorded at all. `"false"` records nothing. A request left out is still served, billed and counted.
`client_ip`	`"full"`	`"truncated"` keeps the network part of the address and drops latitude, longitude and postal code; `"none"` stores no address and no region or city.
`vary_values`	`"pseudonymized"`	`"clear"` stores every vary rule's value as derived, instead of a keyed pseudonym.
`[[telemetry.headers.*]]`	narrow lists	Which headers are recorded, and which are stored as a keyed digest (`sanitize`) or left out (`exclude`).

gpilot.toml

```
[telemetry]
# Leave the cluster's health probes out of Insights.
record      = "!starts_with(header:user-agent, 'kube-probe/')"
# 203.0.113.77 is stored as 203.0.113.0.
client_ip   = "truncated"
# The default, stated so a review can see it.
vary_values = "pseudonymized"
```

`vary_values = "clear"` is an explicit choice for the whole service: a rule whose value is a session cookie, a token or an email address then stores that value in clear, and the header sanitize rules do not cover it. See vary values.

A header list you write replaces the default one entirely, its sanitize rules included, so put `authorization` and `cookie` first. See `[telemetry.headers]` .

Pseudonymizing values

`pseudonymize(s)` replaces a value with a keyed pseudonym before the cache, your origin or your telemetry see it. It is accepted in `cache.policy.public` , `cache.policy.private` and a vary rule's `value` . Equal inputs give equal pseudonyms within the service, so the partition still groups the same callers:

gpilot.toml

```
# The organization stays readable, the user id never leaves the edge.
[[cache.policy.vary]]
name  = "x-user"
value = "auth:idp.org_id + ':' + pseudonymize(auth:idp.sub)"
```

Under the default `vary_values = "pseudonymized"` , Insights shows a pseudonym either way; `pseudonymize` then decides what the cache partitions on and what a `plain` origin receives. Under `vary_values = "clear"` , it decides which part of a value is readable. See keeping identifiers out of Insights with `pseudonymize` .

What the origin receives

Your origin receives the request as the client sent it, with these exceptions. See request headers your origin receives.

What	By default	Change it with
The client's IP address	Not sent.	`origin.forward_client_ip = true` sends it in `gp-client-ip`.
Forwarding headers a client sent (`X-Forwarded-*`, `Forwarded`, `X-Real-IP` and others)	Removed.	`origin.forwarded_headers = "pass"`
A vary rule's derived value	Not sent.	`pass_to_origin`: `"anonymized"` sends a keyed `v1=` digest, `"plain"` the value itself.
Any other header or cookie, `Authorization` included	Sent unchanged.	`[origin.headers]` and `[origin.cookies]` `hide` or `only` lists.

gpilot.toml

```
# The origin learns a stable, opaque id per shopper, and never sees the token.
[[cache.policy.vary]]
name           = "x-shopper"
value          = "pseudonymize(auth:idp.sub)"
pass_to_origin = "plain"

[origin]
headers = { hide = ["authorization"] }
```

A filter in `[origin.headers]` applies just before the request is forwarded, after the credential was verified and the cache key was built. With the credential hidden, have your origin check the request signature to know a request came through GraphPilot.

Combining the options

`pseudonymize`, a vary rule's `pass_to_origin` and `telemetry.vary_values` act one after the other, each on the result of the one before:

1. **The rule's value is derived.** `pseudonymize` replaces its argument right there, so the derived value already holds the pseudonym. The cache key is built from this derived value.
2. **The origin receives it** as `pass_to_origin` says: `"plain"` sends the derived value, `"anonymized"` a keyed `v1=` digest of it, `"never"` nothing.
3. **Insights records it.** A rule at `"anonymized"` records exactly the `v1=` value the origin received, whatever `vary_values` says. A rule at `"plain"` or `"never"` follows `vary_values`: `"pseudonymized"` (default) records `hmac-sha256:` and 16 hex digits of the derived value, `"clear"` the derived value itself. `private` and `public` values are always recorded as a keyed digest.

For a vary rule with `value = "pseudonymize(auth:idp.sub)"`:

`pass_to_origin`	`vary_values`	The origin receives	Insights records
`plain`	`pseudonymized`	The pseudonym	A digest of the pseudonym
`plain`	`clear`	The pseudonym	The pseudonym
`anonymized`	either	A `v1=` digest of the pseudonym	The same `v1=` digest
`never`	`pseudonymized`	Nothing	A digest of the pseudonym
`never`	`clear`	Nothing	The pseudonym

Hashing a pseudonym a second time is harmless: both steps are deterministic and use the service's key, so segments group and the cache hits exactly as with one step, and no clear value appears anywhere. It also adds nothing. `pseudonymize` earns its place in two cases:

- **With** `pass_to_origin = "plain"`, when the origin needs a stable value per user without learning who the user is.
- **With** `vary_values = "clear"`, to keep one part of a value readable, such as the organization, and hide the rest.

With `"anonymized"`, or with the default `vary_values` and a rule at `"never"`, it is redundant.

Retention and deletion

What	How long
Request telemetry in Insights	1 day on Glider (Free), 7 days on Jet, 30 days on Airliner, 90 days on Jumbo. The GraphQL errors of a request go with it; error groups and filter suggestions are deleted once they have not occurred within the plan's retention. See what Insights needs.
The raw telemetry copy	At most 90 days per file, in the EU, used only to rebuild Insights. See the raw telemetry copy.

To delete earlier:

- **Reset history.** An Admin can delete a service's request telemetry, purge history and alert history at any time in the portal, under **Settings** → **General**, without deleting the service. The request counts billing is based on and every invoice are kept. See reset history.
- **Delete a service, an organization or your account.** Its telemetry, logs and stored files stop being readable at once and are deleted within minutes, once each service is off the edge. See delete the service, delete an organization and delete your account.

A deletion or a reset keeps some records on purpose: the billing and invoice history, the record of the accepted terms, the raw telemetry copy and backups until they expire. The full list is what a deletion keeps.

A data-minimal setup

- [ ] `telemetry.client_ip = "truncated"` or `"none"`.
- [ ] `telemetry.vary_values` left at `"pseudonymized"`.
- [ ] `telemetry.record` leaves out traffic you do not need, such as health probes.
- [ ] Every replaced header list starts with the sanitize rules for credentials.
- [ ] Personal parts of a partition value wrapped in `pseudonymize`.
- [ ] Vary rules at `pass_to_origin = "never"` or `"anonymized"`, unless the origin needs the value.
- [ ] `origin.forward_client_ip` left off, unless the origin needs the address.
- [ ] Personal values sent as GraphQL variables, not written into the document.

The worked example, with what Insights then shows, is keeping request telemetry minimal.

What a service records and passes on follows from its configuration, and the responsibility for that configuration, such as choosing `vary_values = "clear"`, lies with you. How GraphPilot processes personal data on your behalf is set out in the data processing agreement.

Related

- Security and data protection
- Configuration reference: `[telemetry]`
- What Insights records
- Where data lives

SECURITY

Authentication at the edge

GraphPilot verifies the JWTs your clients send, and that buys you two things.

A private response can be cached without being shared. Without a verified claim there is nothing to separate one caller's copy from another's, and rather than risk serving the wrong one, GraphPilot does not cache the response at all. How narrowly it is separated is your private cache policy to write: a claim like `sub` gives one entry per person, while a claim naming an organization or one of your customers gives one entry that everyone in it shares. Verifying the token is what makes either possible; it is not what decides which you get.

A broken credential is stopped at the edge. A token with a bad signature, an expired one, or one failing a constraint you configured is answered with `401` from the nearest location and never reaches your origin. Your origin does not spend a request rejecting it, and does not have to be reachable for it to be rejected. This is the default and it is configurable per reason.

Be precise about what the second one is worth, because it is narrower than "the edge protects your API": only credentials that fail to *verify* are stopped. A valid token belonging to somebody who may not see the data still

reaches your origin, because whether they may is your origin's decision and not something a signature answers. See What this does not do.

Both are visible to your end users, so it is worth knowing exactly what happens.

Configuring a provider

```
[auth.providers.idp]
type = "jwt"
allowed_algorithms = ["RS256"]
jwks_url = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers = ["https://idp.example.com/"]

[cache.policy]
private = 'auth:idp.sub'
```

The block name is what the policy refers to. `auth:idp.sub` reads the `sub` claim of a token verified by the `idp` provider; nested claims work too (`auth:idp.org.id`). Every `.` steps into a nested claim. A step may contain letters, digits, `_` and `-`; any other claim name is written in quotes and is then one step, dots and slashes included, so a claim whose name is a URL is read as `auth:idp.'https://acme.example/org_id'`. Quoted steps escape like string literals (`\'` for a quote). In `gpilot.toml`, such a path goes in a double-quoted string, which holds the single quotes as they are: `private = "auth:idp.'https://acme.example/org_id'"`.

`allowed_algorithms` is required, and it is a security control. The algorithm a token names in its own header only selects the verifier after this list has accepted it. Accepting whatever the token claims would allow a downgrade: an attacker presents an HS256 token to an RS256 provider and signs it with the public key as the shared secret.

`allowed_issuers` is required for a provider that verifies with a public key: one that sets `jwks_url` or `public_key`, or lists an asymmetric algorithm (`RS*`, `PS*`, `ES*`, `ES256K`, `EdDSA`). Without it, or with an empty list, the configuration is refused when it is read and the deployment fails:

```
a JWT provider that verifies with `jwks_url`, `public_key` or an asymmetric algorithm
must set
`allowed_issuers`: public keys can be shared across an issuer platform's tenants, and
without an
issuer check a token minted for another tenant verifies here
```

A provider that only uses a shared secret (`HS*`, `BLAKE2B`) may leave it unset.

The key to verify with comes from `jwks_url`, a PEM `public_key`, or a shared `secret`, and `default_key_id` names the key to use for a token that carries no `kid`. Each is in the configuration reference.

> ⚠ **A shared secret is stored as written**
>
> A `secret` is part of the configuration document: it is stored and deployed to the edge as written, and anyone with `service:edit`, a member or an API key, can read it back with `gpilot config get`. Keep a shared HMAC secret out of version control, or use an asymmetric provider (`jwks_url` or `public_key`), which holds only public material.

Optional constraints, all unset by default: `allowed_issuers` (for a shared-secret provider), `allowed_audiences`, `required_nonce`, `required_key_id`, `time_tolerance`, `max_validity`, `max_token_length`, `max_header_length`, `accept_future`, `allow_missing_exp`.

Set `allowed_issuers` and `allowed_audiences`. A signature only proves which key signed a token. Some identity platforms sign every tenant's tokens with the same keys and tell tenants apart only by `iss` and `aud` (Firebase Auth is the common case). A provider that checks neither accepts a token minted in someone else's project as a verified identity, with its own cache partition and whatever `auth:` conditions its claims satisfy. A public-key provider without `allowed_issuers` does not deploy at all, as above. For the rest, every deployment warns about a JWT provider that sets no `allowed_audiences`, and about a shared-secret provider that sets no `allowed_issuers` (`source` `auth_provider`, see diagnostics); the deployment still goes ahead.

```
[auth.providers.idp]
type = "jwt"
allowed_algorithms = ["RS256"]
jwks_url = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers = ["https://idp.example.com/"]
allowed_audiences = ["my-api"]
```

A token must carry `exp`. A token that verifies but states no expiry is rejected with `AUTH_INVALID_TOKEN`, because nothing else would ever stop accepting it. If your issuer genuinely mints tokens without `exp`, set `allow_missing_exp = true` on the provider, and set `max_validity` with it, so that a token's `iat` bounds how long it is accepted. A token that does state an expiry is held to it either way.

Names come from the JWT registry, which is what your identity provider documents: `ES256K` with a capital K, and `EdDSA` rather than `Ed25519`. These fourteen are accepted, spelled exactly as shown:

Family	Names
HMAC (shared secret)	`HS256`, `HS384`, `HS512`
BLAKE2b (shared secret, not in the JWT registry)	`BLAKE2B`
RSA	`RS256`, `RS384`, `RS512`
RSA-PSS	`PS256`, `PS384`, `PS512`
Elliptic curve	`ES256`, `ES384`, `ES256K`
Edwards curve (Ed25519)	`EdDSA`

An unknown name is refused when the configuration is read, not when a request arrives.

By default the token is read from `Authorization`, with a `Bearer` prefix stripped if present. The scheme is matched case-insensitively, so `bearer` and `BEARER` work too. Set `header` to read from a different one, where a bare token without any scheme is also accepted.

Example: Auth0 with one entry per organization

A B2B shop signs its customers in with Auth0 Organizations, and every member of one customer sees the same negotiated price list. Auth0 adds an `org_id` claim to the access token of a user who signed in through an organization, and that claim is what the price list is keyed on:

gpilot.toml

```toml
[auth.providers.auth0]
type               = "jwt"
allowed_algorithms = ["RS256"]
jwks_url           = "https://shop-example.eu.auth0.com/.well-known/jwks.json"
allowed_issuers    = ["https://shop-example.eu.auth0.com/"]
allowed_audiences  = ["https://shop-api.example.com"]

[cache.policy]
private = 'auth:auth0.sub'

[[cache.policy.vary]]
name       = "x-gp-organization"
value      = 'auth:auth0.org_id'
scope_only = true
```

schema.graphql

```graphql
enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION @vary(names: ["x-gp-organization"])
}

type PriceList @cacheControl(maxAge: 300, scope: ORGANIZATION) {
    currency: String!
    entries: [Price!]!
}
```

- **Copy the issuer from a real token.** Auth0's `iss` is your tenant's domain with a trailing slash, and `allowed_issuers` has to match it exactly. `allowed_audiences` is the identifier of the API you registered in Auth0. Auth0 can issue `aud` as an array; a token is accepted when any of its audiences is on the list, and a token without `aud` is refused.

- **A user without an organization is not stored.** A token without `org_id` derives no `x-gp-organization`, so a price list answered to it is not cached rather than shared (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`).
- **Key on everything the price list depends on.** If members of one customer see different prices by role, the role has to be in the key too, as a string claim: an array such as Auth0's `permissions` resolves to nothing, and a namespaced claim is read by quoting its name (`auth:idp.'https://shop-api.example.com/role'`). See is your field really organization-uniform?

Products and categories, scoped `PUBLIC`, stay one entry for every customer, because the rule is marked `scope_only`. The recipe caching per Auth0 organization adds the role to the key and walks through the requests to send and what Insights shows.

Opaque API keys

Not every credential is a JWT. A provider of type `api_key` recognizes an opaque key by its prefix:

```
[auth.providers.orgkey]
type     = "api_key"
prefixes = ["sk_"]
```

This provider verifies nothing, and cannot. An API key is a secret its issuer holds, with no signature and no key material to check against. Your origin remains the authority on whether a key is valid, exactly as before.

What it does buy you is two things:

- **A JWT and an API key can share one header.** A value matching none of the prefixes is treated as *not this provider's credential* rather than as a broken one, so the JWT provider reading the same header still gets its turn. Without this, every API key on `Authorization` would be refused as a malformed token.
- **API key traffic stays cacheable.** The provider contributes one claim, `key`, holding a SHA-256 digest of the key. `private = "auth:orgkey.key"` therefore resolves, and responses are partitioned per key instead of falling out of the cache entirely. The digest is what travels, never the key itself.

`prefixes` is required and may not be empty or contain an empty string. Such a list would claim every credential on the header, every broken token would read as somebody else's credential, and the service would silently stop refusing anything at the edge. The configuration is rejected rather than accepted with that effect.

`header` defaults to `Authorization`, with a `Bearer` prefix stripped if present.

> **Note**
>
> These are keys **your** clients send to **your** API. They are unrelated to the GraphPilot API keys you create in the portal, which authenticate you to GraphPilot itself.

Migrating between issuers

Configure both, reading the same header:

```
[auth.providers.old]
type = "jwt"
allowed_algorithms = ["RS256"]
jwks_url = "https://old-idp.example.com/.well-known/jwks.json"
allowed_issuers = ["https://old-idp.example.com/"]

[auth.providers.new]
type = "jwt"
allowed_algorithms = ["RS256"]
jwks_url = "https://new-idp.example.com/.well-known/jwks.json"
allowed_issuers = ["https://new-idp.example.com/"]
```

A token minted by either issuer is accepted, and the provider it does not belong to stays quiet. A credential is only refused when **no** provider reading that header could verify it, so you do not need your clients to cut over at the same moment as your configuration.

Point your cache policy at whichever provider carries the claim you partition by. During the overlap you will want both, which a policy expresses with `||`:

```
[cache.policy]
private = 'auth:new.sub || auth:old.sub'
```

What happens to a request

The request	GraphPilot's response
No credential	Forwarded unchanged. Nothing is rejected. A private response is not cached, since there is nothing to partition it by.
A token that verifies	Forwarded, and its claims become available to your cache policy. Private responses are cached, partitioned by whatever that policy resolves to.
A token that does not verify (bad signature, expired, failing a constraint)	Rejected with **401**. It never reaches your origin.
A token naming a key GraphPilot does not have	Rejected with **401**, with a message that points at the key state rather than at the token.

A request without a credential is never rejected. Adding an auth provider to a service does not close it off: public queries keep working exactly as before.

The last two rows share a status code and differ in message on purpose. A client told its token is invalid will try to authenticate again; when the real cause is that GraphPilot does not yet have the key to check it with, that retry cannot succeed, so the response says so instead.

Choosing what a rejection does

Rejecting is the default, and you can turn it off per reason:

```
[auth.rejection]
invalid_token = "reject"              # the default for all three
unknown_key_id = "continue"
key_material_unavailable = "continue"
```

`continue` forwards the request as if it carried no credential, leaving the decision to your origin. It does not weaken the cache: without a verified claim there is nothing to partition by, so a private response is simply not cached, exactly as for an anonymous request.

The setting is per reason because the reasons are yours to weigh differently. A broken token is your client's problem and stopping it at the edge saves your origin the work. Stale key material is GraphPilot's, and you may prefer your users reach your origin while it catches up rather than see a 401 they can do nothing about.

Key rotation

Your issuer's key set (JWKS) is fetched from `jwks_url` when you deploy and delivered to the edge with the service, so on the normal path your identity provider is not in the request path: if it is slow or down, requests carrying a known key are still verified.

A deployment whose key set is unusable stops before it goes live, because the new version would refuse every token for that provider with `401`. The previous deployment keeps serving. The message starts with one of these codes and names every address that was tried:

Code	When
`JWKS_URL_REFUSED`	The `jwks_url` is not a URL, is not `https://`, has no host, or carries credentials. Plain `http://` is accepted only on `localhost`, `127.0.0.1` and `[::1]` (a key set over plain http could be swapped by anyone on the path), and refused with `a key set must be served over https` for every other host. Not retried.
`JWKS_FETCH_FAILED`	No address answered with a `2xx`. The deployment retries it, since the issuer may only be down for a moment.
`JWKS_NO_USABLE_KEY`	An answer came back, but it is not JSON, has no `keys` array, is empty, or holds no key that can verify one of the provider's `allowed_algorithms` (a key marked for encryption does not count). Not retried.

A provider that allows only HMAC algorithms never verifies with this set, so for it any non-empty set is enough. The `jwks_url` has to be reachable from the public internet: a connection to an address in a private network is not made, which ends in `JWKS_FETCH_FAILED`. A `jwks_url` on one of the service's own hostnames is fetched from your origin first, with the same path and query, because during a deployment that hostname still serves the previous version, or nothing yet; the `jwks_url` itself is the fallback. That request to your origin is not signed, so

an origin that refuses unsigned requests has to serve the key set without a signature, or `jwks_url` has to point at your identity provider directly.

Each fetch is bounded: 10 seconds in total, 5 seconds without data, at most 3 redirects and a body of at most 512 KiB. An address past any of these counts as not answering. The codes are also listed under deployment failures.

A token naming a key id (`kid`) that the delivered set does not hold is what a rotation between deployments looks like. When that happens, GraphPilot fetches `jwks_url` again, so a rotation is usually picked up by the first request that carries the new key, without a redeploy.

An edge location fetches a provider's `jwks_url` at most once a minute, however many requests name an unknown key, and only over `https` (plain `http` only on `localhost`, `127.0.0.1` and `[::1]`). For any other plain `http` URL the refresh is skipped and logged, and the keys delivered at deploy time keep verifying. Until a fetch finds the key, a token naming it is rejected with `AUTH_UNKNOWN_KEY_ID` (or forwarded, if you set `unknown_key_id = "continue"` above). So after a rotation, a token with the new key can be rejected for up to a minute if the edge location fetched shortly before the issuer published it.

So plan a rotation the way a JWKS endpoint is designed for: publish the new key before you sign with it. Both keys are served, GraphPilot accepts both, and no request depends on a fetch.

WebSocket subscriptions

A provider verifies credentials that arrive as request headers, which on a WebSocket means the handshake and nothing else. A token sent in the `connection_init` payload, as browser clients of `graphql-transport-ws` do, is never seen at the edge, so your origin has to verify it. See subscriptions and WebSockets.

What this does not do

GraphPilot is not your authorization layer. It checks that a token is authentic and then uses a claim as a cache dimension. Whether that user may see the data is your origin's decision, and every request that carries a valid token still reaches it (unless it is served from that user's own cache entry).

SECURITY

Hardening a public API

Putting your origin behind GraphPilot moves some of its exposure to the edge. This page lists what GraphPilot refuses on your behalf, and what it does not. The default value of every limit named here is on size limits and request limits; this page says what each control is for.

Which controls run on a cache hit

A cache hit is answered without parsing the GraphQL document, so the controls split in two. Those that read the raw request or the caller run on every request, hits included. Those that read the document run only when a

request reaches GraphPilot's GraphQL handling, which is on a cache miss or a pass.

Control	Runs on
Request size (`max_body_bytes`)	Every request
GraphQL on other paths	Every request
Credentials (auth providers)	Every request
Client profiles, including a profile's own body limit and its hashes-only rule	Every request
Persisted queries switched off (`PERSISTED_QUERY_NOT_SUPPORTED` for a request carrying a hash)	Every request
Introspection	Misses and passes
Query depth	Misses and passes
Document size and selections	Misses and passes
Operation allowlist (`only_registered`)	Misses and passes

A hit can only serve a response your origin already produced for a request that passed these checks at the time, so skipping them on a hit costs nothing while the configuration stays the same. After you tighten a document control, responses cached under the old rules keep being served until they expire or are purged.

Introspection

> **GraphQL only**

Blocked by default. An operation selecting `__schema` or `__type` is refused with `INTROSPECTION_DISABLED` and never reaches your origin; `request.introspection = "allowed"` lifts the block. `__typename` is never blocked; it carries no schema shape. Introspection sent to a path that is not a GraphQL path is not a way around the block: it is refused with `404` and `NOT_A_GRAPHQL_PATH`.

Your own schema stays available to you in the portal, under **Service → Schema** (**schema.graphql** holds the deployed file), which reads it from GraphPilot rather than through your service hostname. Blocking introspection costs your team nothing.

> **Note**
> Blocking introspection is obscurity, not security. It raises the cost of discovering your schema; it does not stop anyone who already has your client bundle. Treat it as one layer, and put the allowlist below underneath it if the API is genuinely public.

Platform probes

GraphPilot itself sends two kinds of request through a service's own hostname: the GraphQL detection of a new service's setup (`{ __typename }` on `/graphql` , `/api/graphql` , `/query` and `/`), and the introspection query that

reads its schema, during the setup and for every deployment of a service whose schema source is **Introspect at deploy**.

A platform probe is one of two fixed documents GraphPilot sends, the detection query `{ __typename }` or the standard introspection query, as a `POST` with a JSON body to a path with no query string, signed by GraphPilot for that one service with a timestamp no more than five minutes off. Any other request carrying GraphPilot's signature is refused with `403`. The edge recognizes a probe by that signature and document and then:

- **lets it through** whatever the deployment says about introspection, the GraphQL paths, persisted queries, client profiles and authentication, so introspection blocked here does not block GraphPilot reading your schema;
- **still applies** the service's depth and document size limits and the request size limit, so a very tight selection limit can refuse the introspection query;
- **never caches it**: every probe reaches your origin;
- **forwards it like any request**: from the edge's IP ranges, signed with the service's signing key, with the configured origin timeouts;
- **records it apart from your traffic**: a probe is not a request unit, not in Insights or the requests log, and does not tick the onboarding "first request".

Your origin decides whether to answer introspection. GraphPilot holds no credentials for it, so an origin that disables introspection in production or requires an admin secret for it answers the probe with an error, and the setup or the deployment says so.

A probe sent with a wrong or stale signature is refused with `401`, and a signed request that is not one of the two documents (another method, a query string, any change to the document) with `403`; neither reaches your origin. The recognition needs a runtime with this feature: on an older runtime, a probe to a service without a schema is forwarded as an ordinary request (and counted as one), and a probe to a service with a schema is treated like any other introspection.

Query depth

> **GraphQL only**

An operation nested deeper than `request.max_query_depth` allows is refused with `400` and `QUERY_DEPTH_LIMIT_EXCEEDED`, before your origin sees it. The message names the measured depth and the limit, for example "The operation nests 12 levels deep; this service allows 10." The default of `10` sits past what ordinary product queries reach and short of what a nesting attack needs: a schema with a cycle in it (`user { posts { author { posts … } } }`) multiplies your origin's work with every level. A subscription sent as an HTTP request is measured like a query or a mutation; one sent over a WebSocket is not measured at all (see what the edge does not check).

gpilot.toml

```
[request.max_query_depth]
client_document = 10      # documents the client sent
```

```
registered_document = 10  # documents resolved from your persisted query registry
```

Both default to `10`, and `0` switches either one off. GraphPilot's own ceiling of 128 levels still applies with both at `0`, because past it the document cannot be parsed safely; see query depth. The names say where the document came from, not which protocol carried it: an APQ handshake, where the client sends the document alongside its hash, is measured as a client document. A client profile can set a lower ceiling of its own; the stricter of the two decides.

How depth is counted

- **Selection sets are counted, starting at the operation's own.** `{ a }` is 1, `{ a { b } }` is 2. The depth of an operation is its deepest single path; sibling fields do not add up, and an alias is a branch of its own.
- **Fragments are flattened before measuring**, so nesting hidden behind fragment spreads measures the same as nesting written out.
- **A type condition is not a level.** `... on Post` only says which type the fields beside it belong to.
- **A field your deployed schema does not define is not counted.** The edge does not refuse it either: unless the edge rewrites the document, your origin receives the text the client sent, that field included, and its own validation decides. An operation made only of such fields is refused with `400` and `INVALID_GRAPHQL_REQUEST`.

Limitations

- **A query GraphPilot cannot parse is refused, not measured.** A request with a syntax error, one naming an operation its document does not define, or one whose document holds several operations and names none is refused with `400` and `INVALID_GRAPHQL_REQUEST` rather than forwarded, because your origin might parse it differently and run what GraphPilot never checked. Valid GraphQL that GraphPilot's parser does not accept yet (braced unicode escapes such as `\u{1F600}` in a string) is refused the same way.
- **A cache hit is not measured again.** The check runs only on a miss or a pass, so a response stored while the limit was higher stays servable until it expires or is purged. See which controls run on a cache hit.
- **Exempting registered documents is only safe with `preset = "trusted"`.** Under `apq`, any client may register any document and then send its hash, so `registered_document = 0` there is an exemption for anyone willing to make two requests.
- **Introspection is not measured.** Nothing under `__schema` is counted, so the standard introspection query measures 1 and GraphiQL, schema downloads and code generation work on the default. For the same reason a deeply nested introspection query is not caught by this limit; only GraphPilot's own 128-level ceiling applies to it. When introspection is blocked, which is the default, that query is refused anyway.

See query depth for the limit beside the others.

Request size

A body larger than `request.max_body_bytes` is refused with `413` and `PAYLOAD_TOO_LARGE`, before it is parsed and before it reaches your origin. The check reads the bytes actually received, so an untruthful `Content-Length` buys

nothing. Tighten it to the largest request your clients actually send; the default is on request body.

> ⚠ **Caution**
> The body limit does not bound execution cost. A document with thousands of aliased selections is small and still expensive to plan. That is what the document limits below are for.

Document size and selections

GraphQL only

A document longer than `request.max_document_bytes` is refused with `400` and `DOCUMENT_SIZE_LIMIT_EXCEEDED` before it is parsed, and an operation making more selections than `request.max_selections` is refused with `400` and `DOCUMENT_SELECTION_LIMIT_EXCEEDED` before it is planned. Both apply to registered documents as much as to documents a client sends, and `persisted_query.max_document_bytes` and `persisted_query.max_selections` can give registered documents ceilings of their own (see persisted queries).

Selections are counted with every alias and every fragment spread expanded, so alias spam and fragment multiplication are both caught. The defaults are on query document, and the counting rules on how selections are counted.

An allowlist of operations

GraphQL only

The strongest control available today. With persisted queries set to `preset = "trusted"` (or `only_registered = true`), the only operations your service runs are the ones you registered:

- free-form query text is refused outright,
- an unregistered hash is refused,
- and an unreachable registry **fails closed** rather than falling open.

This turns “what can be executed against my API” from an open question into a list you publish from your build. It is the answer to depth attacks, alias amplification and introspection at once, because none of them can be expressed by an operation you did not ship.

The trade is that clients must be built against a manifest. See choosing a setup. Like the other document controls, the allowlist is checked on a cache miss: a response cached before you switched stays servable until it expires or you revoke the operation, which purges it.

GraphQL on other paths

GraphPilot runs its GraphQL checks only on the paths in `request.graphql_paths`. Your origin may answer GraphQL on other paths as well (Apollo Server’s `startStandaloneServer` answers on every path), so a request on any other path that carries a GraphQL operation is refused with `404` and `NOT_A_GRAPHQL_PATH` rather than

forwarded past those checks. The message names the GraphQL paths, for example `This service serves GraphQL at /graphql. Send the operation there.`

On another path, a request carries an operation when it has one of these:

- a JSON body (`application/json` or any `+json` type) whose top-level `query` is a GraphQL document that defines an operation, or that carries `extensions.persistedQuery`. A JSON array is checked object by object, and a body that repeats `query` is checked on every copy;
- an `application/graphql` body that is such a document;
- an `application/x-www-form-urlencoded` body with such a `query`, or with an `extensions` parameter holding `persistedQuery`;
- the same `query` or `extensions` parameter in the URL, on any method except `OPTIONS`.

JSON, form and `application/graphql` bodies and the query string are inspected, compressed bodies included. A body that cannot be inspected is refused like one that carries an operation.

A `query` that is not a GraphQL document is not an operation: `{"query":"shoes"}`, `?query=shoes` and an Elasticsearch-style `{"query":{…}}` reach your origin as before. A request that is not refused is forwarded exactly as the client sent it, byte for byte.

The refusal applies while at least one GraphQL check is in force: blocked introspection, the operation allowlist (`persisted_query.only_registered`), a client profile with `persisted_query = "required"`, or a depth, document size or selection limit on the service or on a profile. The introspection block and the three limits are on by default, so a service that never changed them refuses. A service that has switched every one of them off has nothing an operation could go around, and forwards it. An empty `request.graphql_paths`, which a deployment without a schema always runs with, refuses nothing.

It runs on every request, cache hits included, after the request size check, so a body over `max_body_bytes` is answered with `413` first. Your logs carry `INGRESS_OFF_PATH_GRAPHQL` with `form` (where the operation was found: `json_body`, `document_body`, `form_body` or `query_string`) and `protections` (the checks it would have gone around), which tells you which client to point at the GraphQL path. See codes.

Three forms are not recognized off the configured paths: a `multipart/form-data` upload carrying an `operations` part, the operations sent over a WebSocket after the upgrade, and a persisted document sent as a bare `documentId`. Mount your GraphQL server on the configured paths only, and none of them matters.

Credentials

- **JWTs are verified at the edge**, with an explicit algorithm allowlist, and a token that does not verify is refused before your origin is called. See authentication.
- **A response computed for one caller is not shared**, and where GraphPilot cannot tell it is refused rather than shared. See authenticated requests.
- **Requests to your origin are signed**, so your origin can refuse everything that did not come through GraphPilot. See request signing.

Refusing callers you did not declare

Naming your clients lets the edge refuse one it does not recognize, or one running a build you have retired, before your origin is reached.

```
[client]
unknown = "reject"

[[client.profile]]
name        = "web-app"
when        = "auth:idp.client_id == 'web-app'"
version     = "header:gp-client-version"
min_version = "2.1.0"
```

What this is worth depends entirely on what `when` reads, and the difference is larger than it looks. Over a **header**, it is the caller's own claim about itself: it catches stale deployments and misrouted traffic, and it stops nobody who has read your public app's requests. Over a **verified claim**, as above, it is reading something the caller could not have invented, and only then is it a control an attacker has to get past. The difference, and what it costs, is worked through in matching on a verified claim.

A floor for a mobile app needs one more step when the app reports its build number after a hyphen, as in `apollographql-client-version: 4.2.0-118`. A hyphen marks a prerelease, and a prerelease sorts below its release, so `4.2.0-118` is older than `4.2.0` and a floor of `4.2.0` would refuse the very build it was written for. Read the release part only:

```
[[client.profile]]
name        = "ios-app"
when        = "header:apollographql-client-name == 'ios-app'"
version     = "split(header:apollographql-client-version, '-', 0)"
min_version = "4.2.0"
```

`4.2.0-118` is now read as `4.2.0` and served, and `4.1.3-97` as `4.1.3` and refused with `403` and `CLIENT_VERSION_UNSUPPORTED`. The request's **Client** tab in Insights shows the version as it was read. Both headers are set by the app itself, so this retires old builds and stops nobody who sends a newer number. The recipe retiring old mobile app versions walks through it end to end.

A profile can also give one client a smaller body limit, a shallower depth ceiling, a smaller document size or selection limit, or a hashes-only rule, each stricter than the service-wide setting it narrows. See telling your clients apart.

What the edge does not check

Stated plainly, because these are the ones an attacker will reach for. Every control above runs on an HTTP request that reaches the edge. What follows is traffic those controls never see, and limits GraphPilot does not have.

Not checked	What it means for you
WebSocket messages	A WebSocket upgrade passes your auth providers and is signed, and is then handed to your origin as a tunnel. Nothing after the handshake is read: no depth, document size, selection or introspection check, no persisted-query rule, no size limit on a message, and no check of a credential sent in `connection_init`. Every subscription operation is checked only by your origin. No setting refuses WebSocket upgrades: a handshake on any path, GraphQL or not, that passes the checks on the handshake itself (your auth providers, for example) is handed to your origin, whatever your persisted-query, introspection and limit settings say. An API that serves no subscriptions refuses the upgrade at its origin. See subscriptions and WebSockets.
Requests sent straight to your origin	A request that does not go through GraphPilot is checked by nothing on this page. Until your origin verifies the signature, anyone who learns its address can go around every limit here.
GraphQL on paths outside `request.graphql_paths`	The GraphQL checks run only on the configured paths. An operation GraphPilot recognizes on another path is refused while any GraphQL check is in force, but the three forms it does not recognize and every path on a service that has switched all GraphQL checks off reach your origin unchecked. Serve GraphQL only on the configured paths.
Bodies forwarded unchecked	A body kind you list in `request.forward_unchecked_bodies`, such as a multipart upload, reaches your origin without a GraphQL check. Your origin is its only reader.
Alias, directive and token counts	There is no limit of their own. An alias counts as a selection, so alias spam is bounded by `max_selections`, and the number of tokens only by `max_document_bytes`. Directives are not counted, so the document size is the only bound on them.
Query cost analysis	Depth and the number of selections are bounded; cost is not. A shallow query over a large list is cheap to write and expensive to answer, and nothing at the edge prices it.
Rate limiting	Not available. There is no per-second limit and no cap on requests in flight. A caller can send as many operations as they like. Put a rate limiter in front of GraphPilot, or at your origin for the traffic that reaches it.
Field-level authorization	The edge verifies who is calling and never decides what they may see. That stays with your origin.
CORS or `Origin`-header allowlisting	The edge never refuses a request because of its `Origin`, and a preflight is answered by your origin or a cached copy of its answer. Making the edge's own refusals readable in a browser is a separate matter, below. (Restricting your origin to GraphPilot's IP ranges is a different thing, covered in origin allowlist.)
Payload inspection	Bodies are parsed as GraphQL, not scanned.
Client identity, on its own	A client profile matched on a header is friction, not a boundary: the header is whatever the caller sent. Only a profile matched on a verified claim refuses someone who is trying.

The mitigations that do exist for cost and volume are the allowlist and the document limits above, plus a short freshness lifetime on anything expensive, so that a repeated expensive query is answered from cache rather than from your origin.

One edge case is covered, so it is not on the list:

- **A body on a GraphQL path that is not a single JSON operation** is not a way around the document controls. An `application/graphql` document is checked like any other operation. Anything else (a JSON batch, a multipart upload, text, a missing content type) is refused with `415` and `UNSUPPORTED_GRAPHQL_BODY`, and forwarded unchecked only if you list its kind in `request.forward_unchecked_bodies`. A JSON body that repeats a key or nests more than 128 levels deep is refused with `400` and `INVALID_GRAPHQL_REQUEST` whatever that setting lists, and so is a `POST` (or a `GET` with a body) that also carries GraphQL parameters in its URL, which some servers read before the body. A path below a GraphQL path, or the same path in another letter case or encoding, is the GraphQL path. A request to any other path is plain HTTP: the controls that run on every request apply to it, and the GraphQL ones do not. An operation sent there is refused with `404` and `NOT_A_GRAPHQL_PATH` rather than forwarded, apart from the three forms named there, so still serve GraphQL only on the configured paths.

CORS on refusals from the edge

`response.cors.allow_origins` lists the origins allowed to read GraphPilot's own rejections. For those origins, an edge rejection carries CORS headers, so your app can read its error code instead of seeing a network error. See edge rejections.

Enforce the same limits at your origin

The edge limits are a first filter, not the only one. Keep a second copy of them where every operation ends up, so that a path the edge does not cover, a setting switched off by mistake, or a request that never went through GraphPilot still meets a limit:

- **Lock your origin to the edge.** Verify `GP-Signature` on every request, the WebSocket upgrade included, and refuse what does not carry a valid one. See request signing and, as a second layer, the origin allowlist.
- **Limit the WebSocket at your origin.** Set a maximum message size on your WebSocket server, verify the credential in `connection_init` before you acknowledge the connection, and close the connection when that credential expires. See subscriptions and WebSockets.
- **Limit operations in your GraphQL server.** Apply a depth limit, an alias or field count limit and, where your server offers one, a cost limit to every operation it executes, over HTTP and over the WebSocket alike. The edge defaults (depth 10, 4,000 selections, a 64 KiB document) are a reasonable starting point. Keep introspection off in production there too.
- **Limit the request body.** A body limit in your HTTP server, no larger than `request.max_body_bytes`, keeps that bound in place for any request that reaches your origin some other way.

Related

- Go-live checklist for the order to do these in.
- Limits for the platform ceilings behind these limits.
- Codes for what each refusal reports.
- Telling your clients apart for declaring clients and giving each its own limits.

- Subscriptions and WebSockets for what your origin checks inside a WebSocket.

SECURITY

Persisted queries

> **GraphQL only** A plain HTTP API has no operations to register.

A persisted query is an operation your service already knows. The client sends a SHA-256 hash instead of the document, the edge looks the hash up in your service's registry and puts the document it finds back into the request. What reaches your origin is an ordinary GraphQL request, so **your origin needs no support for any of this**; see what your origin receives. Three things follow, and you configure them separately:

- **Requests get small.** A hash is 64 characters whatever the operation is. That matters most for `GET` requests, where the operation travels in the URL and a large document runs into the 8 KB URL limit.
- **The registry can become an allowlist.** If the only operations your service runs are the ones you registered, an attacker cannot craft a new one, however valid it is.
- **Your inventory becomes readable.** `gpilot pq list` lists the operations you registered and `gpilot apq ls` the ones your clients registered on their own. Together they answer "what runs against this service".

How a client sends one

GraphPilot speaks the Apollo protocol: the hash travels in the `persistedQuery` extension, version `1`.

```
{
  "variables": { "id": "42" },
  "extensions": {
    "persistedQuery": {
      "version": 1,
      "sha256Hash": "8f3c…"
    }
  }
}
```

The same request as a `GET`, which is the form that is cacheable in shared caches downstream of the edge:

```
GET /graphql?
extensions=%7B%22persistedQuery%22%3A%7B%22version%22%3A1%2C%22sha256Hash%22%3A%228f3c…
%22%7D%7D&variables=%7B%22id%22%3A%2242%22%7D
```

> ⚠ **Not documentId**
>
> Some clients and servers exchange persisted operations as a top-level `documentId` field. GraphPilot does not read that field. Configure your client for the Apollo-style `extensions.persistedQuery` shape shown above.

The hash is the SHA-256 of the document text exactly as it is registered, hex encoded. It is not normalized, reformatted or reordered first, so the same operation written with different whitespace is a different hash.

What your origin receives

The hash never leaves the edge. A client sends this:

```
{
  "variables": { "id": "42" },
  "extensions": {
    "persistedQuery": { "version": 1, "sha256Hash": "8f3c…" }
  }
}
```

and your origin receives this:

```
{
  "query": "query Product($id: ID!) { product(id: $id) { name price } }",
  "variables": { "id": "42" }
}
```

The document came from the registry, not from the client. If a request carries a hash and a document together and the registry holds that hash, the registered document runs; otherwise anyone holding one valid hash could run a different operation under it.

The `persistedQuery` extension was removed. The edge adds selections of its own further down the pipeline, for instance the fields your surrogate keys are built from, so the hash would no longer describe the document your origin receives.

Everything below sees the resolved document. Cache lookup, surrogate keys, selection set expansion and the request limits all run on the operation, not on the hash, so an operation sent as a hash and the same operation sent in full share one cache entry.

Persisted queries are therefore a client-side and edge-side change only: you can switch them on and off for a service whose origin you cannot modify.

Registering your operations

Push the manifest your client tooling already produces:

```
gpilot pq push ./src/gql/persisted-documents.json
```

The format is detected from the file's contents rather than its name, because the same shape ships under several names:

Format	Produced by	Usual file name
`apollo`	Apollo Client, Apollo Kotlin, Rover	`persisted-query-manifest.json`
`map`	graphql-codegen, Relay, Hive, urql	`persisted-documents.json`, `persisted_queries.json`, `po.json`
`relay-array`	Relay compiler	`persisted_queries.json` (array form)
`operation-output`	Apollo Kotlin and Apollo iOS (legacy)	`operationOutput.json`
`graphql-source`	Hand-written operations	`*.graphql`, `*.gql`

`gpilot pq formats` prints this list with a link to each tool's documentation.

Every entry is verified before it is written: the id the manifest declares must be the SHA-256 of the document beside it. An entry whose id is anything else is reported rather than repaired, because a document registered under a hash your clients do not send is never found. The report names the one configuration line that fixes it, per tool. The common cases are a Relay compiler still on its `md5` default and a custom `createOperationId` in an Apollo config.

`gpilot pq push --rehash` registers each document under GraphPilot's hash instead of the id the manifest declares. That is only correct when you regenerate your clients at the same time; on its own it registers documents under ids no client sends.

A push is **additive and idempotent**. Nothing is removed, and re-sending an entry does nothing, so the operations of a release that is still in the field keep working while the next one registers alongside them. Remove operations explicitly when you want them gone:

```
gpilot pq rm 8f3c… 91ab…
```

That revokes the entries and purges everything cached for them.

In your deployment pipeline

Push before you ship the client, not after. A client that reaches a user before its manifest reaches the registry sends hashes nothing can resolve. Under `apq` that costs one extra round trip per operation; under `only_registered` it is a `403`.

```
gpilot pq push ./dist/persisted-documents.json --dry-run   # parse and verify only
gpilot pq push ./dist/persisted-documents.json
#  … then deploy the client
```

Configuration

Persisted queries are configured under `[persisted_query]` in your service's `gpilot.toml`: four independent switches (`enabled`, `apq`, `only_registered`, `audit_unregistered`), a `preset` that names the common combinations, two cache lifetimes and two document ceilings. Every key, its default, what each preset sets and the two refused combinations are in the configuration reference.

```
[persisted_query]
preset = "apq"
apq    = false   # resolve hashes, but let only a push fill the registry
```

`enabled` **is off by default, and that is a defense.** With resolution on, a client sending random hashes costs a registry lookup per request.

The cache lifetimes are long for documents and short for misses. A resolved document stays in the local cache of the location that resolved it for seven days, safely so: a hash points at one document forever or at nothing, and `gpilot pq rm` purges rather than waiting for expiry. "This hash resolves to nothing" is remembered for a minute, so a client repeating an unknown hash does not cause a lookup per request, and a newly registered operation is seen quickly.

A registered document can have ceilings of its own. A document your team pushed is not the same risk as one a stranger sends, so `persisted_query.max_document_bytes` and `max_selections` are the ceilings to raise when a generated document outgrows 64 KiB, rather than raising the `[request]` values for anonymous callers too. Lowering them bounds what a push can commit your edge to. A push checks each document against the ceilings your service is deployed with and marks an oversized one `REJECTED` without failing the rest of the batch. See where the limits apply.

Choosing a setup

Bandwidth only, public API. `preset = "apq"`. Clients register what they send, nothing is refused, and you need no build step. On a service anyone may call, the handshake is a write, so strangers can add to your registry and your inventory stops answering the question it exists for.

You register from CI, and your API stays open. `enabled = true` with `apq` off. Hashes resolve, free text still runs, so customers and scripts can send their own operations, and only your push fills the registry.

Allowlist. `preset = "trusted"`. Only registered operations run. Anything else is refused with `403`, including a full query sent as text and your own ad-hoc tooling, so it belongs on a service whose only callers are clients you build.

Migrating onto the allowlist. `preset = "trusted_migrating"` first. Everything still runs, and every operation the registry does not hold is logged as `PERSISTED_QUERY_UNTRUSTED_OPERATION`. Once nothing new appears, switch to `trusted`. Migrating to trusted documents walks through it.

> ⚠ **Cached responses outlive the switch**
>
> The allowlist is checked when a request misses the cache, not on a cache hit. A response cached while an operation was still allowed keeps being served after you switch to `trusted`, until it expires or is purged. Revoking the operation with `gpilot pq rm` or `gpilot apq rm` purges its cached responses too, so clear what you do not want to keep as part of the switch.

What the edge answers

A client receives `PERSISTED_QUERY_NOT_SUPPORTED` for a hash sent to a service with `enabled = false`, `PERSISTED_QUERY_NOT_FOUND` for a hash the registry does not hold, `FORBIDDEN` (`403`) for an operation `only_registered` refuses, and `METHOD_NOT_ALLOWED` (`405`) for a non-query sent as a `GET`. Your logs carry the finer codes behind them. Every code, with its status and meaning, is in codes.

`PERSISTED_QUERY_NOT_SUPPORTED` and `PERSISTED_QUERY_NOT_FOUND` are GraphQL errors in a `200` response, because that is what the Apollo protocol expects: a client treats a `4xx` as a transport failure and would not resend the document.

If the registry is unreachable, an `only_registered` service refuses rather than falling back, because an allowlist that opens in an outage has stopped being one. Every other setup runs the operation. Locations that have resolved a hash recently keep serving it from their local cache either way, so an outage does not stop traffic everywhere at once.

Notes and limits

A hash addresses one operation, which is also what keeps its cache entry still. A client that sends whole `.gql` files instead is keyed by the file, so editing any operation in it changes the key of every operation in it. See the cache key covers the whole document.

Documents from your registry are subject to the same request limits as documents from clients, unless `persisted_query.max_document_bytes` or `max_selections` set their own. `request.max_query_depth` sets its depth ceiling for both, separately.

An operation sent to a path that is not a GraphQL path does not go around the allowlist: it is refused with `404` and `NOT_A_GRAPHQL_PATH`, a persisted query hash included. The allowlist covers every body sent to a GraphQL path (`request.graphql_paths`), not only a JSON operation. An `application/graphql` document is read as one operation and checked like any other; it reaches your origin as JSON, with `Content-Type: application/json`. Any other body GraphPilot cannot read as a single operation (a JSON batch, a multipart upload, text, no content type, or JSON that names no operation, such as a Relay-style `{"doc_id": …}` persisted query) is refused with `415` and `UNSUPPORTED_GRAPHQL_BODY` before it reaches your origin. It is forwarded unchecked, as the bytes the client sent, only if you list its kind (`batch`, `multipart` or `other`) in `request.forward_unchecked_bodies`, which takes it

past the allowlist, the introspection block and the document limits. A JSON body that repeats a key, or nests more than 128 levels deep, is never forwarded that way: it is refused with `400` and `INVALID_GRAPHQL_REQUEST`, because your origin might read a different operation out of it than GraphPilot checked.

Persisted queries have no effect on subscriptions. A WebSocket upgrade bypasses the edge's GraphQL handling entirely and is relayed as a tunnel, so nothing sent over that connection is resolved against the registry.

SECURITY

Request signing

GraphPilot signs **every request it forwards to your origin** with a per-service **signing key** and adds the signature to the request headers. Signing is always on. To *verify* those signatures you need the key, which the portal shows you in full once: when you create the service there, and again each time you rotate it. Verifying is up to your origin, and it is recommended.

A valid signature confirms that a request passed through GraphPilot rather than reaching your origin directly. That matters most when you rely on GraphPilot to enforce token or API-key validation: a signed request has been through those checks, an unsigned or invalid one has not. Signing does not replace TLS or your own user authentication.

> ⚠ **A WebSocket is signed once**
>
> A WebSocket handshake is signed like any other request, but the connection it opens is a tunnel: the frames that follow are not signed, and nothing in them was inspected at the edge. A valid signature on the handshake tells you the connection came through GraphPilot, and nothing about the operations sent over it. See Subscriptions and WebSockets.

The signing key

A service is issued its signing key when it is created, and is signed with it from its very first request. **The key is shown once: in the portal, in the dialog that follows creating the service; from the CLI, in the output of `gpilot service create`.** Copy it then and deploy it to your origin, and your origin can verify from the first request it receives.

> ⓘ **Created the service with the CLI?**
>
> `gpilot service create` prints the signing key once, in its **Signing key** row, and as `signing_key` with `-o json` or `-o yaml`. If you did not keep the key, get a new one by rotating in the portal. The service is signed from its first request either way; until then, only your origin's ability to verify is missing.

> ⚠ **Copy it when you see it**
>
> GraphPilot shows each key in full only once: when the service is created (in the portal dialog or in the output of `gpilot service create`), or, for a new key, when you rotate. Copy it straight into your secret manager.
>
> Everywhere else the key is **masked**: the `gpsk_` prefix, a run of bullets, and the key's last four characters.
>
> ```
> gpsk_••••••••••••••••••••••7fQa
> ```
>
> The last four characters tell one key from another, such as the active key from the pending one during a rotation. They are not enough to reconstruct the key.
>
> **GraphPilot cannot recover a key you did not copy.** If you lose it, rotate. A key nobody holds is not a hole in the signing, only in your ability to verify it.

The key is a secret. Treat it like a password: store it in a secret manager, inject it into your origin through your platform's secret store, and never commit it to source control. Anyone holding it can forge valid signatures.

The dialog that shows it

The full key appears in one dialog, **Copy your signing key** after creating the service or **Copy your new signing key** after a rotation, in a read-only field with a copy button. **Escape and a click outside it do nothing**, and it has no close cross: the only way out is **I have copied the key**, so the key cannot be lost to a stray click. After that, the service page only ever shows the mask.

How GraphPilot signs a request

GraphPilot adds two headers to each forwarded request:

```
GP-Timestamp: 1718700000
GP-Signature: v1=Base64(HMAC-SHA256(signing_key, canonical_string))
```

- `GP-Timestamp` is the time the request was signed, as Unix seconds. It is part of the signed data and drives replay protection.
- `GP-Signature` carries the signature, Base64-encoded, prefixed with a scheme version (`v1=`) so the scheme can evolve without breaking verifiers.

Header names are case-insensitive: `gp-signature` and `GP-Signature` are the same header.

The canonical string

The signature is an HMAC-SHA256 over a **canonical string**: a deterministic representation of the request that both sides build the same way. It is the following four fields, joined by a single newline (`\n`):

```
{HTTP_METHOD}
{PATH_AND_QUERY}
```

```
{TIMESTAMP}
{BODY_HASH}
```

Field	Description
`HTTP_METHOD`	The request method, uppercased (e.g. `POST`, or `QUERY` if your service forwards reads as `QUERY`).
`PATH_AND_QUERY`	The request target as sent to your origin: the path including the query string (e.g. `/graphql?op=ping`).
`TIMESTAMP`	The same value as the `GP-Timestamp` header.
`BODY_HASH`	Lowercase hex of `SHA-256(raw_body_bytes)`. For a request with no body, this is the SHA-256 of the empty string.

Binding the method and path stops a captured signature from being replayed against a different endpoint.

The signature is then:

```
signature = Base64( HMAC-SHA256(signing_key, canonical_string) )
```

The HMAC key is the **complete key string as UTF-8 bytes**, `gpsk_` prefix included. Do not strip the prefix, and do not base64-decode the rest, even though it looks like base64: the string you copied is the key exactly as it is.

What the signature does not cover

The signature covers the four fields above: the method, the path with its query string, the timestamp and the SHA-256 of the body. It covers **no request headers**.

A valid signature proves that the request came through your GraphPilot service and that those four parts were not changed on the way. It says nothing about whether the other headers are true. In particular, it does not vouch for forwarding headers a client set itself, such as `X-Forwarded-For`, `X-Forwarded-Host`, `X-Forwarded-Proto`, `Forwarded` or `X-Real-IP`.

- **Do not trust forwarding headers because a request is signed.** This is the “trust proxy” setting in many frameworks. By default GraphPilot removes the forwarding headers a client sent before the request reaches your origin (`origin.forwarded_headers`, default `"strip"`). With `"pass"`, your origin receives them exactly as the client sent them, so a framework that trusts them takes the client’s word for its address, host and scheme.
- **For the client’s real address, read `gp-client-ip`.** With `origin.forward_client_ip` on, the edge sets this header to the address it saw for the connection. A client cannot forge it: a value a client sends is removed, and the edge writes its own. See request headers your origin receives.

Verify the signature on your origin

Reject any request that fails verification. The steps:

1. Read `GP-Timestamp` and `GP-Signature`. If either is missing, reject the request.
2. Check the timestamp against your replay window. If it is too old or too far in the future, reject the request.
3. Rebuild the canonical string from the request, using the **raw** request body (the exact bytes received, before any JSON parsing or re-serialization).
4. Compute `Base64(HMAC-SHA256(signing_key, canonical_string))`.
5. Compare your value against the one in the header using a **constant-time** comparison, to avoid leaking information through timing. Accept only on a match.

Node.js

```
import { createHmac, createHash, timingSafeEqual } from "node:crypto"

const TOLERANCE_SECONDS = 300

// `rawBody` must be the exact bytes received (Buffer or string),
// not a re-serialized object.
export function verifyGraphPilotSignature({ method, pathAndQuery, rawBody, headers,
signingKey }) {
  const timestamp = headers["gp-timestamp"]
  const signatureHeader = headers["gp-signature"]
  if (!timestamp || !signatureHeader) return false

  // Reject stale or future timestamps.
  const age = Math.abs(Math.floor(Date.now() / 1000) - Number(timestamp))
  if (!Number.isFinite(age) || age > TOLERANCE_SECONDS) return false

  const bodyHash = createHash("sha256").update(rawBody).digest("hex")
  const canonical = [method.toUpperCase(), pathAndQuery, timestamp, bodyHash].join("\n")
  const expected = createHmac("sha256", signingKey).update(canonical).digest("base64")

  // Header looks like "v1=<base64>".
  const provided = signatureHeader.replace(/^v1=/, "")
  const a = Buffer.from(expected)
  const b = Buffer.from(provided)
  return a.length === b.length && timingSafeEqual(a, b)
}
```

Python

```
import base64
import hashlib
import hmac
```

```python
import time

TOLERANCE_SECONDS = 300

# `raw_body` must be the exact bytes received, not a re-serialized object.
def verify_graphpilot_signature(method, path_and_query, raw_body, headers, signing_key):
    timestamp = headers.get("GP-Timestamp")
    signature_header = headers.get("GP-Signature")
    if not timestamp or not signature_header:
        return False

    # Reject stale or future timestamps.
    try:
        age = abs(int(time.time()) - int(timestamp))
    except ValueError:
        return False
    if age > TOLERANCE_SECONDS:
        return False

    body_hash = hashlib.sha256(raw_body).hexdigest()
    canonical = "\n".join([method.upper(), path_and_query, timestamp, body_hash])
    expected = base64.b64encode(
        hmac.new(signing_key.encode(), canonical.encode(), hashlib.sha256).digest()
    ).decode()

    # Header looks like "v1=<base64>".
    provided = signature_header.removeprefix("v1=")
    return hmac.compare_digest(expected, provided)
```

Go

```go
package graphpilot

import (
  "crypto/hmac"
  "crypto/sha256"
  "encoding/base64"
  "encoding/hex"
  "net/http"
  "strconv"
  "strings"
  "time"
)

const toleranceSeconds = 300

// rawBody must be the exact bytes received, read before any parsing.
```

```
func VerifyGraphPilotSignature(method, pathAndQuery string, rawBody []byte, headers
http.Header, signingKey string) bool {
  timestamp := headers.Get("GP-Timestamp")
  signatureHeader := headers.Get("GP-Signature")
  if timestamp == "" || signatureHeader == "" {
    return false
  }

  // Reject stale or future timestamps.
  ts, err := strconv.ParseInt(timestamp, 10, 64)
  if err ≠ nil {
    return false
  }
  age := time.Now().Unix() - ts
  if age < 0 {
    age = -age
  }
  if age > toleranceSeconds {
    return false
  }

  bodyHash := sha256.Sum256(rawBody)
  canonical := strings.Join([]string{
    strings.ToUpper(method), pathAndQuery, timestamp, hex.EncodeToString(bodyHash[:]),
  }, "\n")
  mac := hmac.New(sha256.New, []byte(signingKey))
  mac.Write([]byte(canonical))
  expected := base64.StdEncoding.EncodeToString(mac.Sum(nil))

  // Header looks like "v1=<base64>".
  provided := strings.TrimPrefix(signatureHeader, "v1=")
  return hmac.Equal([]byte(expected), []byte(provided))
}
```

> ⚠ **Use the raw body**
>
> Most web frameworks parse and re-serialize the request body before your handler sees it, which changes the bytes (whitespace, key order) and breaks the hash. Capture the raw body before parsing. In Express, for example, keep the buffer with `express.json({ verify: (req, _res, buf) ⇒ { req.rawBody = buf } })`.

Check your implementation against a known vector

Run your verifier against this vector before you trust it in production. GraphPilot's own test suite pins the same one.

signing key

```
gpsk_test_key
```

request body (no trailing newline)

```
{"query":"{ me { id } }"}
```

canonical string (three newlines, no trailing newline)

```
POST
/graphql?op=ping
1718700000
64d4ae404c393f3916a308845f1df6e2ccd4243d889d3883f9b2dd7da9d31114
```

expected signature

```
708YYmz8ifG4D6z8Fl1XVh1oIDtn8NFRhNkTZPR+ZWM=
```

The last line of the canonical string is the SHA-256 of that body, so the vector checks your body hashing as well as the HMAC. The header your origin would receive for it is `GP-Signature: v1=708YYmz8ifG4D6z8Fl1XVh1oIDtn8NFRhNkTZPR+ZWM=`.

If you get a different value, the usual causes are, in order of likelihood: the body was re-serialized before hashing rather than hashed raw, the method was not uppercased, the query string was dropped from the path, or the lines were joined with `\r\n`.

Trust and transport

Signing is one of three things that decide whether traffic reaching your origin can be trusted. The other two are not GraphPilot settings: the connection itself, which is encrypted only when your origin URL is `https://` (see TLS and ports), and who may connect at all, which your firewall decides (see the origin allowlist). An allowlist admits the whole edge network; only the signature proves a request came through **your** service.

Replay protection

A signature alone does not stop someone from capturing a valid request and sending it again; the timestamp closes that gap. Reject any request whose `GP-Timestamp` is more than **5 minutes (300 seconds)** away from your origin's current time, in either direction, and keep your origin's clock in sync (NTP) so legitimate requests are not rejected.

For stricter protection, also track the signatures (or a nonce) you have already accepted within the window and reject duplicates. The timestamp window bounds how much state you need to keep.

Rotating the signing key

Rotate the key periodically, immediately if it may have been exposed, and to replace the key you were shown at creation if you did not keep it. The new key is shown in full exactly once, like the first.

GraphPilot signs with **exactly one active key at a time**, so **you** control the cutover: a new key is created *pending*, you deploy it to your origin, and only then do you activate it.

> **If the key is already leaked, invert the order**
>
> The staged sequence below keeps the old key valid while you deploy the new one, which is the wrong shape for an active leak. For a leak, rotate, copy the new key, and **activate it immediately**: the leaked key is useless from that moment, and every request fails verification at your origin until you have deployed the new key there. You are trading a short outage for closing the exposure.
>
> If you cannot accept the outage, the interim measure is the IP allowlist: a leaked key is only useful to somebody who can also reach your origin.

1. In **Organization** → **Service** → **Settings** → **General**, scroll to the **Danger zone** and, under **Signing key**, choose **Generate a new pending key**. This creates a new key in a **pending** state. GraphPilot keeps signing with the current **active** key, so nothing changes yet. The pending key is shown **in full, once**, in the dialog that follows. Copy it there: from then on it is masked like every other key, and GraphPilot cannot show it to you again. If you lose it before step 2, choose **Cancel rotation**, which discards the pending key and changes nothing else, and then generate another one: while a pending key exists, that is the only way to get a fresh one.
2. Add the pending key to your origin so it accepts a request that verifies against **either** the active key or the pending key. Deploy this change.
3. Back in the portal, choose **Activate pending key**. The dialog asks you to tick that your origin already accepts the pending key before it will let you go through with it. GraphPilot then signs every request with the new key, and because your origin already accepts it, no request is rejected during the switch.
4. Once the change has propagated and any in-flight requests signed with the old key have drained (a few minutes, comfortably within the replay window), remove the old key from your origin. The rotation is complete.

The only moment your origin needs to accept two keys is the window between step 2 and step 4. Outside of it, a single key is active and accepted on both sides.

Best practices

- Verify on **every** request and reject anything that does not verify, instead of logging and continuing.
- Use a **constant-time** comparison, and hash the **raw** request body.
- Stage the pending key on your origin before activating it.

Related

- Origin allowlist to refuse traffic that did not come from the edge network.

- Authentication for verifying your users' tokens at the edge.
- Hardening a public API for the other controls GraphPilot applies.

SECURITY

Origin allowlist

If your origin is reachable over the public internet, anyone who learns its address can send it a request directly and skip GraphPilot. Verifying the request signature catches that, but only after the request has arrived and been parsed. Refusing it at the network layer is the cheaper first line.

GraphPilot reaches your origin from a published set of IP ranges. Allow those at your origin's firewall or load balancer, and nothing else needs to be open for GraphPilot to work.

Where to read the current set

In the portal, under **Organization** → **Service** → **Settings** → **General**, in the section the portal titles **Origin allowlist**. The page shows IPv4 and IPv6 separately, with a copy button for each family and one for the whole set. Copied values are one range per line, in CIDR notation, which is the form most firewalls and scripts split on.

The same set is on the GraphPilot API, which is what to read if you want to build the rule from a script:

```
query {
  services(condition: { subdomain: "my-shop" }) {
    ipRanges {
      ipv4
      ipv6
    }
  }
}
```

`services` returns a list, holding the one service with that subdomain. `ipv4` and `ipv6` are lists of ranges in CIDR notation. Reading the field needs the `service:view` permission, so an API key for a script needs nothing more.

The ranges are the same for every service in your organization today: they describe Fastly's edge network, which GraphPilot runs on, not your service. Read them per service anyway; the field is per service so that this can change.

With shielding, only the shield location contacts your origin for most eligible requests, so far fewer edge nodes do. The ranges to allow stay the same, because the shield is part of the same network. Keep the whole set in your rule: when the shield is unreachable before a request is sent, the edge location falls back and reaches your origin directly, and requests that never go through the shield (subscriptions, other plain HTTP methods, and requests that skip the cache) always reach your origin from their own edge location.

> **Note**
> Allow only the family your origin actually routes. Pasting IPv6 ranges into a rule that cannot express them is a common way to have the whole rule rejected, which then leaves the origin either wide open or fully closed depending on how the firewall fails.

The set changes

The ranges are published by Fastly and they move when the network does: new ranges are added as capacity is added, and old ones are withdrawn. **There is no version of this list that stays correct forever**, and GraphPilot does not promise a fixed number of locations.

Two things follow from that:

- **GraphPilot does not notify you when it changes.** There is no email, no webhook, and no entry you can subscribe to. The portal and the API always show the current set; nothing pushes the change to a rule you have already written.
- **A rule built once from a copy will eventually be wrong.** The failure is one-sided and quiet: traffic from a newly added range is dropped by your firewall, so those requests fail to reach your origin while everything from the older ranges keeps working. What you see are origin connection failures on a fraction of traffic, not an obvious outage.

The way to stay correct is to re-read the set rather than to remember it:

- Read `ipRanges` from the API on a schedule (daily is comfortably often enough) and reconcile the firewall rule from what it returns, in whatever form your firewall automates.
- If you cannot automate it, put a recurring reminder against the portal page instead, and check it again whenever you are investigating origin connection failures that affect only part of your traffic.
- Widen before you narrow: add new ranges as soon as they appear, and remove withdrawn ones afterwards. Removing first is what turns a routine change into failed requests.

Fastly also publishes the same list at api.fastly.com/public-ip-list if you would rather read it from the source. Reading it from GraphPilot and reading it from Fastly give the same answer today, and only the field on the GraphPilot API is kept working.

What an allowlist does not do

> **Caution**
> An IP allowlist is a coarse control: it admits the whole edge network, not only your service. Anyone else with a service on the same network is inside your rule. It raises the bar against someone who merely learns your origin's address; it does not prove a request came through **your** service. Verifying the signature is what proves that, and the two are meant to be used together.

There is **no mutual TLS** to the origin today, and **no per-service source address**. If your threat model needs a source address that only your traffic uses, an allowlist will not give it to you.

An empty or unreachable list is not an instruction to close the firewall. If the portal cannot load the ranges, leave the existing rule in place and try again rather than narrowing it on a failed read.

Related

- Request signing to prove a request came through your service.
- GraphPilot API reference for `Service.ipRanges`.
- Hardening a public API for the controls GraphPilot applies itself.

Deploy and operate

DEPLOY AND OPERATE

Deployments

A deployment takes your configuration and, for a GraphQL API, your schema, builds them into an edge package, and puts it live at every edge location. It is the only way anything you configure takes effect.

What goes into one

- `gpilot.toml`, the service configuration. The CLI also reads `gpilot.yaml`, `gpilot.yml`, `gpilot.json` and `gpilot.json5`, and converts them to TOML before uploading.
- `schema.graphql`, your GraphQL API's schema, for a GraphQL API. Surrogate keys and cache directives are derived from it. Where a deployment that names no schema takes one from is the service's schema source: the newest upload by default, or introspection of your origin at deploy time.

A deployment without a configuration file is valid too: every setting is then at its default, the deployment records no config version, and the portal's **Config** page lists each default value. The portal offers it as **Default configuration**, the API as `withoutConfig`, and the first deployment of a new service's setup is one.

A plain HTTP service needs no schema. Without one, the deployment is built from the configuration alone: GraphQL features are off, `request.graphql_paths` is `[]`, and every path is served as plain HTTP. Whether a service has a schema is decided per deployment, so the next deployment with a schema turns GraphQL on again, and one without a schema turns it off.

What you upload is compiled into the package by the deployment build. Nothing is read at request time, which is why a configuration change needs a deployment and why a request never pays for reading one.

The origin URL is **not** part of this. It belongs to the service and changes with `gpilot service edit`, taking effect on the next deployment. Only its scheme, host and port are used; see TLS and ports.

Deploying from the CLI

```
gpilot deploy                                # ./schema.graphql and ./gpilot.toml
gpilot deploy --schema api.graphql --config prod.toml
```

Before it starts, `gpilot deploy` shows whether the configuration and the schema are new, changed or unchanged, and asks you to continue. Without a schema file it says `No schema found: deploying as a plain HTTP service, GraphQL features are off.` before asking, and shows `Schema removed: GraphQL features off` when the live deployment had a schema. Pass `--yes` to skip the question. In a non-interactive run, such as CI, it proceeds without asking.

The command then waits for the deployment to finish, up to `--timeout` (20 minutes by default), and prints each step as it completes, followed by any findings. Pass `--detach` to return as soon as the deployment is accepted. `--force-build` builds again even when the schema, configuration and GraphPilot version are unchanged.

If the progress stream closes before the deployment reports how it ended, `gpilot deploy` keeps checking the deployment until it ends: completed exits `0`, failed exits `1`. `--timeout` bounds the whole wait, watching and checking together. If the deployment is still open when it runs out, the command exits `7`, meaning the outcome is unknown and the deployment may still be running; check it with `gpilot status`.

```
gpilot status          # recent deployments
gpilot cancel          # stop one that is still running
```

Deploying from the portal

The portal lists a service's deployments under **Deployments**, where you can also start one (**New deployment**), redeploy an earlier one, or stop a build. A new deployment can take each file from the latest upload, an earlier deployment or your machine, or deploy without a schema or without a configuration file. See Deployments in the portal.

What happens while it runs

A deployment moves through `STARTED` → `PENDING` → `COMPLETED`, or ends in `FAILED` or `CANCELLED`. These are the GraphPilot API's spellings. The CLI writes the same statuses in lowercase (`started`, `pending`, `completed`, `failed`, `cancelled`), in its tables and in `-o json` alike, so a script that checks `gpilot status` compares against `completed`.

Your service keeps serving the whole time. The current version answers traffic until the new one is ready, and traffic moves over when it is. A deployment that fails to build never becomes live, so a broken configuration costs you a failed deployment rather than an outage.

One deployment of a service runs at a time. Starting a new one while another is still `STARTED` or `PENDING` cancels the running one: it ends `CANCELLED`, never goes live, and its `errorMessage` reads `Replaced by a newer deployment of this service before it went live.` A deployment GraphPilot stopped for another reason reads `Cancelled before it went live.` A deployment canceled on request (`gpilot deployment cancel`, `cancelDeployment`, or **Stop this build** in the portal) has no `errorMessage`.

> **Note**
>
> A first deployment for a new service takes longer than later ones, because the edge service and its certificate have to be provisioned before anything can be built.

The cache survives a deployment

Cache entries are keyed by the operation, not by the deployment that produced them, so **deploying does not clear the cache**. That is usually what you want: a configuration change should not cost you your hit rate.

It does mean a change only affects responses stored *after* it. Two cases where that matters:

- **You shortened a freshness lifetime (TTL).** Entries written under the old one keep it until they expire.
- **You changed a partitioning policy.** Entries written under the old one keep their old discriminator.

If the change was made because the cached data is now wrong, deploy and then purge:

```
gpilot deploy --yes && gpilot purge --all --yes
```

`gpilot purge` always asks for confirmation and refuses in CI without `--yes`, so keep the flag in scripts.

A schema change is the usual reason. Removing a field does not make cached responses containing it disappear.

Deploy the schema before the origin

GraphPilot reads a query's lifetime and scope from the schema you deployed, not from the one your origin runs. A query that selects a field, or a fragment on a type, the deployed schema does not declare is still forwarded and answered, but its response is never stored, at any depth and whatever your origin's `Cache-Control` says: nothing tells GraphPilot who may share that field's value. `gp-cache` reports `PASS` with `CACHE_SKIPPED_UNRESOLVED_SELECTION` until a deployment carries the schema that declares it.

So when a release adds fields or types:

1. deploy the new schema to GraphPilot,
2. then deploy the origin that serves it,
3. then ship the clients that query it.

Adding fields to the deployed schema before your origin has them costs nothing: no client asks for them yet. The other order costs hit rate for every query that already selects the new fields, for as long as the schema lags. Removing a field goes the other way round: stop querying it, take it out of your origin, then deploy the schema without it.

A deployment can be refused

Configuration is validated at deploy time, not at request time, so mistakes surface here. A deployment is refused when it starts, or fails in its build, rather than being quietly adjusted, when:

- an origin timeout exceeds its ceiling or is below 100 ms,
- a retry configuration cannot fit inside the per-request time budget,

- the configuration does not parse, names an unknown key in a strict section, or carries an invalid enum value or header name,
- a policy expression does not parse.

The message names the value and the limit. Nothing is silently adjusted. When the build refuses the configuration or the schema, the message starts with `The configuration was refused:` or `The schema was refused:`, followed by the reason the build gave.

> **Caution**
>
> Schema **directive** validation is the exception. An invalid `@cacheControl` or `@surrogateKey` usage is recorded as a diagnostic but does not fail the deployment. Check the findings after every deployment that changes the schema: see reading diagnostics.

While an organization is being deleted

While an organization is being deleted, or the account that pays for it, none of its services is deployed again. A new deployment is refused by the API like any other change, with “This organization is being deleted” or “This account is being deleted”. A running deployment fails, without a retry, if the deletion starts before it goes live. Its message is:

```
This organization is being deleted, so it accepts no deployments. The service stays switched off until the deletion is complete. Contact support if this is not what you expect.
```

When the account is being deleted, the message reads:

```
This account is being deleted, so its organizations accept no deployments. The service stays switched off until the deletion is complete. Contact support if this is not what you expect.
```

A deployment can fail with a support reference

A deployment that fails for a reason outside your configuration and schema does not show the build’s output. Its message is one sentence:

```
The deployment failed because of an internal error. Contact support and quote reference <reference>.
```

The reference identifies the failed run in GraphPilot’s logs. Contact support and include it. The portal, `gpilot` and the GraphPilot API (`Deployment.errorMessage`, and the `error` of the step that failed) all show the same message.

Diagnostics

Findings that do not refuse a deployment are recorded by the build as diagnostics. Each is an **error** (something you configured has no effect), a **warning** (it works as written, but probably not as meant) or a **notice** (nothing to fix, only something worth knowing about what was deployed). None of them fails the deployment.

They come from two places, your schema and your configuration. `origin` says which (`schema` or `config`), and `source` names the check that produced it:

`source`	`origin`	Located by	Examples
`cache_control`, `surrogate_key`, `auth`, `type_relation`	`schema`	`line` and `column` in your schema, where the finding has a position	An `of` that does not resolve, an `ignore` that is not a Boolean, an explicit key on a type the configuration excludes.
`policy_expression`, `cache_policy_vary`, `request_graphql_paths`, `no_schema`, `auth_provider`, `plan`	`config`	`key` (the configuration key it is about, where one applies) and the message, which names the entry. `line` and `column` are always `null`.	See below.

The configuration findings are below. A deployment built by an earlier runtime reports `policy_expression` as `radiogram` until the service is deployed again.

`source`	Severity	Finding
`policy_expression`	warning	A policy expression negates a predicate over a value the request may not carry, so it has no answer for a request without it and the key's own direction decides. The key is named, such as `client.profile[0].when`.
`cache_policy_vary`	error	A `[[cache.policy.vary]]` entry whose `name` is not a legal header name. The entry is skipped and has no effect. Reported once per entry, and the message quotes the name.
`request_graphql_paths`	error	`request.graphql_paths` is set, but the deployment has no schema. The key is ignored and every path is served as plain HTTP. Add a schema to serve GraphQL on those paths, or remove the key. Reported for any value, `[]` included.
`auth_provider`	warning	A JWT provider sets no `allowed_issuers`, no `allowed_audiences`, or neither, so it accepts tokens from any issuer or for any audience its keys verify. A provider that verifies with a public key and sets no `allowed_issuers` is not warned about: its deployment fails. The key is the provider's table, such as `auth.providers.idp`, and the message names what is missing. Reported once per provider.
`policy_expression`	notice	`telemetry.record` is `false`, so the service records no request in Insights. Requests are still served and counted for billing. The key is `telemetry.record`.
`no_schema`	notice	The deployment has no schema: it runs as a plain HTTP service, with GraphQL features off. Recorded on every deployment without a schema, with `key` `null`.

source	Severity	Finding
plan	warning	The configuration sets origin.shield, but the plan does not include shielding. The shield is deployed and stays inactive until the plan includes it, with no redeployment needed. The key is origin.shield. Added by GraphPilot from the plan at the time of the deployment, not by the build.

Reading diagnostics

Diagnostics are shown in three places:

- **In the portal. Schema → Findings** shows the findings of the live deployment, placed on the definitions they are about. Each deployment's page under **Deployments** shows its own. **Config → Findings** lists configuration keys that do nothing: unknown keys and keys that have no effect where they stand. See findings.
- **In the CLI.** `gpilot deploy` prints the diagnostics (schema and configuration) when the deployment completes or fails, unless it was run with `--detach`. It does not print the list of configuration keys that do nothing; that list is in the portal and on the GraphPilot API as `Deployment.configAnalysis`.
- **On the GraphPilot API**, as `Deployment.diagnostics`:

```
query {
  organization {
    services {
      subdomain
      deployments {
        id
        createdAt
        diagnostics {
          origin
          source
          severity
          message
          key
          line
          column
        }
      }
    }
  }
}
```

Each record has those seven fields. `severity` is `ERROR`, `WARNING` or `INFO` (a notice, which never counts as an error or a warning). `key` is `null` for a schema finding, and `line` and `column` are `null` where the finding has no position.

The list itself answers one more question. `null` means nothing was recorded: the deployment never got as far as building. An empty list is the good case: the schema and configuration were checked and nothing was found.

Staging, rollback and a way out

There is no promotion flow between environments: a deployment builds from the files in front of it. A staging environment is a second service you deploy the same files to first. To undo a deployment, redeploy an earlier one, from its page under **Deployments** (**Redeploy**) or by deploying the previous files from the CLI. And if GraphPilot itself is the problem, the way out is where your clients point. Staging and rollback walks through all three.

Taking a service down

```
gpilot service destroy <subdomain>
```

Tears down the deployed infrastructure and stops serving, while keeping the service, its schema, its configuration and its history. Deploy again to bring it back. The teardown runs in the background, so the command reports that the request was accepted rather than waiting. It asks for confirmation; pass `--yes` in a script.

To remove the service itself, with its history and hostnames, delete it in the portal (see deleting a service) or with `gpilot service delete`.

Related

- Quickstart for a first deployment end to end.
- Configuration reference for what the file accepts.
- Findings in the portal.
- Command reference for every flag.

DEPLOY AND OPERATE

Deploy from CI

This guide sets up a job that deploys your service whenever `main` changes, first for GitHub Actions, then for GitLab CI. The same steps work in any CI system that can run a shell script: install the CLI, provide an API key, run `gpilot deploy`, and act on the exit code.

What you need

- **An organization API key** with the permissions for the steps your job runs: deploying, and optionally purging and pushing persisted queries. Create it in the portal under your organization's **Settings → API**

keys. Which permissions each step needs is listed under jobs; grant only those. Name it after the pipeline: the portal shows the key's name as who made each deployment.

- **The key as a CI secret** named `GPILOT_TOKEN`: in GitHub under **Settings** → **Secrets and variables** → **Actions**, in GitLab as a masked CI/CD variable under **Settings** → **CI/CD** → **Variables**. The CLI reads that variable and uses it in place of a login.
- **Your service's subdomain**, the `<subdomain>` in `https://<subdomain>.<label>.graphpilot.cloud`.
- `gpilot.toml`, and for a GraphQL API `schema.graphql`, in the directory the step runs in, or paths to them (`--config`, `--schema`). If your schema is generated, generate it in an earlier step; the Apollo Server, GraphQL Yoga and PostGraphile pages show how.

1. Add the job

GitHub Actions

.github/workflows/graphpilot.yml

```yaml
name: Deploy to GraphPilot

on:
  push:
    branches: [main]

# Deploy one push at a time, in order, rather than racing two deployments.
concurrency:
  group: graphpilot-shop-api
  cancel-in-progress: false

jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      GPILOT_TOKEN: ${{ secrets.GPILOT_TOKEN }}
    steps:
      - uses: actions/checkout@v5

      - name: Install the gpilot CLI
        run: |
          curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
          echo "$HOME/.local/bin" >> "$GITHUB_PATH"

      - name: Deploy
        run: gpilot deploy --service shop-api --yes
```

Replace `shop-api` with your subdomain.

GitLab CI

The same three steps as a GitLab job. `resource_group` plays the part of `concurrency` above: one deployment at a time, in order.

.gitlab-ci.yml

```
deploy-graphpilot:
  stage: deploy
  image: debian:bookworm-slim
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  resource_group: graphpilot-shop-api
  before_script:
    - apt-get update && apt-get install -y --no-install-recommends curl ca-certificates
    - curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
    - export PATH="$HOME/.local/bin:$PATH"
  script:
    - gpilot deploy --service shop-api --yes
```

`GPILOT_TOKEN` reaches the job as the CI/CD variable you created. GitLab sets `CI=true` as well.

2. Understand what each part does

Installing the CLI

The install script fetches the latest release every time it runs and puts `gpilot` in `~/.local/bin`, which the workflow then adds to the `PATH` (see installation).

Installing the latest release on every run is deliberate. The CLI checks its version against a minimum that GraphPilot publishes with its releases, and **a CLI below that minimum refuses to run** and exits with code `6`. A version that is merely behind prints a warning and carries on. A pinned or cached binary works until the next minimum passes it, and then your deployments stop.

If a release you depend on is ever refused before you can upgrade, set `GPILOT_SKIP_VERSION_CHECK=1` on the step to skip the check for that run. Treat it as a temporary escape, not a setting to leave in place.

Running without prompts

GitHub Actions and GitLab CI set `CI=true`, and the CLI switches to non-interactive mode when it sees it (see interactive vs. non-interactive). In that mode `gpilot deploy` proceeds without asking, so `--yes` is not strictly needed for it. Pass it anyway: it states the intent in the workflow. `gpilot purge`, below, refuses to run without it.

What `gpilot deploy` does

`gpilot deploy` (short for `gpilot deployment create`) uploads `gpilot.toml` and, for a GraphQL API, `schema.graphql`, starts a deployment, and **waits for it to finish**, 20 minutes at most by default (`--timeout 45m` to change that). When the build completes or fails, it prints the deployment's diagnostics (schema and

configuration): warnings on a success, the reasons on a failure. The same diagnostics are in the portal, which also lists the configuration keys that do nothing; the CLI does not print that list (see findings).

Without a schema file in the working directory and without `--schema`, the service is deployed as plain HTTP, with GraphQL features off, and the job log says `No schema found: deploying as a plain HTTP service, GraphQL features are off.` For a GraphQL API, make sure the step runs where `schema.graphql` is, or pass `--schema`: a missing file does not fail the step, it deploys the service without GraphQL. A `--schema` path that does not exist does fail it.

`--detach` returns as soon as the deployment has started. The job then succeeds without knowing whether the deployment did, so use it only if something else watches the outcome.

3. Act on the exit code

The job fails whenever `gpilot` exits with anything but `0`. The codes are stable, so a script can branch on them. These are the ones a deploy step returns; exit codes has the full list.

Code	Meaning in a deploy step
`0`	The deployment completed. Warnings, if any, were printed and do not fail the step.
`1`	The deployment failed, was canceled from somewhere else, or the wait timed out. Read the printed diagnostics.
`2`	A flag or value the CLI does not know.
`3`	The API key is missing, revoked, or rejected (HTTP `401`).
`4`	The GraphPilot API could not be reached or returned an error, including a permission the key lacks: a valid key without `deployment:edit` exits `4`, not `3`.
`6`	This CLI is older than the minimum version. Reinstall to get the latest.
`7`	The outcome is unknown: the progress stream ended before the deployment reported one, and checking it until `--timeout` ran out did not settle it. The deployment is still open, or every check failed; check with `gpilot status`.

`--timeout` covers the whole wait, watching and checking together. When the stream ends early, the CLI keeps checking the deployment until it ends or `--timeout` runs out: a deployment that completed still exits `0`, one that failed exits `1`.

A `1` after the deployment started does not mean the service is down: a deployment that fails never goes live, and the previous version keeps serving (see deployments).

4. Add the optional steps

Purge after deploying

Permission `cache:purge`

Deploying does not clear the cache (see the cache survives a deployment). Entries stored before the deployment keep the lifetime and partitioning they were stored with. If a change makes those entries wrong, purge after the deploy step:

```
      - name: Purge the cache
        run: gpilot purge --all --yes --service shop-api
```

In GitLab, add the same command as another line under `script`.

Only do this when the change calls for it. A full purge sends the next request for every operation to your origin. Purging through the API covers narrower options.

Push persisted queries

If your clients send persisted queries, register the operations from your client build before that build is released:

```
      - name: Push persisted queries
        run: gpilot pq push ./dist/persisted-documents.json --service shop-api
```

`gpilot pq push` only adds: entries already registered are left alone, and nothing is removed. Add `--dry-run` to check a manifest without registering anything.

Smoke-test the deployment

A deploy step that exits `0` says the new version is live, not that it answers the way you expect. Send one request through the service and fail the job if it does not come back clean:

```
      - name: Smoke test
        run: |
          body=$(curl -sS --fail-with-body "https://shop-
api.k7m2qp.graphpilot.cloud/graphql" \
            -H 'content-type: application/json' \
            -H "GP-Correlation-ID: smoke-$GITHUB_SHA" \
            -d '{"query":"query Smoke { products(first: 1) { id } }"}')
          echo "$body" | jq -e '.errors == null and (.data.products | length) == 1'
```

`--fail-with-body` fails the step on an HTTP error status, and the `jq -e` check fails it on a GraphQL error, which arrives with `200`. The `GP-Correlation-ID` header labels the request, so on Airliner (Business) and above `gpilot requests --service shop-api --correlation-id "smoke-$GITHUB_SHA" --wait --timeout 60s` finds exactly this request in its telemetry. To deploy to staging first and promote the same commit after an approval, see the recipe promoting from staging in CI.

Check that it works

Push a change to `main` and watch the job. A green job with exit code `0` means the new version is live. Confirm it from your machine:

```
gpilot status --service shop-api
```

The newest deployment is `completed`, lowercase in both the table and `-o json`. In the portal, the service's **Deployments** list shows the same deployment with its findings. Then send a request through the service and check that it still answers as before: a `gp-cache: HIT` on a request that was cached before the deployment is expected, because the cache survives a deployment.

Pitfalls

- **A pinned or cached CLI** stops working the day the minimum version passes it, with exit code `6`. Install the latest release in every run.
- `--detach` turns every started deployment into a green job, whether it went live or not.
- `gpilot purge` **without** `--yes` is refused in CI.
- **Persisted queries pushed after the client ships** leave the first users with hashes the registry cannot resolve. Push before you release the client.
- **A schema deployed after the origin** leaves every query that selects a new field uncached until it lands (`CACHE_SKIPPED_UNRESOLVED_SELECTION`). Run the GraphPilot deploy step before the step that releases your origin; see deploy the schema before the origin.
- **One service is one environment.** To rehearse a change, deploy it to a second service first; see staging and rollback.

Related

- Staging and rollback
- Promoting from staging in CI
- Migrating to trusted documents, which pushes a manifest from CI
- Deployments
- API keys
- CLI overview

DEPLOY AND OPERATE

Staging and rollback

A deployment that fails to build never goes live, so a broken file costs you a failed deployment rather than an outage. A deployment that builds and behaves wrongly is another matter. This guide sets up the three things you want before that day: a staging service to catch it, a rollback to undo it, and a way around GraphPilot if the problem is GraphPilot. What a deployment is and what happens while it runs is on deployments.

The running example is `shop-api` in production, with its origin at `https://api.example.com`, and a staging origin at `https://staging-api.example.com`. Both deploy from the same repository, where `schema.graphql` and `gpilot.toml` are versioned next to the application.

What you need

- The `gpilot` CLI, logged in, and access to the service's **Deployments** in the portal.
- A plan that includes more than one service, for the staging service. See pricing.
- Your configuration files in version control, unless `gpilot.toml` holds a JWT provider's shared `secret` (see below).

1. Create a staging service

There is no promotion between environments: a deployment builds from the files in front of it. A staging environment is therefore a second service, with its own subdomain and its own origin:

```
gpilot service create \
  --name "Shop API staging" \
  --subdomain shop-api-staging \
  --url https://staging-api.example.com \
  --environment staging
```

`--environment` is a label (`production` unless you set it) that tells the two services apart in lists. It changes nothing about how the service runs.

The two services share nothing: separate cache, separate keys, separate telemetry. That is what makes a staging deployment a real rehearsal, and it also means a purge on one never touches the other.

2. Deploy to staging first

Deploy the same files to both, staging first:

```
gpilot deploy --service shop-api-staging --yes
# verify against https://shop-api-staging.k7m2qp.graphpilot.cloud, then
gpilot deploy --service shop-api --yes
```

In CI, run the first line on pushes to a staging branch and the second on `main`; one organization API key can deploy both. See deploy from CI, and the recipe promoting from staging in CI for one workflow that deploys staging on every merge and production after an approval.

Verify against staging what the change was meant to do: the `gp-cache` status and `gp-cache-reason` of the requests it affects, and the deployment's findings in the portal. See findings.

3. Roll back to a deployment that worked

To go back, redeploy the version that worked.

In the portal, open **Deployments**, open the deployment that was good and choose **Redeploy**. The dialog starts with that deployment's schema and configuration selected, so the new deployment builds from exactly those two files. It runs on the service's runtime pin, or the stable version when the service has none, unless you pick another version under **Proxy version**. The version the good deployment ran on is marked **same as this deployment** there, as long as it has not been withdrawn.

From the CLI, deploy the previous files again:

```
git checkout <previous-commit> -- schema.graphql gpilot.toml
gpilot deploy --service shop-api --yes
```

Keep both files in version control either way: git tells you why a version changed, which the deployment list does not. The exception is a `gpilot.toml` that holds a JWT provider's shared `secret`, which the file carries as written: keep that file out of the repository, or use a provider with a public key.

Rolling back to an older runtime has a time limit. After some runtime versions, an older runtime is refused seven days after the service first went live on the newer one, with a message naming the version to use:

```
This service can no longer run runtime <version>. Deploy runtime <minimum> or later.
```

If no runtime the service can run is registered yet, the message ends with "No runtime it can run is registered yet; contact support." instead.

A service pinned to an older runtime within those seven days keeps what that runtime needs, so a rollback you pin in time is not cut off. A rollback of the schema and configuration on the current runtime is never affected.

If the bad version was wrong about caching rather than about serving, entries it stored keep their lifetime and partitioning after the rollback, because the cache survives a deployment. Purge them once the rollback is live:

```
gpilot purge --all --yes --service shop-api
```

4. Know how to take GraphPilot out of the path

If GraphPilot itself is the problem, the way out is where your clients point, not a setting.

- **On a custom domain**, point the DNS record back at your origin. Clients need no change, and the switch takes effect as your DNS TTL allows. Keep that TTL short.
- **On the service hostname** (`shop-api.k7m2qp.graphpilot.cloud`), clients have to be pointed at your origin directly. Make that endpoint configurable in your clients before you need it: it is the difference between a

configuration change and a release.

Either way your origin then serves the full, uncached load. Size for it.

Check that it works

After each deployment, the newest entry in the list is `completed` (the CLI writes statuses in lowercase, in the table and in `-o json`):

```
gpilot status --service shop-api-staging
gpilot status --service shop-api
```

After a rollback, send a request the bad version answered wrongly with `curl`. Once you have purged, the first answer is a `MISS` from your origin under the restored configuration, and the `gp-cache` status and `gp-cache-reason` match what the good version reported. A `HIT` before the purge can still be an entry the bad version stored.

Rehearse step 4 once on staging: switch a test client to the staging origin directly and back.

Pitfalls

- **A redeploy uses the service's runtime pin, else the stable version,** unless you choose another. If you suspect a GraphPilot change, pick what the good deployment ran on under **Proxy version** (marked **same as this deployment**).
- **Staging keeps its own cache.** A `HIT` on staging says nothing about production's entries, and staging's hit rate reflects staging's traffic.
- **An origin locked to GraphPilot refuses direct traffic.** If your origin accepts only GraphPilot's IP ranges or only signed requests, step 4 needs a way to relax that. Decide it in advance.
- **A long DNS TTL** turns a five-minute exit into an hour. Lower it before an incident, not during one.

Related

- Deployments
- Deploy from CI
- Promoting from staging in CI
- Custom domains
- Taking a service down

DEPLOY AND OPERATE

Edge locations

GraphPilot runs on Fastly's global edge network. A request is routed to a nearby location, handled there, and answered from there when the cache can answer it. There is nothing to configure: routing is handled for you and the set of locations may change over time.

The cache is per location

This is the part worth internalizing, because it shapes what hit rate you should expect.

Each edge location holds its own cache. A response stored in Frankfurt is not stored in Singapore. The first request for an operation at each location is a `MISS` and goes to your origin, however many times the same operation has been served elsewhere.

All of this holds unless you switch on shielding, which is off by default. With shielding, one location's cache stands between every other location and your origin.

Two consequences follow:

- **Your hit rate depends on how your traffic is distributed.** An operation requested a thousand times an hour from one region hits nearly always. The same operation requested once an hour from thirty regions may never hit at all, because each entry expires before that location sees a second request. Traffic concentration, not total volume, is what makes a cache work.
- **Origin load falls by less than your hit rate suggests at first.** Steady state is what matters; the warm-up is paid once per location per entry.

If an operation is expensive and rarely requested, a longer freshness lifetime is the lever: it gives each location's entry more chances to be reused. See how long an entry lives.

If your audience is spread across many regions, shielding is the other lever: it pays the warm-up once for the whole network instead of once per location.

> **Note**
> This is how every CDN works, and it is not a limitation specific to GraphPilot. It is stated here because a GraphQL cache is often evaluated by hit rate on a small sample of traffic, where the per-location warm-up dominates and the numbers look worse than production will be.

Shielding

Shielding routes cache misses through one edge location, the **shield**, before they reach your origin. Every location asks the same shield, so its cache is in effect shared by the whole network and your origin sees about one miss per entry per lifetime, however many locations request it. Each location still keeps its own copy, so a request is still answered from nearby once its location has the entry. It is optional and off by default, and used on paid plans. Name the shield location closest to your origin:

```
[origin]
shield = "eu-frankfurt"
```

How a shielded request flows, which requests go through the shield, what happens when it cannot be used, and what it costs in request units are on shielding.

Shield locations

The shield setting takes one of these values. Each one is named `<region>-<city>`. London and Sydney have more than one location, so their values carry a third part; any of them works. Pick the location closest to your origin. It is also where the shared copy of your cached responses is kept (see where data lives). A value that is not in this list refuses the deployment. When it looks like a typo of a known value, the message suggests up to three values to use instead.

Value	City	Country	Region
`af-capetown`	Cape Town	ZA	Africa (`af`)
`af-johannesburg`	Johannesburg	ZA	Africa (`af`)
`ap-auckland`	Auckland	NZ	Asia-Pacific (`ap`)
`ap-brisbane`	Brisbane	AU	Asia-Pacific (`ap`)
`ap-chennai`	Chennai	IN	Asia-Pacific (`ap`)
`ap-christchurch`	Christchurch	NZ	Asia-Pacific (`ap`)
`ap-delhi`	Delhi	IN	Asia-Pacific (`ap`)
`ap-hongkong`	Hong Kong	HK	Asia-Pacific (`ap`)
`ap-hyderabad`	Hyderabad	IN	Asia-Pacific (`ap`)
`ap-kolkata`	Kolkata	IN	Asia-Pacific (`ap`)
`ap-melbourne`	Melbourne	AU	Asia-Pacific (`ap`)
`ap-mumbai`	Mumbai	IN	Asia-Pacific (`ap`)
`ap-osaka`	Osaka	JP	Asia-Pacific (`ap`)
`ap-perth`	Perth	AU	Asia-Pacific (`ap`)
`ap-seoul`	Seoul	KR	Asia-Pacific (`ap`)
`ap-singapore`	Singapore	SG	Asia-Pacific (`ap`)
`ap-sydney-syd`	Sydney	AU	Asia-Pacific (`ap`)
`ap-sydney-wsi`	Sydney	AU	Asia-Pacific (`ap`)

Value	City	Country	Region
`ap-tokyo`	Tokyo	JP	Asia-Pacific (`ap`)
`ca-montreal`	Montreal	CA	Canada (`ca`)
`ca-toronto`	Toronto	CA	Canada (`ca`)
`eu-amsterdam`	Amsterdam	NL	Europe (`eu`)
`eu-brussels`	Brussels	BE	Europe (`eu`)
`eu-copenhagen`	Copenhagen	DK	Europe (`eu`)
`eu-dublin`	Dublin	IE	Europe (`eu`)
`eu-frankfurt`	Frankfurt	DE	Europe (`eu`)
`eu-helsinki`	Helsinki	FI	Europe (`eu`)
`eu-lisbon`	Lisbon	PT	Europe (`eu`)
`eu-london-lcy`	London	GB	Europe (`eu`)
`eu-london-lhr`	London	GB	Europe (`eu`)
`eu-london-lon`	London	GB	Europe (`eu`)
`eu-madrid`	Madrid	ES	Europe (`eu`)
`eu-manchester`	Manchester	GB	Europe (`eu`)
`eu-marseille`	Marseille	FR	Europe (`eu`)
`eu-milan`	Milan	IT	Europe (`eu`)
`eu-oslo`	Oslo	NO	Europe (`eu`)
`eu-paris`	Paris	FR	Europe (`eu`)
`eu-sofia`	Sofia	BG	Europe (`eu`)
`eu-stockholm`	Stockholm	SE	Europe (`eu`)
`eu-vienna`	Vienna	AT	Europe (`eu`)
`me-fujairah`	Fujairah	AE	Middle East (`me`)
`sa-bogota`	Bogota	CO	South America (`sa`)
`sa-riodejaneiro`	Rio de Janeiro	BR	South America (`sa`)
`sa-saopaulo`	Sao Paulo	BR	South America (`sa`)
`us-ashburn`	Ashburn	US	United States (`us`)

Value	City	Country	Region
`us-atlanta`	Atlanta	US	United States (`us`)
`us-boston`	Boston	US	United States (`us`)
`us-chicago`	Chicago	US	United States (`us`)
`us-dallas`	Dallas	US	United States (`us`)
`us-denver`	Denver	US	United States (`us`)
`us-houston`	Houston	US	United States (`us`)
`us-losangeles`	Los Angeles	US	United States (`us`)
`us-miami`	Miami	US	United States (`us`)
`us-minneapolis`	Minneapolis	US	United States (`us`)
`us-newyork`	New York City	US	United States (`us`)
`us-newark`	Newark	US	United States (`us`)
`us-portland`	Portland	US	United States (`us`)
`us-sanjose`	San Jose	US	United States (`us`)
`us-seattle`	Seattle	US	United States (`us`)

Purging is global

Invalidation does not have this property. A purge, whether it came from a mutation, the API or the CLI, removes the entry at **every** location, not only the one that served the request that triggered it.

That asymmetry is deliberate and is the point of surrogate keys: entries are cheap to create per location and must be removable everywhere at once, or invalidation could not be relied on.

With shielding, a purge clears the entry at the shield too, because purges are global. A purge through the API, the CLI or the portal on a shielded service is sent a second time shortly after the first; see mutations and purges.

Where data lives

This section covers data location and data residency. Two different things are stored in two different places, and the distinction matters for a data protection (GDPR) review.

Cached responses

A cached response lives in Fastly's cache at the locations that served it, worldwide. It is stored as the bytes your origin produced, plus the metadata needed to serve it again: its lifetime, its surrogate keys, and the discriminators it is partitioned by.

The discriminator values themselves are not stored with the response, and neither is the credential the request carried.

There is no region pinning and no data-residency guarantee for cache entries. A response is held wherever it was requested from. If a class of data must not leave a region, do not make it cacheable: mark it `no-store` at your origin, or exclude it with a bypass.

With shielding, an entry is also stored at the shield location, whichever location requested it. A shield in the EU keeps that shared copy in the EU; the copies at the requesting locations stay where they are. Take the shield location into account in a data protection review.

Telemetry and logs

Request telemetry is stored under the rules on what Insights needs. A request that was not stored is still counted for usage, but no record of it is kept, so there is nothing to look up in Insights and nothing for `purgeByRequest` to resolve.

When it is stored, request telemetry is kept in the **EU** for at most **90 days**. How far back you can look in Insights depends on your plan: 1 day on Glider (Free), 7 days on Jet, 30 days on Airliner and 90 days on Jumbo, after which a request is deleted. See what Insights needs. Once a request is no longer stored, it can no longer be resolved, which is also what bounds `purgeByRequest`.

What is recorded about each request is yours to configure, and the defaults are narrow: `authorization` and `cookie` are replaced by a keyed digest rather than stored, and only a handful of descriptive headers are kept. See `[telemetry]`.

The raw telemetry copy

Each recorded request and its log lines are also written to files in GraphPilot's raw telemetry store, held in Google Cloud Storage in the EU, so Insights can be rebuilt after a loss.

- **They hold the same record Insights does:** the request's telemetry row and its log lines, including the client IP address and the geo and network data looked up from it, the captured headers and the GraphQL document. `telemetry.client_ip` is applied at the edge before anything is written, so on a `"truncated"` or `"none"` service the files never hold the full address either. A request that is not recorded is not in them.
- **Each file is kept for at most 90 days** after it was written, then deleted automatically. The files are protected against deletion for part of that period, so a mistake or an incident cannot remove the copy.
- **Deleting a service, an organization or your account does not delete these files early.** The service's Insights data is removed from the Insights database within minutes; its files stay until their 90 days run out. This is a deliberate exception: deleting them earlier would mean deleting the copy that protects everyone else's Insights data.
- **They are used for nothing else.** They are not readable through the portal, the CLI or the GraphPilot API, and are read only to fill and rebuild Insights. A rebuild deletes again the data of every service, organization and account that was deleted in the meantime.
- **There is no further copy of the files.** They are not included in any other backup.

- **The Insights database has backups of its own**, taken daily by its provider and kept for 30 days. Data removed from the database stays in those backups until they expire.

What a deletion keeps

Deleting a service, an organization or an account removes its telemetry, logs and stored files within minutes, once each service is off the edge. These records are kept on purpose:

- **The request counts usage is billed from**, and the billing and invoice history, which GraphPilot is required to keep.
- **The record of the terms** an organization accepted: which version, when, and by whom, with the email address and name the accepting user's login had at that moment. It is kept as evidence of the contract, also once that login is deleted.
- **The raw telemetry copy**, for at most 90 days after each file was written, used only to rebuild Insights data.
- **Database backups**, overwritten within 35 days, and **backups of the Insights database**, kept for 30 days.
- For an account: **the record of your calls to the GraphPilot API** from the portal and the CLI, for at most 90 days.

Resetting a service's history keeps the same records, and also the measurements alerting compares against, because the service goes on running.

Latency

Serving from a nearby location is most of where the latency saving comes from, and how much you gain depends on the distance between your users and your origin rather than on the cache alone. Performance and benchmark methodology explains what is measured, how, and how to reproduce it.

Custom domains

A custom domain is served from the same network as the service hostname, with a certificate issued for it. Attaching one changes nothing about routing, caching or invalidation.

DEPLOY AND OPERATE

Shielding

Plans Jet (Starter) and above

Without shielding, each edge location holds its own cache: the first request for an entry at every location is a miss that goes to your origin. Shielding puts one edge location, the **shield**, between every other location and your origin. A miss anywhere asks the shield first, so the shield's cache is in effect shared by the whole network and your origin sees about one miss per entry per lifetime, however many locations request it.

Shielding is off by default. It is worth switching on when your audience is spread across many regions, or when your origin is expensive to call; it is less useful when most traffic arrives at one or two locations anyway.

Switch it on

Name the shield location in the `[origin]` table of `gpilot.toml` and deploy:

```
[origin]
shield = "eu-frankfurt"
```

- **Pick the location closest to your origin.** The shield is the only location that connects to your origin, so its distance to the origin is what every shielded miss pays. The values are in the shield location table; a value that is not in it refuses the deployment, and the refusal suggests the closest values when there are any.
- **Your plan decides whether the setting is used.** Shielding is included in every paid plan. On Glider (Free) a configured shield still deploys, with a warning on `origin.shield` that names the location and says it is inactive on this plan and turns on by itself after an upgrade, with no redeploy needed. The edge checks the plan on every request, so an upgrade switches the shield on and a downgrade to Glider switches it off without a new deployment. While it is off, every miss goes to your origin directly.
- **The portal shows it.** The service's **Config** view names the shield's city and country next to `origin.shield`, and says whether the shield is active on the plan. See schema and config findings.

To switch shielding off, remove `shield` and deploy again.

Example: an origin in Frankfurt

`shop-api` runs in a Frankfurt data center, and a third of its traffic comes from North America and Asia. Without a shield, each of those edge locations fills its own cache from Frankfurt. `eu-frankfurt` is the shield location next to the origin:

gpilot.toml

```
[origin]
shield = "eu-frankfurt"
```

A value from another vocabulary is refused before anything is deployed. The CDN's own code for the same location, `frankfurt-de`, fails the deployment with:

```
unknown shield location "frankfurt-de", did you mean "eu-frankfurt"?
```

After the deployment, a miss at an edge location in Chicago asks the shield first. The request's **Overview** tab in Insights shows the **Shield** set with Frankfurt as the location: the first request for an entry reads **Result** `MISS` there, and the same request from Singapore a moment later reads `HIT`, because the shield already holds the entry and your origin is not called. The **Shield** section of the **Cache hit** card sums this up for the time range; its

Offload is the share of requests your origin never saw. Usage rises with it: a miss through the shield counts at two edge locations (see request units).

How a shielded request flows

1. A request arrives at the nearest edge location and is looked up in that location's cache. A hit is answered there, exactly as without shielding.
2. On a miss, the edge location sends the request to the shield instead of to your origin. It sends the request as the client sent it, signed so the shield can tell it came from a GraphPilot edge location and which client address it was for.
3. The shield looks the entry up in its own cache, under the same cache key and the same partition. On a hit there, it answers and your origin is not contacted.
4. On a miss at the shield, the shield calls your origin with the same TLS, request signing, timeouts and retries as any other origin request, and stores the response.
5. The response travels back to the edge location, which stores its own copy and answers the client. Later requests at that location are answered from its own cache.

The edge location stores the shield's response with what is left of its lifetime: the shield's age is subtracted from the max-age, so both copies expire together. When nothing is left, the response is served but not stored at the edge location (`CACHE_SKIPPED_SHIELD_ENTRY_EXHAUSTED`). A response the shield answered from a stale copy, or without consulting its cache, is served but not stored at the edge location either (`CACHE_SKIPPED_SHIELD_RESPONSE_NOT_STORABLE`); see stale windows with shielding.

Misses from many locations meet again at the shield, so request collapsing works across locations too.

Which requests go through the shield

Only a cache miss goes through the shield, and only when the request is one of these:

Request	Through the shield
A GraphQL query	Yes
A GraphQL mutation	Yes, see mutations and purges
A GraphQL request carrying only a persisted query hash	Yes; the shield resolves the hash
A GraphQL document the edge location cannot parse	Yes; the shield refuses it the same way
A plain HTTP `GET` or `HEAD`	Yes
A GraphQL subscription, any other plain HTTP method, a WebSocket upgrade	No, straight to your origin
A request that never consults the cache (a `PASS` with a `CACHE_PASS_*` code, for example one matching `cache.policy.bypass`)	No, straight to your origin

Request	Through the shield
A request served in lean mode	No, straight to your origin

A background refresh of a stale entry at an edge location also goes through the shield.

Mutations and purges

A mutation goes through the shield, so its automatic purge runs at the shield first and at the edge location after it, on the way back. Purging in that order keeps the edge location from refilling from a shield copy that is about to be purged.

A purge through the API, the CLI or the portal is sent twice on a shielded service, so an edge location purged before the shield cannot refill from the shield's copy. A purge clears the entry at the shield too, because purges are global.

When the shield cannot be used

The edge location sends the shield exactly one request and does not retry it; retries run between the shield and your origin. What happens when the hop fails depends on the request:

Failure	A query, `GET` or `HEAD`	A mutation	A request whose kind the edge cannot see (only a hash, or a document it cannot parse)
The shield could not be reached before the request was sent (for example DNS, connect or TLS)	Sent to your origin by the edge location (`FALLBACK`)	`502`, nothing sent to your origin (`UNAVAILABLE`)	Sent to your origin by the edge location (`FALLBACK`)
The send failed in a way that leaves unclear whether the shield received it, or the answer did not come from the shield	Sent to your origin by the edge location (`FALLBACK`)	`502` (`UNAVAILABLE`)	`502` (`UNAVAILABLE`)
No usable answer within the time budget, or the connection broke off	`504` (`TIMEOUT`)	`504` (`TIMEOUT`)	`504` (`TIMEOUT`)
The shield answered with a body the edge location could not decode	`502`	`502`	`502`
The shield answered, including with an error status from your origin	Passed through as it is	Passed through as it is	Passed through as it is

A mutation never falls back to your origin, because the edge location sending it itself could run it twice and would purge this location before the shield. A fallback is logged as `SHIELD_UNAVAILABLE_FALLBACK`, a refused

mutation as `SHIELD_UNAVAILABLE_NO_FALLBACK`, and a timeout answers `GATEWAY_TIMEOUT`; see codes. The `reason` on a `SHIELD_UNAVAILABLE_FALLBACK` or `SHIELD_UNAVAILABLE_NO_FALLBACK` line names the step that failed; quote it to support.

While the shield cannot be reached, a service keeps working on queries at the cost of the per-location warm-up, as without shielding.

The time budget. The edge waits for the shield's whole origin sequence (every attempt your `[origin.retry]` policy allows, and the gaps between them) plus a fixed 5 seconds for the hop, never more than the 120-second budget. An unset `first_byte_timeout_ms` counts as 15000 ms here. A configuration whose total exceeds the 120-second budget is refused by the deployment build, naming `origin.shield`. See connection to the origin.

Headers

- **What your clients see does not change.** The response headers come from the edge location that answered the client: `gp-cache` says whether that location had the entry, so a response the shield supplied reads `MISS` there. See cache statuses.
- **The hop headers stay inside GraphPilot.** The headers the shield and the edge location exchange never reach your client or your origin, and the same names are removed if your origin sends them (see reserved names).
- **Your origin sees the client, not the edge location.** With `origin.forward client ip`, `gp-client-ip` carries the address the first edge location saw, never a GraphPilot address. See headers.
- **Your origin is called from the shield.** Keep the whole GraphPilot address set in your origin allowlist; requests that do not go through the shield, and fallbacks, still come from other locations.

Insights and logs

A shielded request is one request in Insights, with the hop recorded on it:

- **The request detail panel** shows a **Shield** set on its **Overview** tab: the shield location with its city and country, how the hop ended (**Result**) and, when the shield called your origin, that call's status, latency and retries; the **Diagnostics** set names the shield server that answered. On the **Cache** tab, a request the shield answered without storing shows the shield's own reason, marked **Decided at the shield**. When the shield called your origin, the **GraphQL** tab shows the document your origin received as the shield recorded it; when the shield answered from its cache, the tab says nothing was sent to the origin.
- **The list** can be filtered by **Shield result** and show it as a column, on every plan. A request the shield sent to your origin (`MISS` or `PASS`) is marked when GraphPilot changed its document, and the **Rewritten by GraphPilot** filter finds it. Shielded requests recorded before runtime 1.6.0 stay unmarked.
- **GraphQL errors** your origin answered the shield with are recorded on the request, as on an unshielded one: the **GraphQL errors** column and preset, the error groups and the alert rule **GraphQL errors in your responses** count them, including errors removed before the client saw them. A request the shield answered from its cache (`HIT` or `STALE`) records none. Shielded requests recorded by runtime 1.8.0 or earlier carry no GraphQL errors.

- **Origin outcome** reads **Called via shield** when the shield called your origin, **Not called (shield cache)** when it answered from its cache, and **Shield failed** when it gave no usable answer, so a shielded request is never counted as served without your origin when it was not.
- **The Cache hit card** has a **Shield** section on a service that had shielded traffic in the time range: **Hit** (the share of shielded requests the shield answered from its cache, fresh or stale), **Offload** (the share of all requests the shield answered, so your origin never saw them), **Fallback** (the share of shielded requests where the shield could not be used: `FALLBACK`, `TIMEOUT` or `UNAVAILABLE`) and **Shielded** (how many requests went to the shield). A click on **Hit** or **Fallback** filters the list to those requests. A falling **Hit** with a low **Fallback** means the shield is reachable but misses; a rising **Fallback** means the shield is not being used.

In the log fields, `shield_pop` names the shield location and `shield_result` how the hop ended (`HIT`, `MISS`, `STALE`, `PASS`, `FALLBACK`, `TIMEOUT` or `UNAVAILABLE`). `shield_node` names the server within the shield that answered, so it is empty when no answer came from the shield (`FALLBACK`, `TIMEOUT`, `UNAVAILABLE`). The fields are empty on a request that did not go to the shield. See log fields.

Request units

Usage is counted in request units, the way the CDN bills: per request at each edge location that ran it, by the size of the body that location produced. The shield is an edge location too, so a request sent through the shield usually uses **twice** the request units of the same request without a shield:

- A 2 KB response through the shield is 2 request units, a 25 KB one 6 (two locations, three started 10 KiB each).
- A request the shield ran counts at both locations, whatever it answered, and also when the hop timed out.
- A request that fell back before the shield ran it counts at the edge location only.
- A background refresh through the shield is a request of its own, counted at the shield.
- A hit at the edge location, and any request that does not go through the shield, counts at one location as before.

Insights still shows a shielded request as one request, so the request units for a period are higher than the request count Insights shows. The full table is on quotas and lean mode.

Where the shared copy lives

An entry is also stored at the shield location, whichever location requested it. A shield in the EU keeps that shared copy in the EU; the copies at the requesting locations stay where they are. Take the shield location into account in a data protection review. See where data lives.

> **Note**
>
> Shielding changes where misses go, not what is cached. Cache keys, partitions, lifetimes and surrogate keys are the same at the shield and at the edge location.

Related

- Edge locations: why the cache is per location, and the shield location table.
- Connection to the origin: timeouts, retries and fallback in detail.
- Configuration reference: `origin.shield`.
- Codes: the `SHIELD_*` codes and `shield_result` values.

DEPLOY AND OPERATE

Quotas and lean mode

Four things are worth knowing before you go live: the request quota your plan includes, fair use on Glider (Free), the per-plan ceiling on how long the edge may wait for your origin, and how many services and custom domains your plan includes.

> **Note**
> The full set of figures each plan includes lives on the pricing page, because they are commercial rather than technical. This page is about **behavior**: what changes when a limit is reached, with the request allowance and its overage price below.

The request quota

The request quota is enforced on Glider (Free) only. Usage is counted in request units, per billing account, over a monthly window that starts when the account moves to Glider, at sign-up or after a paid plan ends, and repeats monthly from there. One billing account can fund several organizations, and their request units count together. Paid plans have no quota: each plan includes a number of request units per monthly billing period, and request units above it are billed as overage, per started million.

Plan	Included request units a month	Above the allowance
Glider (Free)	100K	No overage. Every service the billing account funds is switched to **lean mode** until the window resets or the plan changes. Glider is offered under fair use.
Jet (Starter)	500K	€6.00 per started 1M request units
Airliner (Business)	5M	€5.00 per started 1M request units
Jumbo (Enterprise)	Per contract	Per contract

Glider (Free) is meant for trying GraphPilot out and for small projects, not for production traffic: it comes with no support commitment, its services run lean once the allowance is used up, and it is offered under fair use.

Emails about the request quota

On Glider (Free), everyone whose role is **Admin** in an organization the billing account funds gets an email at each of these points:

When	What the email says
The account has used **80%** of the request units included in the current window	The share used, the included request units, the day the window resets, and what lean mode would withdraw. Not sent when the allowance is already used up by the time the email goes out: the lean mode email covers it.
The services switch to **lean mode** because the allowance is used up	The included request units, what lean mode withdraws, and the day full serving returns.
Full serving returns, because the window reset or the plan changed	That edge caching is on again.

Each email goes out once for each crossing: the warning once per window, the other two once per switch. None goes out while the account is locked, or once the deletion of the account has started. Lean mode for an overdue payment sends none of these emails: the overdue payment emails apply then.

Prices exclude VAT. On a paid plan, going over the allowance throttles nothing and switches nothing off: the request units above it are billed. A renewal payment that fails is a different matter: the services run in lean mode until it arrives, and the account is locked if it is still unpaid 14 days after the payment reminder. See overdue payments. “Per started million” means the request units above the allowance are rounded up to whole millions: on Jet, 500,001 request units in a period are billed as 1M of overage, and so are 1,500,000; 1,500,001 are billed as 2M.

Request units

Requests are counted **per request at each point of presence, and by response size.** Every edge location that runs a request counts it once, weighted by the response body that location produced:

> sum, over the **edge locations a request passed through**, of the **size units of the body each one produced** = **request units**

Without shielding that is one location, so a request uses the size units of its response. With shielding, the bodies the shield returns to the edge location and the edge location sends to the client are usually the same size, so a request through the shield uses twice the size units of its response.

Edge locations. A request that goes through the shield counts at two edge locations, the one that received it and the shield. Every other request counts at one: a cache hit, a request on a service without shielding, and one that never reaches the shield. A background refresh through the shield, after a stale copy was served, is a request of its own, counted at the shield.

Size units. One per started 10 KiB (10,240 bytes) of the uncompressed response body, headers excluded, at least one. At the edge location that received the request, the size is the body GraphPilot sends to the client; at the shield, the body the shield returns to the edge location, which the edge location asks for uncompressed. Both are measured before any compression on the way to the client, which is also how the CDN bills a compressed response, so the same query costs the same whatever the client's `Accept-Encoding`. A response without a body (a `HEAD` request, a `204` or a `304`) is one size unit.

Request	Response body	Request units
Answered from the edge location's cache	2 KB	1 × 1 = 1
A cache miss on a service without shielding	25 KB	1 × 3 = 3
A cache miss sent through the shield	25 KB	2 × 3 = 6
A `HEAD` request, a `204` or a `304`, sent through the shield	none	2 × 1 = 2
A background refresh through the shield of a 25 KB entry	25 KB	1 × 3 = 3

Shielding is available on paid plans only, so on Glider (Free) every request counts at one edge location, by size. Glider's quota is counted in request units too.

Every request your service runs counts, whoever sent it and however it ends; the one exception is a purge through GraphPilot (below). That includes the requests the edge refuses before your origin sees them: a credential the service does not accept, such as a token whose signature does not verify (`401`); an operation that is not on the service's allowlist (`403`); a request over one of the service's request limits (`413` or `400`); a request from a client the service does not accept (`403`). Flood traffic counts the same way, request by request, whether or not the edge refuses it. A refused request counts at the edge location that refused it, with the size units of its error response, usually one. Edge codes lists every refusal.

Purges through GraphPilot are free. A call to the purge endpoint that fails authentication, answered `401`, counts as one request unit, like any other request someone sends to your service.

Two figures are **not** counted in request units:

- **Insights** counts every request a client sent once, whatever its size or route, and shows no background refreshes. The request units for a period are therefore higher than the request count Insights shows whenever responses are larger than 10 KiB or requests go through the shield.
- **Logged requests**, the Insights meter of the paid plans, count each logged request once. Glider (Free) has no such meter: recording stops with lean mode when the request units are used up.

To estimate the request units of a traffic profile, use the pricing calculator.

Example: a month of traffic

`shop-api` serves 2,000,000 requests a month with shielding on. 70% are answered from the edge location's cache, with a 4 KB body; the other 30% are misses that go through the shield, with a 25 KB body:

Requests	Count	Request units each	Request units
Cache hits at the edge location, 4 KB	1,400,000	1 × 1 = 1	1,400,000
Misses through the shield, 25 KB	600,000	2 × 3 = 6	3,600,000
Total	**2,000,000**		**5,000,000**

Insights shows 2M requests for the month; usage shows 5M request units. That fits within Airliner's 5M exactly, and on Jet it is 4.5M request units above the allowance, billed as 5M of overage because overage is counted per started million. Background refreshes through the shield and refused requests add to the total, so leave headroom. Without shielding, the same misses use 3 request units each, 3.2M request units in total.

Lean mode

Lean mode removes **the paid feature set, which today means edge caching and stored request telemetry, and nothing else**.

Still runs	Withdrawn
Authentication and claim resolution	Cache lookups: every request goes to your origin
Policy expressions and vary headers	Cache writes: no response is stored
Request signing	The `gp-cache` and `gp-cache-reason` headers (a request that bypasses the cache still reports `PASS`)
Usage counting: request units are still counted and billed against the allowance	Stored request telemetry and logs: requests and log lines are not recorded (see below)
Invalidation	

Four consequences follow, and all four are deliberate.

Your API keeps working. A quota is not a kill switch. Requests are still authenticated, still signed and still forwarded. What you lose is the speed and the origin offload, which is what you were paying for. The one thing that can stop a Glider (Free) account is fair use, far above the request quota.

Your origin takes the full load. Every request reaches it, including the ones a warm cache had been absorbing. If your origin was sized behind a cache, this is the number to plan for.

Responses carry no `gp-cache` header. The cache steps aside entirely, so there is no status to report. A response through GraphPilot without `gp-cache` , while `response.cache_headers` is on, is the sign that the service runs lean. See troubleshooting.

Nothing is recorded while lean. See the next section.

Nothing is recorded in lean mode

Requests served while the billing account runs lean are **not recorded**: no request telemetry and no log lines are stored for them, for any service the account funds. They are not held back or hidden. They are never written, so:

- Insights, the service Dashboard, the **Purges** page and `gpilot requests` show no requests from the lean window, and `purgeByRequest` cannot resolve one.
- **An upgrade does not bring them back.** Recording resumes for requests served after the plan changes (or the monthly window resets), and the lean window stays empty in every request view.
- **They are still counted.** The usage figure on the billing page and in the API includes the request units of every request served in lean mode, because counting does not depend on recording. That is the number your allowance and any invoice are based on.

If you need a record of every request, upgrade before the allowance runs out. The billing page shows how much of it is used.

The portal says so wherever it matters: a **Paid features are paused** notice below the toolbar of every organization page, with **Upgrade** (or **Contact support** if GraphPilot paused the account), and empty states on Insights and **Purges** that name lean mode as the cause. See paid features are paused.

Invalidation keeps running

Purging is not withdrawn in lean mode, and that is not an oversight. A mutation served during a lean window still invalidates entries written **before** it. If it did not, those entries would come back live and stale the moment normal service resumed, and nothing would have recorded that they were wrong.

Purging a key nothing holds removes nothing, so this costs one call per mutation carrying surrogate keys, on exactly the services that are not generating most of the traffic.

Fair use

Glider (Free) is meant for side projects and for trying GraphPilot out. Lean mode keeps a Glider account serving above its request quota, but Glider is offered under fair use: if an account's traffic goes far beyond what Glider is meant for, GraphPilot may suspend its services and lock the account until the start of the next month, or until it moves to a paid plan. Nothing is deleted. Paid plans have no such limit: their usage above the plan's allowance is billed as overage.

- **Per billing account**, across every organization and service it funds, like the request quota.
- **Per month (UTC).** The month starts on the 1st at 00:00 UTC, whatever day the request quota's window starts on. For an account that moved to Glider from a paid plan during the month, usage counts from the move, so nothing used on the paid plan counts, and a warning or a lock from before the move no longer applies.
- **Lean mode does not stop the count.** Requests served in lean mode count toward fair use like any other, whether or not they are recorded.

What happens:

Step	What happens
Warning	One warning email a month, while the account is not locked, to the billing contacts of every organization the account funds, and a notice on every page of those organizations. Nothing changes in how the services are served. A warning usually comes before a lock; traffic that rises very quickly can lead to a lock without one.
Lock	The account is **locked**: every service it funds is suspended, the same way as for an overdue payment. The API behind them is no longer served, nothing is deleted (services, configuration, domains and deployments are kept), and the GraphPilot API refuses every change outside billing with `ACCOUNT_LOCKED` and `extensions.reason` `FREE_TIER`. A lock email names the day the lock lifts.

The lock lifts at the start of the next month, or at once on a paid plan. The services are then redeployed, usually within minutes, and an email says the account is active again:

- **On its own at the start of the next month (UTC)**, within its first hour.
- **At once when the account moves to a paid plan.** Billing stays open while the account is locked, and a paid plan can be chosen on the billing page, once no invoice is outstanding: while a payment is overdue, the invoice has to be paid first.

More rules:

- **The payment lock and the fair-use lock never lift each other.** An account has one lock at a time, and a locked account is not locked again under fair use. When an account on Glider pays off an overdue payment, it is unlocked as usual, its services are redeployed, and an email says it is active again. If its traffic this month still goes beyond fair use, it is locked under fair use again, with its own lock email. The start of a month does not lift a lock for an overdue payment.
- **A payment reminder keeps running under a fair-use lock.** If a payment reminder is open when the account is locked under fair use, and its deadline passes unpaid, the account is then locked for the overdue payment instead. The start of the next month lifts only a fair-use lock: an open reminder keeps its deadline, and an unpaid invoice stays owed.
- **A fair-use lock does not delay a payment reminder.** If a payment becomes overdue while the account is already locked under fair use, the payment reminder goes out within the hour all the same, and its 14 days run from that reminder, not from the end of the fair-use lock. The account stays locked under fair use until the deadline passes; then the lock for the overdue payment replaces it, as above. Paying before the deadline ends the reminder and leaves the fair-use lock as it is.
- **A lock by support replaces a fair-use lock**, and so does a lock for an overdue payment. Once that lock lifts within the month, the account is locked under fair use again if its traffic this month still goes beyond it. If support lifts a fair-use lock instead, it stays lifted for the rest of that month.

The portal shows both states below the toolbar: see fair use.

Request telemetry

What is stored, up to which allowance and for how long, is on what Insights needs. A service can also store less on its own, with `telemetry.record`; requests it leaves out are still counted against the allowance. In lean mode nothing is stored, and that window is not filled in later: see nothing is recorded in lean mode. Without stored

telemetry, the Insights request views, the **Purges** page and `gpilot requests` have nothing to show, and `purgeByRequest` has no request to resolve. In the portal, an empty request log and an empty purge history say that nothing is recorded in lean mode rather than that nothing matched, and the usage figures keep counting.

Origin timeout ceilings

How long the edge may wait for your origin's first response byte depends on your plan. A deployment configuring more than its plan allows is **refused**, not quietly reduced. The table and the reasoning are on what your plan allows.

Services and custom domains

- **Services are capped per plan**, counted across every organization your billing account funds. Creating a service past the cap is refused with `SERVICE_QUOTA_EXCEEDED` ("Your plan's service limit is reached").
- **Custom domains are available from Jet (Starter)** and are also counted across the whole billing account, not per service. Adding one on Glider (Free) is refused with `DOMAIN_NOT_IN_PLAN`, and adding one past the plan's allowance with `DOMAIN_QUOTA_EXCEEDED`.
- **A billing account funds at most 3 organizations**, on every plan. Creating a fourth is refused with `ORGANIZATION_QUOTA_EXCEEDED` before anything is created. Deleting an organization frees its place; to fund more, contact support. See organizations per billing account.

The numbers for each plan are on the pricing page. Contact support if you need more.

What has no limit

- **No rate limit and no request cap at the edge.** GraphPilot imposes no per-second limit and no cap on requests in flight. See limits. This is a technical statement, not a promise of unlimited use: Glider's allowance still applies, Glider (Free) is offered under fair use, and GraphPilot can lock an account that misuses the service, for example with traffic far above its plan's allowance, traffic generated without a real use case, or attacks sent through the platform.
- **No cap on schemas or on surrogate keys per service**, beyond the 16,000 bytes of keys per cache entry.
- **No cap on purges.** The purge API bounds one call at 1000 keys; it does not bound how many calls you make.

Seeing where you stand

Usage is in the portal on the organization's **Billing** page. Because the quota is counted per billing account, that is where it is shown, not in an organization's settings. See billing and usage.

Related

- Connection to the origin
- Billing and usage
- Custom domains

- Troubleshooting

DEPLOY AND OPERATE

Go-live checklist

GraphPilot handles the parts of running a service you would expect. This is the short list of points that are easy to overlook and expensive to discover in production.

Secure the connection to your origin

Your origin is still reachable from the internet once GraphPilot is in front of it, and its address is not a secret.

- [ ] **Verify the signature**

  GraphPilot signs every request, so verifying the signature on your origin is what proves a request came through GraphPilot. Your origin needs the signing key. It is shown once, when the service is created: in the portal, in the dialog that follows creating the service there, and from the CLI, in the **Signing key** row `gpilot service create` prints. Neither shows it again. If you did not keep it, rotate it in the portal to get a new one. Reject anything that fails verification, and always compute the signature over the **raw** request body, before any framework parses or re-serializes it. Check your implementation against the published test vector.

- [ ] **Register the origin as https**

  GraphPilot speaks TLS to your origin when the origin URL is `https://`, and does not when it is `http://`. There is no separate switch. Without a port in the URL, GraphPilot connects to port 443 for `https://` and port 80 for `http://`. See trust and transport.

- [ ] **Allowlist the edge at your firewall**

  Verifying the signature happens after the request has arrived. Refusing everything else at the network edge is the cheaper first line, and the current ranges are in the portal and on the GraphPilot API. The set changes, so read it again rather than trusting a copy: see the origin allowlist.

- [ ] **Keep your clock in sync**

  Run NTP on your origin. The signature is time-bound, so clock drift makes valid requests fail and widens the replay window. See replay protection.

- [ ] **Pin every JWT provider to your issuer and audience**

  A signature only proves which key signed a token. Set `allowed_issuers` and `allowed_audiences` on every JWT provider, to the `iss` and `aud` your identity provider puts in its tokens; otherwise a token another tenant of a shared identity platform minted verifies as one of your callers. A provider with `jwks_url`, `public_key` or an asymmetric algorithm does not deploy without `allowed_issuers`; a missing `allowed_audiences` draws an `auth_provider` warning on every deployment. A token without `exp` is refused unless the provider sets `allow_missing_exp`. See configuring a provider.

- [ ] **Serve GraphQL only on the configured path**

  GraphPilot reads a request as GraphQL on the paths in `request.graphql_paths`, on paths below them and in any letter case. Every other path is plain HTTP and skips the GraphQL checks. An operation sent there is refused with `404`, but an upload or a subscription is not recognized there, so an origin that also answers GraphQL elsewhere can still be reached past the checks that way. Keep REST routes out from under the GraphQL path too: a body there that is not an operation is refused with `415`.

- [ ] **Decide what happens to an unverifiable credential**

  By default a token that does not verify is refused at the edge and never reaches you. If you would rather it arrive as anonymous during a key rotation, that is a per-reason setting. See rejection.

Make sure it actually caches

- [ ] **Confirm a second identical request is a HIT**

  Not a smoke test, the whole point. Read `gp-cache` on both responses. A `PASS` means nothing was stored, and its `gp-cache-reason` names the rule that refused. See troubleshooting.

- [ ] **Stop setting cookies on read requests**

  A response carrying `Set-Cookie` is never stored. Frameworks that touch the session on every request do this by default, and it switches caching off for everything it touches. This is the single most common reason a first deployment shows no hits. See what refuses storing.

- [ ] **Say how long each response stays fresh**

  Nothing is cached unless your origin's `Cache-Control` or, for a GraphQL API, your schema's `@cacheControl` says so. A service that says neither caches nothing. See when a response is cached.

- [ ] **Deploy the schema before the origin**

  A query that selects a field your deployed schema does not declare is never cached (`CACHE_SKIPPED_UNRESOLVED_SELECTION`). Put the GraphPilot deploy ahead of your origin's in the release, so new fields are known before clients ask for them. See deploy the schema before the origin.

- [ ] **Decide whether you need shielding**

  Each edge location keeps its own cache, so a `HIT` in one region says nothing about the next: the first request for an entry at every location still reaches your origin. If your callers are spread across many regions, or your origin is expensive to call, switch on shielding and name a shield location close to your origin. Every other location then asks the shield instead of your origin, and your origin sees about one miss per entry for the whole network. Shielding is available on paid plans. See the cache is per location.

Get personalization right before you go live

> **Danger**
>
> This is the section where a mistake means one caller seeing another caller's data. GraphPilot fails closed wherever it can, but nothing can check that the value your expression returns identifies one person rather than a group.

- [ ] **Decide who shares each answer**

  Ask of every cached answer: who else gets exactly the same one? Data that is the same for a whole organization, or for one organization and role, belongs in a named scope such as `ORGANIZATION`, backed by a `[[cache.policy.vary]]` entry. It stores one entry per group and keeps your hit rate, where a per-person entry misses for every member. For a plain HTTP origin, a vary rule on the organization claim partitions the whole service the same way. Data that is the same for everyone stays `PUBLIC`, which is also the default: `@cacheControl(scope:)` is `PUBLIC` unless you write a scope, even for a request with a credential, so give every type or field whose answer depends on the caller a scope (a field hint without one keeps its type's scope). Only what really differs per person gets `scope: PRIVATE` (or `Cache-Control: private` from your origin), the last rung, which relies on the expression in the next card. See who shares an entry.

- [ ] **Read your private expression as if it were an access rule**

  `cache.policy.private` decides who shares a cache entry. `auth:idp.sub` and `cookie:session_id` identify one caller; an organization id, a customer id, a role or a plan name identify a group, and everyone in that group then shares one entry. A group belongs in a named scope or a vary rule instead. See private.

- [ ] **Decide how fast a revoked permission must take effect**

  A cache hit does not consult your origin, so a withdrawn permission is not reflected until the entry expires or is purged. If that window matters, shorten the lifetime for those responses or purge on revocation. See revocation.

Know what your writes invalidate

- [ ] **Return the affected objects from your mutations**

  Automatic invalidation follows what a mutation's response actually contained. A mutation declared as returning `Boolean!` invalidates nothing at all. See automatic invalidation.

- [ ] **Plan for creates separately**

  A newly created object does not appear in a cached list that was not empty, because nothing cached contained it. Either return the parent whose key covers the list, or name a key on the mutation's `surrogate-key` response header. See the three ways around it.

- [ ] **Name keys on your plain HTTP writes**

  A plain HTTP write purges only the keys its response names in `Surrogate-Key`, so tag your `GET` responses with keys and send the same keys on the `POST`, `PUT`, `PATCH` or `DELETE` that changes them. See the Surrogate-Key header.

- [ ] **Have a way to purge what changes outside GraphPilot**

  A nightly import, an admin tool or a direct database write never passes through the edge. Wire `gpilot purge` or the purge API into those paths before you need them at 3am. The API key they use needs the `cache:purge` permission.

Before you scale up

- [ ] **Check the origin timeout against your cold starts**

  On Glider (Free) the edge waits 5 seconds for the first byte, not 15. A container that scales to zero can take longer and the request fails with a `502` while it is still booting. See origin timeouts.

- [ ] **Size your origin for a cache-less window**

  A deployment does not clear the cache, but a purge-all does, and on Glider (Free) exceeding the quota withdraws caching entirely. In both cases every request reaches your origin. Glider is offered under fair use: traffic far beyond what it is meant for can have a Glider account's services suspended until the start of the next month, or until it moves to a paid plan. See quotas.

- [ ] **Know how to go back to a previous version**

  In the portal, open **Deployments**, pick the deployment that was good and choose **Redeploy**: it rebuilds from that deployment's stored schema and config. The CLI has no rollback command, so keep `schema.graphql` and `gpilot.toml` in version control anyway and deploy the previous versions from there when you work from the CLI. A `gpilot.toml` that holds a JWT provider's shared `secret` is the exception: it is stored as written, so keep that file out of the repository or use a provider with a public key. See deployments.

- [ ] **Consider persisted queries before opening a public GraphQL API**

  They bound what can be executed, keep requests small enough for `GET`, and make your operation inventory answerable. See persisted queries.

Guides

GUIDES

Guides

Each guide solves one task end to end, on one running example: the goal, the configuration or schema, a request to send, what Insights or the portal shows, and the pitfalls. The concepts behind them are linked, not repeated.

I want to...

- **Move from another CDN**: coming from another CDN
- **Put GraphPilot in front of a REST API**: caching a REST API
- **Cache signed-in data per organization, not per user**: partitioning your cache
- **Show a new item in a cached list right away**: lists and creates
- **Clear cached responses when my data changes**: cache invalidation
- **Purge product pages when an editor publishes in my CMS**: purging from a CMS webhook
- **Cache one answer per language, not per browser**: caching per language
- **Run an A/B test without switching the cache off**: A/B testing with a cookie
- **Cache data per Auth0 organization**: caching per Auth0 organization
- **Name my clients, retire old versions, refuse unknown callers**: telling your clients apart
- **Refuse old builds of my mobile app**: retiring old mobile app versions
- **Allow only the operations my clients send**: migrating to trusted documents
- **Serve subscriptions over WebSockets safely**: subscriptions and WebSockets
- **Deploy from GitHub Actions or GitLab CI**: deploy from CI
- **Test changes on staging, and roll back**: staging and rollback
- **Deploy staging on merge and production after an approval**: promoting from staging in CI
- **Work through an alert email to the cause**: triaging an alert
- **Store as little request data as possible**: keeping request telemetry minimal
- **Find out why nothing is cached**: why is nothing cached?
- **Follow a request into my own logs**: correlating with your logs

---

GUIDES

Coming from another CDN

If you already run an API behind an edge cache, plain HTTP or GraphQL, most of what you configured there has a counterpart in GraphPilot. This page maps the usual concepts, points out where GraphPilot behaves differently, and ends with a checklist for moving traffic over.

How the concepts map

What you have today	In GraphPilot	Read
Cache rules or TTL settings per path	A `Cache-Control` header from your origin	From the origin header
Cache rules or TTL settings per GraphQL type or field	`@cacheControl` in your schema, or a `Cache-Control` header from your origin (which wins when it sets a lifetime or forbids storing)	How long an entry lives
A lifetime in `Surrogate-Control` or `Expires`	`Cache-Control`. GraphPilot reads only `Cache-Control`; use `s-maxage` for a lifetime that only shared caches apply	From the origin header
A stale window, serving old content while refreshing	`stale-while-revalidate` from your origin, or `swr` on `@cacheControl`	Stale windows
Caching per user or per session	The `private` cache policy, which names what identifies one caller, with `Cache-Control: private` from your origin or `scope: PRIVATE` in your schema	Identifying callers, authenticated requests
Caching per customer or per organization	A named scope such as `ORGANIZATION` on your GraphQL types, or a vary rule on the organization claim, so each group shares one entry	Named scopes, vary rules
Varying the cache on a header, cookie or query parameter	Your origin's `Vary`, or vary rules in the config	Identifying callers
Bypassing the cache for previews or editors	The `bypass` cache policy	Bypass
Cache tags your origin sets on a response	A `Surrogate-Key` header on your origin's response	The Surrogate-Key header
Purging by tag, by URL or everything	The purge API and `gpilot purge`: by surrogate key, by cache key (`ck:GET:/products/42` for one path), by request, or all	The purge API
Key fields, and purging a GraphQL entity when it changes	Surrogate keys derived from your schema's key fields, purged automatically when a mutation returns the object; `@surrogateKey` for keys of your own	Surrogate keys, automatic invalidation
Cache debug headers	`gp-cache` (`HIT`, `MISS`, `PASS`, `STALE`), `gp-cache-reason`, and the opt-in `gp-surrogate-key`	Headers, debugging surrogate keys
APQ, persisted or trusted documents	Persisted queries: APQ, and a registry you push from your client's manifest, which can refuse anything it does not hold	Persisted queries
JWT checks at the edge	Authentication providers in the config	Authentication

What you have today	In GraphPilot	Read
An IP allowlist or a shared secret at the origin	Published edge IP ranges, and a signature on every forwarded request	Origin allowlist, request signing
A header that sets the browser's or a downstream CDN's lifetime	The downstream `Cache-Control` header GraphPilot can write	Downstream header
Your own hostname	Custom domains with managed certificates, from Jet (Starter)	Custom domains

Differences worth knowing before you switch

- **Nothing is cached by default.** GraphPilot stores a response only when your origin's `Cache-Control` or your schema's `@cacheControl` allows it. A service that says neither passes everything through.
- **Only `Cache-Control` is read.** `Surrogate-Control` and `Expires` are ignored, so a lifetime your current CDN takes from either one has to move into `Cache-Control`.
- **Choose who shares an entry deliberately.** Data that is the same for a whole customer belongs in a named scope or a vary rule, not in a per-user entry that misses for every member. On a GraphQL API, `@cacheControl(scope:)` defaults to `PUBLIC`, as in Apollo Server. See who shares an entry.
- **A response that sets a cookie is never stored.** If your origin touches the session on every request, fix that first. See when a response is cached.
- **Writes purge what they name.** A GraphQL mutation purges what it returns, so one that returns `Boolean!` purges nothing; see automatic invalidation. A plain HTTP write purges the keys its response names in `Surrogate-Key`; see the Surrogate-Key header.
- **GraphQL query text is hashed verbatim**, and the cache is per edge location. Hit rates depend on how consistently your clients send the same requests and how concentrated your traffic is. See edge locations.
- **Configuration lives in files you deploy.** `gpilot.toml`, and your schema for a GraphQL API, go up together with `gpilot deploy`, so they can live in version control and ship from CI. Keep a JWT provider's shared HMAC `secret` out of version control, or use an asymmetric provider.

Migration checklist

> **Tip**
> Run GraphPilot next to your current setup first. Both can front the same origin at the same time, so nothing has to switch until you have compared them.

1. **Translate your rules.** Move your cache rules into your origin's `Cache-Control` (or, for a GraphQL API, into `@cacheControl` in your exported schema), and write the partitioning and authentication settings into `gpilot.toml`.
2. **Create the service and deploy.** The GraphQL quickstart and the HTTP quickstart cover it. Read the findings the build reports, in the `gpilot deploy` output or in the portal under schema and config findings.

3. **Send a share of real traffic to the service hostname.** Replay production requests, or point one client or one environment at `https://<subdomain>.<label>.graphpilot.cloud`.
4. **Compare hit rates.** Read `gp-cache` on the responses, or use Insights (part of every plan). A `PASS` names its reason in `gp-cache-reason`; see troubleshooting.
5. **Check that writes clear what they should.** Run your usual writes and repeat the requests they affect. Replace the purge calls your backend makes today with the purge API; the API key it uses needs the `cache:purge` permission.
6. **Verify the signature at your origin**, and add the GraphPilot edge IP ranges to your firewall if you restrict who may connect. See request signing. Start rejecting unsigned requests only once all traffic reaches your origin through GraphPilot, or your current CDN's requests fail.
7. **Cut over.** Change your clients' endpoint to the service hostname, or point your own hostname at GraphPilot with a custom domain (from Jet (Starter)). Keep the old path available until the new one has served real traffic for a while.
8. **Work through the go-live checklist.**

Related

- How GraphPilot works
- Features and limitations
- Go-live checklist
- Configuration reference

GUIDES

Caching a REST API

GraphPilot caches and protects a plain HTTP API without reading a schema. Your origin decides what may be cached and for how long, in the headers it already knows, and names what a write changes in one more header. This guide takes a REST API from its first cached response to purging on writes. If you have not deployed a service yet, do the HTTP quickstart first; this page continues from it.

The running example is a catalog API behind `https://catalog-api.k7m2qp.graphpilot.cloud` (`k7m2qp` stands for your organization's hostname label, which the portal shows): `GET /products/42` reads a product, `PUT /products/42` changes it, `GET /products?category=shoes` lists a category, and `POST /search` takes a query in its body.

What you need

- A service whose origin is your API, and the `gpilot` CLI logged in.
- The ability to set response headers on your origin.
- A `gpilot.toml`, and a deployed service.

A REST API needs no GraphQL schema: without a schema file, `gpilot deploy` deploys the config alone, as a plain HTTP service. The plain HTTP requests of a service deployed with a schema get everything on this page too.

1. Say how long a response may be cached

GraphPilot never decides on its own that a response is reusable. Your origin says so with `Cache-Control`:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=60, s-maxage=300, stale-while-revalidate=30
```

`s-maxage` is the lifetime at the edge, `max-age` the one a browser keeps, and `stale-while-revalidate` lets the edge serve an expired copy while it fetches a new one. A response with no `Cache-Control` is not stored. `no-store` and `no-cache` are never stored; `private` is stored only with a `private` policy. Which directives are read, and which headers are ignored (`Expires`, `Surrogate-Control`), is on lifetime from the origin header.

An entry is keyed by the request method and the path, query string included, so `/products?category=shoes` and `/products?category=bags` are two entries. The hostname is not part of the key, so the service hostname and a custom domain share one entry per path. If your origin answers the same path differently per hostname, set `cache.http_key.host = "include"` to key by the full URL.

A request that carries a body is keyed by a hash of its body as well, so two `POST /search` requests with different queries never share an entry.

A `4xx` or `5xx` response is not stored by default, even with a storable `Cache-Control`. If your origin marks a `404` cacheable on purpose, for a product that does not exist, list the status in `cache.policy.store_error_statuses`; see when a response is cached.

The body your client receives is the one your origin sent, byte for byte, on a hit and on a miss.

2. Partition what differs per caller

A shared lifetime serves one answer to everyone who sends the same request. Where the answer depends on who asks, derive the difference in `gpilot.toml`:

gpilot.toml

```
[auth.providers.idp]
type               = "jwt"
allowed_algorithms = ["RS256"]
jwks_url           = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers    = ["https://idp.example.com/"]
allowed_audiences  = ["my-api"]

[cache.policy]
private = 'auth:idp.sub'
```

```
bypass  = 'query:preview'

[[cache.policy.vary]]
name  = "x-org"
value = 'auth:idp.org_id'

[[cache.policy.vary]]
name  = "x-locale"
value = "lower(split(split(split(header:accept-language, ',', 0), ';', 0), '-', 0)) ||
'en'"
pass_to_origin = "plain"
```

- `auth:` reads a verified claim. Use it for anything that is a security boundary: the organization here, the user in `private`.
- `header:` reads a request header the caller chose. Fine for a preference such as the language, never for a boundary. The rule reduces `Accept-Language` to its primary language and sends it to your origin as `x-locale`, which your origin renders from; see caching per language.
- `query:` reads a URL parameter. The query string is already part of the key, so on a vary rule it adds nothing; it is useful in `bypass`, which sends `?preview=1` requests past the cache.

Every vary rule partitions every entry, so configure only those whose value shapes a response. A request from a signed-in caller is shared only when your origin says so with `public` or `s-maxage`; see authenticated requests. Mark a response for one person `Cache-Control: private, max-age=60`, and it is stored per `sub`. A full walk-through with an organization and a role is in partitioning your cache.

3. Tag reads with keys you choose

Name what a response depends on in a `Surrogate-Key` header:

```
# The response to GET /products/42
HTTP/1.1 200 OK
Cache-Control: s-maxage=300
Surrogate-Key: product-42 category-shoes
```

```
# The response to GET /products?category=shoes
HTTP/1.1 200 OK
Cache-Control: s-maxage=300
Surrogate-Key: category-shoes
```

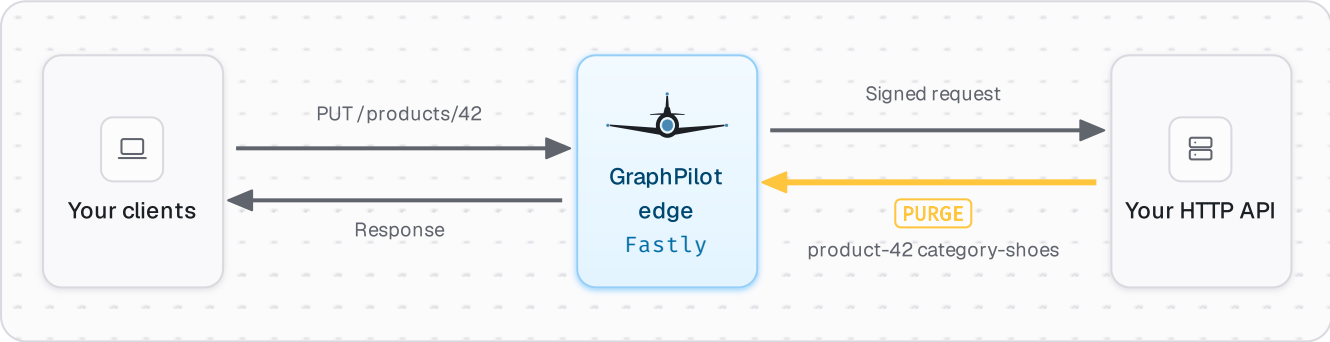
On the response to a `GET` or `HEAD`, the header tags the entry. GraphPilot removes it before the response reaches your client. Keys are separated by spaces or commas, and anything outside printable ASCII is folded (`Müller` becomes `Muller`); the rules are on the Surrogate-Key header.

4. Purge on writes

On the response to a `POST`, `PUT`, `PATCH` or `DELETE`, the same header purges instead of tagging. Answer the write with the keys it changed:

```
# The response to PUT /products/42
HTTP/1.1 200 OK
Surrogate-Key: product-42 category-shoes
```

Every cached response tagged `product-42` or `category-shoes` is removed at every edge location, and the next `GET` for each goes to your origin. The purge is recorded in the service's purge history.



The write's response names the keys it changed in Surrogate-Key; every entry tagged with one of them is removed.

For a change that does not pass through GraphPilot (an import, an admin tool, a database migration), purge from that code or a runbook:

```
gpilot purge product-42              # by a key your origin named
gpilot purge ck:GET:/products/42     # one response, by its cache key
gpilot purge --all                   # everything
```

Each command asks for confirmation first. From a script or CI job, add `--yes`: without a terminal to ask on, the command fails instead.

There is no purge by URL: `ck:<METHOD>:<path>`, with the query string but without scheme and hostname, is how you address one response (the full URL, if the service sets `host = "include"`). The API offers the same, plus purging the entry one logged request read; see purging through the API and CLI.

5. Decide what happens when your origin fails

Retries are off until you switch them on under `[origin.retry]`.

For a plain HTTP request, GraphPilot then repeats only `GET`, `HEAD` and `OPTIONS`, and sends a write once unless you set `retry_unsafe_methods = true` for an origin whose writes are idempotent. Which failures count, the backoff and the time budget are on retries.

Check that it works

Read, write, read again:

```
H="https://catalog-api.k7m2qp.graphpilot.cloud"
curl -sS -D- -o/dev/null "$H/products/42" | grep -i '^gp-cache:'    # MISS
curl -sS -D- -o/dev/null "$H/products/42" | grep -i '^gp-cache:'    # HIT
curl -sS -o/dev/null -X PUT "$H/products/42" \
  -H 'content-type: application/json' -d '{"price":99}'
curl -sS -D- -o/dev/null "$H/products/42" | grep -i '^gp-cache:'    # MISS
```

The third read is a `MISS` because the `PUT`'s response purged `product-42`.

To see the keys an entry carries, switch on `response.surrogate_key_header` and read `gp-surrogate-key`; see debugging surrogate keys. A `PASS` carries `gp-cache-reason` with the rule that refused; why is nothing cached? goes through them.

Pitfalls

- **A cookie on a read disables caching for it.** A framework that refreshes the session on every response sends `Set-Cookie`, and nothing is stored (`CACHE_SKIPPED_SET_COOKIE`). See what refuses storing.
- `Authorization` **on every request.** Without `public` or `s-maxage` on the response, it is filed per caller or not cached (`CACHE_SKIPPED_CREDENTIALED_REQUEST`).
- **Non-ASCII keys are folded, and folding is lossy.** `Müller` and `Muller` become one key, so a purge of either clears both. Prefer identifiers to names in keys.
- `Vary: *` is never stored. Name the headers you vary on.
- `OPTIONS` **is not cached** by default, so CORS preflights reach your origin; see `cache.policy.uncacheable_methods`.
- **Keep REST routes outside a GraphQL path.** If the service also has a schema, every path at or below an entry of `request.graphql_paths` (`/graphql` by default), in any letter case, is read as GraphQL. A `POST /graphql/health` with a JSON body is refused with `415` and `UNSUPPORTED_GRAPHQL_BODY`; a `GET` without a body or GraphQL parameters still passes. Move such routes to their own path, or list the body's kind in `request.forward_unchecked_bodies`.
- `POST /__gp/purge` **is reserved.** GraphPilot answers a `POST` to exactly that path itself, on every hostname of the service, and never forwards it. Do not use it in your API.

What stays GraphQL-only

Everything above works without a schema. These features need a parsed GraphQL operation:

Feature	For a plain HTTP API instead
Lifetimes and scopes from `@cacheControl`	`Cache-Control` from your origin

Feature	For a plain HTTP API instead
Named scopes such as one entry per organization	A vary rule, which applies to every entry
Surrogate keys derived from the schema, selection set expansion	`Surrogate-Key` from your origin
Automatic purging of the objects a mutation returned	`Surrogate-Key` on the write's response
Purge by query hash	Purge by key or by cache key
Persisted queries and trusted documents	Nothing; your paths are your allowlist
Limits on query depth, document size and selections; introspection block	The request body limit, which applies to both
Client profile rules for depth, size, selections and persisted queries	The other client profile rules, which apply to both

The full matrix is on features and limitations.

Related

- Quickstart: HTTP
- Lifetime from the origin header
- Identifying callers
- The Surrogate-Key header
- Request signing, to make your origin accept only traffic that came through GraphPilot

---

GUIDES

Partitioning your cache

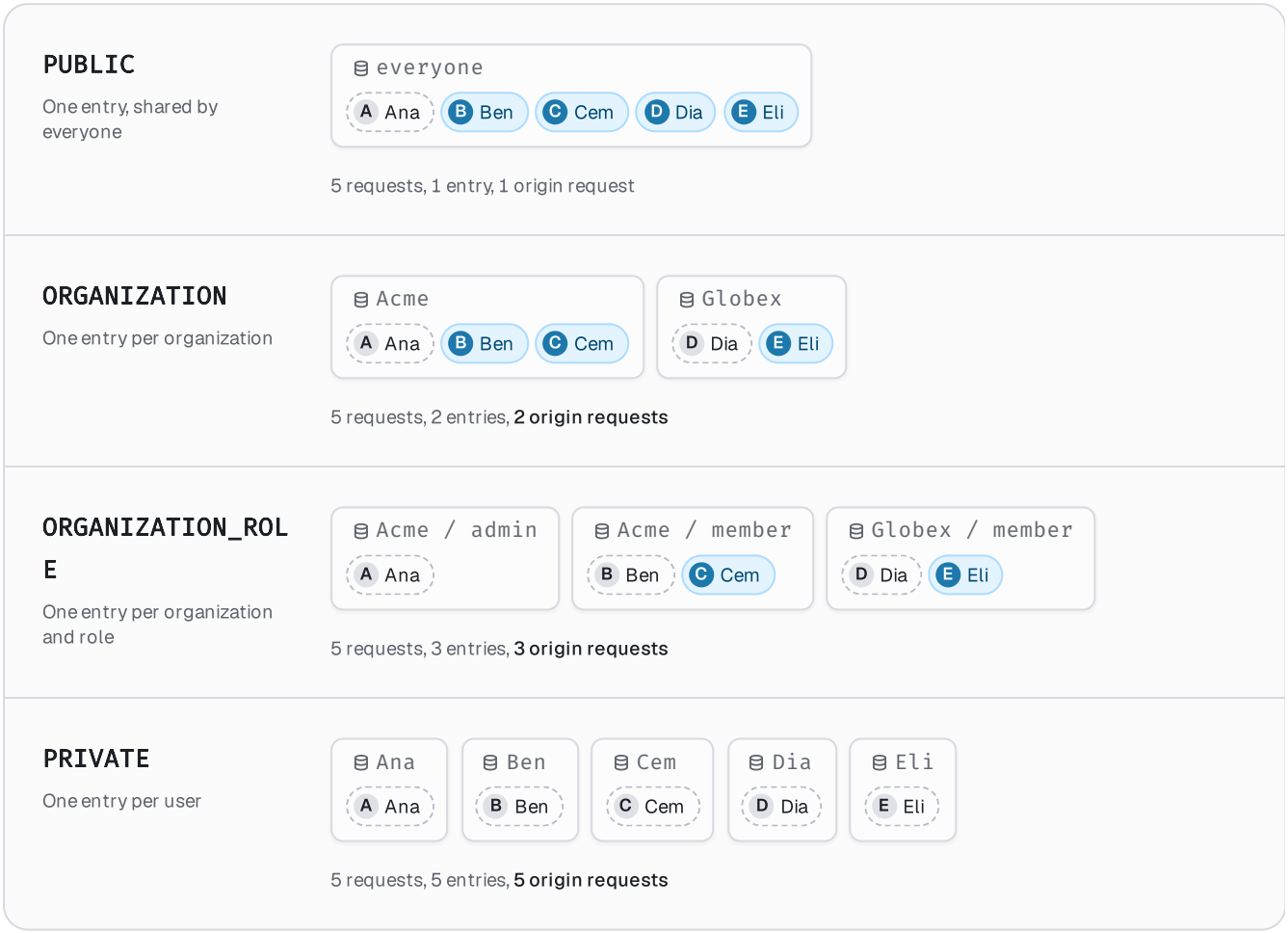
Most data behind a login is not personal. A project list is the same for everyone in the organization, a billing page is the same for every admin in it. Caching such data per user (`PRIVATE`) works, but it pays one origin request per person for an answer the whole team shares. This guide keys it on the organization, and on the role where the role changes the answer. The concepts are on who shares an entry; this page is the setup.

The running example is a SaaS product whose identity provider issues a JWT with two claims besides `sub` : `org_id` and `role` . Five people use it:

User	`org_id`	`role`
Ana	`acme`	`admin`
Ben	`acme`	`member`
Cem	`acme`	`member`

User	org_id	role
Dia	globex	member
Eli	globex	member

Each of them opens the same page once, on a cold cache. How many entries that page's data ends up in, and so how many requests reach your origin, depends on the audience you give it:



Ana, Ben, Cem, Dia and Eli open the same page once each, on a cold cache.

Audience	Scope	Entries	Origin requests
Everybody	PUBLIC	1	1
The organization	ORGANIZATION	2 (Acme, Globex)	2
The organization and role	ORGANIZATION_ROLE	3 (Acme admin, Acme member, Globex member)	3
One person	PRIVATE	5	5

Each row holds on its own because step 2 marks both vary rules `scope_only`: a named scope, and `PUBLIC`, then partitions only on the vary names it lists, so configuring `x-role` for the organization-and-role rung does not split the `PUBLIC` or `ORGANIZATION` rungs by role too. Without `scope_only`, every configured rule partitions every response, and the first three rows would each hold 3 entries.

What you need

- A GraphQL origin whose responses carry no `Cache-Control` of their own, so the schema decides. If your origin sends one, it decides the lifetime and has to say `public` or `s-maxage`: a bare `max-age` on a request carrying a token is filed per caller (see authenticated requests), so the `HIT`s under check that it works do not happen. With `public` or `s-maxage`, every configured vary rule keys the entry on that path, so Ana does not share Ben and Cem's entry; see scopes in the schema. On Apollo Server, see option B. For a plain HTTP API, read the steps and then the plain HTTP section.
- Your identity provider's JWKS URL, its issuer, and the names of the claims you partition on.
- A service you can deploy to, and a token for at least three of the five users.

1. Verify the token at the edge

A claim can only partition the cache once GraphPilot has verified the token it came from. Add a provider to `gpilot.toml`:

gpilot.toml

```
[auth.providers.idp]
type               = "jwt"
allowed_algorithms = ["RS256"]
jwks_url           = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers    = ["https://idp.example.com/"]
allowed_audiences  = ["my-api"]
```

The provider reads the `Authorization` header and strips a `Bearer` prefix. Its claims are now readable as `auth:idp.<claim>`, where `idp` is the block name. Every key is on `[auth.providers.<name>]`, and what happens to a token that does not verify is on authentication.

2. Derive the organization and the role

Add one vary rule per claim, and a `private` expression for the data that really is personal:

gpilot.toml

```
[cache.policy]
private = 'auth:idp.sub'

[[cache.policy.vary]]
name       = "x-org"
value      = 'auth:idp.org_id'
scope_only = true
```

```
[[cache.policy.vary]]
name       = "x-role"
value      = 'auth:idp.role'
scope_only = true
```

`auth:idp.org_id` reads the verified `org_id` claim; a nested claim is a dotted path, such as `auth:idp.org.id`. Each rule writes its value onto the request under the header named in `name`, and GraphPilot removes anything a client sent under that name first, so the value cannot be forged. Keep both from `auth:`: an organization and a role are security boundaries, and a header the caller sets is not. See vary rules.

`scope_only = true` leaves each rule to the scopes that name it, so a response your schema decides pays for `x-role` only where its scope names it, such as `ORGANIZATION_ROLE`. Without the flag, every rule keys every response, and the role would split even the `PUBLIC` price list. `scope_only` needs `pass_to_origin = "never"`, the default; a rule your origin receives cannot carry it. Configure `x-role` only if some data really depends on the role.

3. Name the audiences in the schema

Add the two scopes to your `CacheControlScope` enum and name the vary rules each one partitions on:

schema.graphql

```
directive @vary(names: [String!]!) on ENUM_VALUE

enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION      @vary(names: ["x-org"])
    ORGANIZATION_ROLE @vary(names: ["x-org", "x-role"])
}
```

The names in `@vary` must match the `name` of a `[[cache.policy.vary]]` entry. Declare `@cacheControl` as well, as shown in its definition.

4. Annotate the types

Put each type on the widest audience that is still correct:

schema.graphql

```
type Plan @cacheControl(maxAge: 3600) {                    # PUBLIC: the price list
    id: ID!
    name: String!
}

type Project @cacheControl(maxAge: 300, scope: ORGANIZATION) {
    id: ID!
    name: String!
```

```
}

type Invoice @cacheControl(maxAge: 300, scope: ORGANIZATION_ROLE) {
    id: ID!
    total: Int!
}

type Query {
    plans: [Plan!]!
    projects: [Project!]!
    invoices: [Invoice!]!                                    # admins see all, members
fewer
    me: User @cacheControl(maxAge: 60, scope: PRIVATE)
}
```

`ORGANIZATION` asserts that every member may see every row. If some members may see less, the scope has to name what separates them, as `ORGANIZATION_ROLE` does for invoices. A cache hit never reaches your origin's authorization; see is your field really organization-uniform?

Deploy both files with `gpilot deploy`. The build reports a `@vary` name that no `[[cache.policy.vary]]` entry defines under **Schema → Findings**; see findings.

Check that it works

Send the same query as Ben, then as Cem, then as Dia. `$BEN`, `$CEM` and `$DIA` hold their tokens:

```
for token in "$BEN" "$CEM" "$DIA"; do
  curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
    -H "authorization: Bearer $token" \
    -H 'content-type: application/json' \
    -d '{"query":"{ projects { id name } }"}' | grep -i '^gp-cache:'
done
```

```
gp-cache: MISS          # Ben: the first request for Acme members, stored
gp-cache: HIT           # Cem: same organization, same role, same entry
gp-cache: MISS          # Dia: Globex has its own entry
```

Ana, also at Acme, gets a `HIT` too: `ORGANIZATION` partitions only on `x-org`, and `x-role` is marked `scope_only`, so configuring it for `Invoice` does not split this entry by role. In Insights, the **Segmentation** field on the request's **Cache** tab shows what the entry was split by.

A `PASS` with `gp-cache-reason: CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR` means a name the scope lists was missing for that request. The pitfalls below cover why.

Pitfalls

- **A name with no vary rule behind it never fills.** `@vary(names: ["x-org"])` without an `x-org` entry refuses to store every response that selects the type, forever, with `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`. The deployment does not fail; the finding is your only warning.
- **A missing claim refuses the store.** A token without `org_id` resolves `x-org` to nothing, and a named scope then refuses rather than sharing. The same happens to a claim that is an object, an array or `null`. A number or a Boolean resolves to its JSON text, so an `org_id` issued as `42` partitions as `"42"`.
- **An empty claim shares the entry.** A verified token whose `org_id` is `""` resolves `x-org` to the empty string, and the empty string is a value: `ORGANIZATION` stores the response, and every caller whose claim is empty reads that one entry, whatever organization they belong to. Have your identity provider never issue the claim empty. A rule without `scope_only` does not have this gap: for a signed-in caller, an empty value there refuses the store with `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`, at the cost of partitioning every response on it. See an empty value is a value.
- **A role changes when the token does.** The role comes from the token, so a user promoted or demoted keeps reading the old role's entries until they get a new token. Keep token lifetimes short where roles change often, and purge what a demotion must hide at once; see revocation.

When PRIVATE is right

Use `PRIVATE` for what differs per person: `me`, a shopping cart, notifications, a user's settings, rows only they own. It partitions by `cache.policy.private` (`auth:idp.sub` above) and misses once per person. A query that selects one `PRIVATE` field is private as a whole, so keep personal fields out of queries that load shared data.

Per-user data you can still read by organization

Notifications differ per person, so they belong on the last rung. Scoped `PRIVATE`, they are keyed on `cache.policy.private`, and Insights records that value only as a keyed digest: you can see that notification requests miss, but not whether they miss for Acme or for Globex. A named scope keyed on the organization and a pseudonym of the user is just as narrow and keeps the organization readable.

Add a third rule beside the two from step 2, and show vary values in clear:

gpilot.toml

```
[[cache.policy.vary]]
name       = "x-org-user"
value      = "auth:idp.org_id + ':' + pseudonymize(auth:idp.sub)"
scope_only = true

[telemetry]
vary_values = "clear"
```

Name it in a scope of its own and put the per-user type on it:

schema.graphql

```
enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION      @vary(names: ["x-org"])
    ORGANIZATION_ROLE @vary(names: ["x-org", "x-role"])
    ORGANIZATION_USER @vary(names: ["x-org-user"])
}

type Notification @cacheControl(maxAge: 30, scope: ORGANIZATION_USER) {
    id: ID!
    text: String!
}
```

All five people open their notifications once. That is five entries and five origin requests, as with `PRIVATE`, and the **Segmentation** field on each request's **Cache** tab in Insights reads:

User	Segment recorded for `x-org-user`
Ana	`4:acme:43:` and Ana's pseudonym
Ben	`4:acme:43:` and Ben's pseudonym
Cem	`4:acme:43:` and Cem's pseudonym
Dia	`6:globex:43:` and Dia's pseudonym
Eli	`6:globex:43:` and Eli's pseudonym

Each part is written as its length in bytes, a colon and the value, because the value combines two sources; the 43 characters after the second colon are the pseudonym. The other rules are recorded in clear too: Ben's `projects` request shows `acme` for `x-org`, and his `invoices` request `acme` and `member`, which is the point of the setting.

- **Keep `scope_only` on the rule.** Without it the rule partitions every response, and the `PUBLIC` price list and the `ORGANIZATION` projects would be stored once per person.
- **`vary_values = "clear"` covers every rule.** Before you set it, check that no rule's value is a personal identifier in clear: an email address, a user id, a session cookie. Wrap such a value in `pseudonymize`, as here.
- **Without `clear`**, the segment is recorded as an `hmac-sha256:` digest like every other vary value, and the organization is no more readable than under `PRIVATE`.
- **It fails closed.** A token without `org_id` or `sub` derives no `x-org-user`, so its notifications are not stored (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`).

For a plain HTTP API

Plain HTTP has no schema, so there are no named scopes: steps 1 and 2 are the whole setup, and every vary rule applies to every cached response, `scope_only` or not.

- **Have your origin mark shared responses `public` or `s-maxage`.** A request carrying `Authorization` is otherwise filed per caller or not cached. `Cache-Control: s-maxage=300` on `GET /projects` gives one entry per organization with only `x-org` configured.
- **Mark personal responses `private`.** `Cache-Control: private, max-age=60` on `GET /me` is partitioned by `cache.policy.private`.
- **A missing claim refuses a signed-in caller only.** Without a named scope, a request that carries a credential and whose rule resolves to nothing, or to the empty string, is not stored (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`), so its answer never lands in the variant anonymous callers read. A request without a credential is stored as the variant without the header, shared by every other anonymous caller. Make sure every token that reaches a partitioned endpoint carries the claim, or those callers miss every time.

The `curl` check above works the same way against `GET /projects`.

Related

- Who shares an entry: the ladder and the buckets
- Scopes in the schema
- Identifying callers: providers, expressions, vary rules
- Policy expressions
- Caching a REST API

GUIDES

Lists and creates

An update or a delete clears the cached lists that contained the object, because the object's key is on both. A create cannot: the new object is in no cached response yet, so its key clears nothing, and a list that already held items keeps showing the old ones until it expires. Why that is so is on automatic invalidation. This guide gives the list a key that the create names, so the create purges it.

The running example is a blog. `posts` lists the latest posts and `createPost` adds one. As plain HTTP, the same is `GET /posts` and `POST /posts`.

What you need

- A service whose list responses are cached (a `MISS` then a `HIT` for the list).

- For GraphQL, the schema you deploy, and an origin that can set a response header on the mutation. For plain HTTP, an origin that can set response headers on the read and the write.

1. Pick one key for the list

Choose a key that names the list, not an object: `post-list`. It has to be printable ASCII with no spaces or commas; see what a key may contain. Prefix it if you want to be sure it never collides with a key GraphPilot adds.

Pick the width deliberately. One key on every list of posts purges all of them on every create, which is usually right for a blog. A key per author (`post-list-7`) purges only that author's lists, at the cost of your origin knowing the author on the write.

2. Tag the list

GraphQL. Put a `static` key on the list field. Every response that selects `posts` then carries `post-list`, whatever came back, an empty list included:

schema.graphql

```
type Query {
    posts(first: Int = 20): [Post!]! @surrogateKey(static: "post-list")
}
```

Put the key on the `Post` type instead if every query that touches a post should be purged together. The difference is which queries carry it; see a group you purge by hand. Your origin can also tag the query response itself with a `Surrogate-Key: post-list` header; both kinds of key are reached by the same purge.

Plain HTTP. Your origin tags the response to the read:

```
# The response to GET /posts
HTTP/1.1 200 OK
Cache-Control: s-maxage=300
Surrogate-Key: post-list
```

3. Name the key on the create's response

A `static` key is never purged automatically, so the write has to name it. Set the header on the response to the create, in the resolver or a middleware that knows the operation succeeded:

```
# The response to the createPost mutation, or to POST /posts
HTTP/1.1 200 OK
Surrogate-Key: post-list
```

On a GraphQL mutation, and on the response to a plain HTTP `POST`, `PUT`, `PATCH` or `DELETE`, this header purges every cached response carrying `post-list`, at every edge location. See the Surrogate-Key header.

Send it only on success. A failed create changed nothing, and a purge it triggers costs origin requests for nothing.

Another way for GraphQL: return the parent. When the list hangs off an object, such as `author { posts }`, let the mutation return that object. Its key is purged automatically and reaches every cached response that contained it, the list included:

schema.graphql

```
type CreatePostPayload {
    post: Post
    author: Author        # purges Author:id:<id>, so author { posts } is refetched
}
```

Selecting the payload's fields is not required from the client; see mutations that return a payload type.

4. Purge from outside, when the create does not pass through

A post created by an import job or an admin tool never produces a response here. Purge the key from that code:

```
gpilot purge --yes post-list
```

Or call `purgeBySurrogateKey`; see purging through the API and CLI.

Check that it works

```
Q='{"query":"{ posts { id title } }"}'
M='{"query":"mutation { createPost(title: \"Hello\") { post { id } } }"}'
send() { curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' -d "$1" | grep -i '^gp-cache:'; }

send "$Q"   # MISS: stored, tagged post-list
send "$Q"   # HIT
send "$M"   # the mutation; its response names post-list
send "$Q"   # MISS: the list was purged, and the new post is in it
```

For plain HTTP, the same with `GET /posts` and `POST /posts`. To see that the list carries the key, switch on `gp-surrogate-key`; see debugging. The service's **Purges** page lists the purge with the request that triggered it; see verifying a purge.

Pitfalls

- **An empty list needs nothing of this.** A cached empty list is tagged `![Post]`, which any mutation touching a post purges. Only a list that already held items needs a key; see why the empty-list key exists.
- **A mutation that returns `Boolean` purges nothing automatically.** It carries no object and no key. Return the object or its parent, or name the key in the header.
- **A selection of `__typename` alone still carries keys.** `{ posts { __typename } }` reports how many posts exist, which a create changes, so it is tagged like a full selection and purged with it. Only `{ __typename }` at the root, which resolves no field, earns no content keys.
- **A nullable key field changes its key when it goes from `null` to a value.** With `author: Author @surrogateKey(of: "id")` on `Book`, a book with no author is tagged `Book:author.id:` and the same book with author `A1` is tagged `Book:author.id:A1`. Purging only the new key misses the cached version without an author. Purge by the object's own key instead, `Book:id:B1`, which reaches it whatever its author was.
- **A key that names too much purges too much.** One `catalog` key on every list empties the whole catalog on each create. Watch the hit rate after you add a key.

Related

- Automatic invalidation: what a mutation purges on its own
- The Surrogate-Key header
- Keys from the schema
- Caching a REST API

GUIDES

Caching per language

Goal: a product catalog answers in the caller's language, and every caller who reads German shares one cached German answer, whatever their browser puts in `Accept-Language`.

Partitioning on the raw header does not get you there. `de-DE,de;q=0.9,en;q=0.8`, `de-AT` and `de` are three different strings, so they would be three cache entries holding the same German text, and the number of entries grows with every browser and every language setting in the wild. A vary rule can reduce the header to the one part your origin actually uses.

The running example is `shop-api`, whose origin translates product names and category names. It supports English and German and answers English for anything else.

What you need

- A deployed service and its `gpilot.toml`.
- An origin that can read the language from a header GraphPilot sets, here `x-locale`.

1. Derive the language

gpilot.toml

```
[[cache.policy.vary]]
name           = "x-locale"
value          = "lower(split(split(split(header:accept-language, ',', 0), ';', 0), '-',
0)) || 'en'"
pass_to_origin = "plain"
```

Read the expression from the inside out, for `de-AT,de;q=0.9,en;q=0.8`:

Step	Result
`split(header:accept-language, ',', 0)`: the first language range	`de-AT`
`split( … , ';', 0)`: without a `;q=` weight, if the first range has one	`de-AT`
`split( … , '-', 0)`: the primary language, without the region	`de-AT` becomes `de`
`lower( … )`: one spelling for `DE` and `de`	`de`
`|| 'en'`: the fallback when there is no header	`en`

`pass_to_origin = "plain"` sends the result to your origin as `x-locale: de`. Answer from that header, not from `Accept-Language`: the cache entry is keyed on `x-locale`, so an answer that depended on anything else in `Accept-Language` would be served to callers it was not meant for.

The rule has no `scope_only`, so it partitions every cached response of the service, which is what a translated catalog needs. Configuring it also claims the name: an `x-locale` header a client sends itself is removed on the way in.

> ⚠ **The fallback matters**
>
> Without `|| 'en'`, a request with no `Accept-Language` derives nothing. An anonymous caller is then stored as the variant without the header, but a caller with a token is not stored at all (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`), because GraphPilot cannot tell which variant a credentialed caller belongs in. With the fallback, every request derives a language.

2. Deploy and send two requests

Deploy with `gpilot deploy`, then ask for the same product with two different German browser settings:

```
for lang in 'de-DE,de;q=0.9,en;q=0.8' 'de-AT'; do
  curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
    -H 'content-type: application/json' \
    -H "accept-language: $lang" \
```

```
    -d '{"query":"query ProductName { product(id: \"p01\") { id name } }"}' \
    | grep -i '^gp-cache:'
done
```

```
gp-cache: MISS
gp-cache: HIT
```

Both derive `de`, so the second request is served from the entry the first one stored. A third request with `accept-language: en-US` derives `en` and is a `MISS` of its own.

3. Read it in Insights

Open one of the requests in Insights. The **Cache** tab shows **Segmentation Vary**, split by `x-locale`. By default the value is stored as a keyed pseudonym (`hmac-sha256:` and 16 hex digits), marked **pseudonymized**, so the two German requests show the same pseudonym and the English one another. **Requests in this bucket** lists every request that read the German entry.

A language is not personal data, so you may prefer to read `de` in clear. That is `telemetry.vary_values = "clear"`, and it applies to every vary rule of the service: switch it only if every other rule's value is fine to store in clear.

Pitfalls

- **Every language a client sends is its own entry.** The rule collapses regions and weights, not languages: a caller sending `xx` gets an `xx` entry, answered in English by your origin. That stays bounded by the languages clients send, which is far fewer than the header strings. If you need a fixed set, let your client send `x-app-locale` from a list it controls and vary on `lower(header:x-app-locale) || 'en'` instead.
- **Only the first language range counts.** `fr-CH,de;q=0.9` derives `fr`, even though your origin only speaks German and English. That is the right entry as long as your origin answers `fr` with its fallback.
- **The rule splits everything.** Responses that are the same in every language, such as prices served by a separate query, are stored once per language too. Keep untranslated data in its own operations if the hit rate matters.
- **Only two languages?** A `public` predicate such as `starts_with(lower(header:accept-language), 'de')` gives exactly two entries. Its value is not sent to your origin, so your origin has to pick the language from `Accept-Language` the same way; see `public`.
- **A `gp-` name does not work.** GraphPilot removes `gp-` headers before your origin, so `pass_to_origin` could never deliver it. Use an `x-` name.

Related

- Vary rules
- Text functions and cardinality
- Partitioning your cache

GUIDES

A/B testing with a cookie

Goal: half of your visitors see a new product page layout, the other half the current one, and both halves are still served from the cache: one entry per variant, not one per visitor.

Two things stand in the way by default. A `Cookie` header makes a request credentialed, which keeps it out of the shared cache, and a `Set-Cookie` on a response keeps that response out of the cache entirely. A bucket cookie carries no identity, so you tell GraphPilot so, and then partition on its value.

The running example is `shop-api`. The client assigns each visitor a bucket and keeps it in a cookie, `ab_bucket=a` or `ab_bucket=b`. The origin renders the new layout for bucket `b`.

What you need

- A deployed service and its `gpilot.toml`.
- A client that assigns the bucket itself and stores it in a cookie. Why the client and not the origin is the first pitfall.

1. Say the cookie carries no session

gpilot.toml

```
[cache.cookies]
allow = [
  "GAESA", "GCLB", "AWSALB", "AWSALBCORS", "AWSELB", "ARRAffinity", "ARRAffinitySameSite",
  "BIGipServer*", "__cf_bm", "__cflb", "incap_ses_*", "visid_incap_*",
  "ab_bucket",
]
```

Setting `cache.cookies.allow` replaces the default list rather than adding to it, so the defaults are restated and `ab_bucket` added at the end. Matching is case-sensitive: `AB_Bucket` would not match.

An allowed name has three effects: a `Cookie` header made only of allowed names does not make the request credentialed, an allowed name in a `Set-Cookie` does not refuse storing the response, and an allowed name is never written to the cache entry. The cookie still reaches your origin and your client untouched. See what `cache.cookies.allow` actually does.

2. Partition on the variant

gpilot.toml

```
[[cache.policy.vary]]
name           = "x-ab-variant-b"
value          = "in(cookie:ab_bucket, ['b']) || 'false'"
pass_to_origin = "plain"
```

`in( … )` is a truth, so the rule derives exactly one of two values: `true` for bucket `b`, and `false` for bucket `a`, for a visitor without the cookie (through `|| 'false'`), and for any value somebody makes up. Two values mean two cache entries per response, however many cookie values arrive. Your origin receives `x-ab-variant-b: true` or `x-ab-variant-b: false` and renders from that header, not from the cookie.

Partitioning on `cookie:ab_bucket` directly would also work for well-behaved clients, but every value a caller sends would become an entry of its own. See cardinality.

3. Deploy and send three requests

```
ask() {
  curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
    -H 'content-type: application/json' \
    "$@" \
    -d '{"query":"query ProductPage { product(id: \"p01\") { id name price } }"}' \
    | grep -i '^gp-cache:'
}

ask -H 'cookie: ab_bucket=b'   # variant B
ask -H 'cookie: ab_bucket=b'   # variant B again
ask -H 'cookie: ab_bucket=a'   # variant A
ask                            # no cookie: also variant A
```

```
gp-cache: MISS
gp-cache: HIT
gp-cache: MISS
gp-cache: HIT
```

The second request reads the entry the first one stored. Bucket `a` gets its own entry, and the visitor without a cookie shares it, because both derive `false`.

4. Read it in Insights

In Insights, a request's **Cache** tab shows **Segmentation Vary**, split by `x-ab-variant-b`. The value is a keyed pseudonym by default, and with only two possible values there are only two pseudonyms: copy one into the **Segment** filter to see every request of that arm, or use **Requests in this bucket** on the Cache tab. The **Vary-segmented** preset lists the requests that were served from a segmented entry.

Pitfalls

- **Assign the bucket in the client.** An allowed cookie is never written to the cache entry, so a cache hit never hands out a `Set-Cookie: ab_bucket=…`. If your origin assigned the bucket, only the visitors whose first request happened to miss would ever get one. Set the cookie in your client, or from an endpoint that is not cached.

- **Every other cookie still counts.** A request that also carries a session cookie, or an analytics cookie such as `_ga`, is credentialed again: `ab_bucket` on the allowlist does not vouch for its neighbours. Those requests are cached only if a private policy identifies the caller.
- **A test with more arms needs more rules or a fixed value set.** `in( … )` answers one question per rule. For three arms, give the client a fixed list of values and partition on `lower(cookie:ab_bucket) || 'a'`, accepting that a made-up value gets an entry of its own.
- **Ending the test does not clear the cache.** Entries keep their variant until they expire. Purge the affected responses when you switch the winner on for everyone; see purging through the API and CLI.

Related

- Header and cookie filters
- Vary rules
- Authenticated requests
- Caching per language

GUIDES

Caching per Auth0 organization

Goal: your B2B app signs users in with Auth0 Organizations. A sales report is the same for every member of one organization with one role, so it should be cached once per organization and role, not once per user, and never served across organizations.

The mechanism is the same for any identity provider that issues JWTs with an organization claim; partitioning your cache explains it in general. This recipe fills in the Auth0 values.

The running example is `shop-api` behind the Auth0 tenant `shop-example.eu.auth0.com`. Its API is registered in Auth0 with the identifier `https://shop-api.example.com`, which becomes the `aud` of every access token for it. With Organizations, Auth0 puts the organization's id in the `org_id` claim. A post-login Action adds the member's role as `shop_role`.

What you need

- An Auth0 API whose access tokens carry `org_id`, and a role claim if what members may see depends on their role.
- A GraphQL schema and a deployed service. For a plain HTTP API, the same vary rules apply without the schema step; see vary rules.

1. Verify the token

gpilot.toml

```
[auth.providers.auth0]
```

```
type               = "jwt"
allowed_algorithms = ["RS256"]
jwks_url           = "https://shop-example.eu.auth0.com/.well-known/jwks.json"
allowed_issuers    = ["https://shop-example.eu.auth0.com/"]
allowed_audiences  = ["https://shop-api.example.com"]
```

- `allowed_algorithms` names RS256, the algorithm Auth0 signs API access tokens with by default. It is required.
- `allowed_issuers` is required for a provider with a `jwks_url`. Copy the issuer with its trailing slash: Auth0 writes `iss` that way, and the comparison is exact.
- `allowed_audiences` keeps tokens Auth0 issued for your other APIs out. Auth0 can issue `aud` as an array; a token is accepted when any of its audiences is on the list, and a token without `aud` is refused.

The provider reads the bearer token from `Authorization`. A token that does not verify is answered `401` at the edge and never reaches your origin. See authentication.

2. Derive the organization and the role

gpilot.toml

```
[cache.policy]
private = 'auth:auth0.sub'

[[cache.policy.vary]]
name       = "x-gp-organization"
value      = 'auth:auth0.org_id'
scope_only = true

[[cache.policy.vary]]
name       = "x-gp-role"
value      = 'auth:auth0.shop_role'
scope_only = true
```

`private` keys data that belongs to one person on the Auth0 user id. The two vary rules derive the organization and the role from the verified token; a client cannot choose either. `scope_only = true` leaves each rule to the schema scopes that name it, so public data stays shared by everyone.

The role is there because the organization alone is not a safe bucket. A cache hit never reaches your origin, so whatever your origin would check for the second caller is not checked. Keyed on the organization alone, a member with a viewer role would be served the report an admin of the same organization had filled the cache with.

> **⚠ Quote claim names with a URL namespace**
>
> A claim is read by its path, and each step of a path is letters, digits, `_` and `-`, or a quoted name. A claim named in the older Auth0 style is read by quoting it: `auth:auth0.'https://shop-api.example.com/role'`. In `gpilot.toml`, write such a value as a double-quoted string, which holds the single quotes as they are: `value = "auth:auth0.'https://shop-api.example.com/role'"`. A claim whose value is an array, such as Auth0's `permissions`, resolves to nothing and cannot partition the cache.

3. Name the scope in the schema

schema.graphql

```
directive @vary(names: [String!]!) on ENUM_VALUE

enum CacheControlScope {
  PUBLIC
  PRIVATE
  ORGANIZATION @vary(names: ["x-gp-organization", "x-gp-role"])
}

type Report @cacheControl(maxAge: 300, scope: ORGANIZATION) {
  range: ReportRange!
  revenue: Int!
  orders: Int!
}

type Me @cacheControl(maxAge: 60, scope: PRIVATE) {
  id: ID!
  favorites: [Product!]!
}
```

A response that selects a `Report` is stored for five minutes, once per organization and role. A response that selects `Me` is stored once per user. A product list with no named scope is stored once for everyone.

Deploy the configuration first and the schema second, or both together. A `@vary` naming a rule that `gpilot.toml` does not define refuses every request it applies to.

4. Send two members of one organization

```
for token in "$ADA_TOKEN" "$GRACE_TOKEN"; do
  curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
    -H 'content-type: application/json' \
    -H "authorization: Bearer $token" \
    -d '{"query":"query Sales { reports(range: DAY) { revenue orders } }"}' \
    | grep -i '^gp-cache:'
```

```
done
```

With Ada and Grace both in `org_acme` with the role `admin`:

```
gp-cache: MISS
gp-cache: HIT
```

A token for a member of another organization, or for a viewer of `org_acme`, is a `MISS` of its own. A token without `org_id` (a user signed in outside any organization) derives no value for `x-gp-organization`, and the scope then refuses to store the response rather than share it: `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`.

5. Read it in Insights

In Insights, the request's **Cache** tab shows **Segmentation Vary**, split by `x-gp-organization` and `x-gp-role`, each as a keyed pseudonym marked **pseudonymized**. Ada's and Grace's requests show the same two pseudonyms, and **Requests in this bucket** lists everyone who read that entry. The **Vary-segmented** preset finds every request served from an entry of this kind, and **Private responses** the per-user ones.

Pitfalls

- **Organization data selected beside private data is private.** A query that selects `me` and `reports` together is stored per user, the narrowest scope it touched. Split the query if the report should be shared.
- **Revoking a member's access takes effect in the cache only when the entry goes.** A removed member's token stays valid until it expires, and the entries it could read stay cached until they expire or are purged. See revoking access.
- **An Auth0 access token for the wrong API is refused.** A token requested without the `audience` of your API is issued for Auth0's own APIs or as an opaque token, and fails `allowed_audiences` or does not parse as a JWT. Request tokens with `audience=https://shop-api.example.com`.

Related

- Partitioning your cache
- Scopes in the schema
- Authentication at the edge
- Revoking access

GUIDES

Purging from a CMS webhook

Goal: product descriptions live in a headless CMS. When an editor publishes a change, the product pages that show it are cleared within seconds, without waiting for their lifetime to run out.

A publish in the CMS never passes through GraphPilot, so automatic invalidation does not see it. The CMS has to say what changed, and a small handler turns that into a purge.

The running example is `shop-api`. `Product` carries `@surrogateKey(of: "id")`, so every cached response that shows product `p01` is tagged `Product:id:p01`. The CMS stores each product's copy under the same id and calls a webhook on every publish with a body such as `{"entry": {"productId": "p01"}}`.

What you need

- An organization API key with the purge permission and nothing else, created in the portal under **Settings → API keys** (see API keys and permissions). Store it where the handler runs, for example as `GRAPHPILOT_API_KEY`.
- A place to run the handler: a serverless function, or a route in an app you already run.
- A CMS that can call a webhook on publish.

1. Find the key the entries carry

Read the key off a real response rather than building it by hand. With `response.surrogate_key_header` switched on:

```
curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"query ProductPage { product(id: \"p01\") { id name } }"}' \
  | grep -i '^gp-surrogate-key:'
```

```
gp-surrogate-key: ck:… q:product Product Query catalogue Product:category.id:audio
Product:id:p01
```

`Product:id:p01` is the key to purge when the copy of `p01` changes.

2. Write the handler

purge-on-publish.ts

```
export async function handlePublish(request: Request): Promise<Response> {
  // Check the CMS's webhook signature here, before anything else.
  const { entry } = await request.json();

  const response = await fetch("https://api.graphpilot.io/purge", {
    method: "POST",
    headers: {
      authorization: `Bearer ${process.env.GRAPHPILOT_API_KEY}`,
      "content-type": "application/json",
    },
    body: JSON.stringify({
```

```
      serviceId: "shop-api",
      keys: [`Product:id:${entry.productId}`],
      soft: true,
    }),
  });

  // Let the CMS retry on a failure: purging the same key again is safe.
  return new Response(null, { status: response.ok ? 204 : 502 });
}
```

`serviceId` takes the service's subdomain. `soft: true` marks the entries stale instead of removing them: with the stale window `Product` has, readers keep getting the old copy for the moment it takes to fetch the new one. Use a hard purge (leave `soft` out) when the old copy must not be shown again, such as a wrong price.

A purge that ran answers `200` with the keys it covered:

```
{ "success": true, "purgedKeyCount": 1, "purgedKeys": ["Product:id:p01"] }
```

`purgedKeyCount` counts keys, not cached responses: a key nothing was stored under still counts. A refused purge answers with a status that matches its `extensions.code`; the list is under over REST.

> ⚠ **Verify the webhook first**
>
> The handler holds a key that can purge your service. Check the CMS's webhook signature or shared secret before calling GraphPilot, or anyone who finds the URL can empty your cache, one origin request per entry at a time.

3. Publish and check

Publish a change to `p01` in the CMS, then ask for the product page again:

```
curl -sS -D- -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"query ProductPage { product(id: \"p01\") { id name } }"}' \
  | grep -i '^gp-cache:'
```

After a soft purge the first answer is `gp-cache: STALE`, served from the old copy while GraphPilot fetches the new one; the next is a `HIT` with the new copy. After a hard purge it is a `MISS`.

4. Find it in the portal

The service's **Purges** page lists the purge with the trigger **Manual** and the outcome *Marked 1 key stale*. **Actor** is empty, because an API key sent it. To find the purges of one product, filter by **Key** with `Product:id:p01`. A purge takes effect at once and can take up to a minute to appear in the list.

Pitfalls

- **One product's key does not reach everything that mentions it.** A purge matches keys exactly, type name included. A cached review list tagged `Review:productId:p01` is not reached by `Product:id:p01`; name both when the CMS change affects both.
- **A new product does not appear in a cached list on its own.** No cached list contains it yet, so its key clears nothing. Purge a key the list carries as well; see lists and creates.
- **Bulk publishes need batching.** One call takes at most 1000 keys. Split a larger import into several calls rather than purging everything.
- **No soft purge for everything.** `{"all": true}` is always hard and sends every next request to your origin. Prefer keys.

Related

- Purging through the API and CLI
- Surrogate keys
- Verifying a purge
- Purges in the portal

GUIDES

Telling your clients apart

Every request comes from something: a web app, a mobile build from eighteen months ago, a partner's integration, a script somebody wrote once. By default GraphPilot serves them all alike and reports them as one stream. A **client profile** gives one caller a name. With a name, you can find it in Insights, hold it to a minimum version, give it its own limits, or refuse it.

This page owns the rules for client profiles; the configuration reference lists the keys.

The running example is a service with three callers: your own web app, which can send a header you agree on; the `gpilot` CLI, which cannot; and the partner Acme, whose token your identity provider issues with `client_id: acme`.

What you need

- A deployed service and its `gpilot.toml`.
- For a profile matched on a verified claim, an `[auth.providers.<name>]` block; see authentication.

1. Name your own app

gpilot.toml

```
[[client.profile]]
```

```
name    = "web-app"
when    = "header:gp-client-name == 'web-app'"
version = "header:gp-client-version"
```

`when` decides whether a request is that client, and is read as a truth. `version` says where its version comes from, and is read as a value. Both are policy expressions, so a header, a cookie, a query parameter, a verified claim or a piece of the `User-Agent` all work. `gp-client-name` and `gp-client-version` are read by expressions but removed before your origin.

A client is declared, not discovered. The name recorded is the one you wrote, never one the caller sent, so `web-app` does not also arrive as `webapp` and `WebApp`. Profiles are tried in file order and the first whose `when` is true wins, so put the narrower profile first.

2. Recognize clients that cannot cooperate

A CLI you did not write, or a mobile build that shipped years ago, cannot be asked to send a new header. Recognize it by what it already sends:

gpilot.toml

```
[[client.profile]]
name    = "gp-cli"
when    = "starts_with(header:user-agent, 'gp-cli/')"
version = "split(header:user-agent, '/', 1)"
```

See text functions.

> ⚠ **A header is not a security boundary**
>
> A profile matched on a header trusts what the caller sent, and anyone who can read your public app's requests can send the same bytes. Against stale deployments and misrouted traffic it works; against someone trying to get past it, it does not. Where a decision has to hold, match on a verified claim.

3. Match a partner on a verified claim

gpilot.toml

```
[auth.providers.idp]
type               = "jwt"
allowed_algorithms = ["RS256"]
jwks_url           = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers    = ["https://idp.example.com/"]
allowed_audiences  = ["my-api"]

[[client.profile]]
name    = "partner-acme"
when    = "auth:idp.client_id == 'acme'"                    # who: from the token
```

```
version = "auth:idp.client_version || header:gp-client-version" # which build
```

`auth:idp.client_id` reads a claim the `idp` provider verified before any expression ran, so the caller cannot choose it. Client profiles are resolved after authentication for exactly this. The claim's name is your identity provider's: `client_id` or `azp` in a client-credentials flow, or any claim it mints. The version is read from the token when present and from the header otherwise, because `||` yields the first side that resolves.

What a claim changes:

Setting	Matched on a header	Matched on a claim
`unknown = "reject"`	Catches stale deployments and misrouted traffic	Refuses anyone who did not authenticate as a declared client
`min_version`	Retires an old build	The same, and the build cannot dodge the floor by claiming to be another client
`cache = "off"`, `max_age`	A switch a stranger can flip, adding origin load	A switch only that client can flip

Nobody can inflate another client's numbers by copying its header, either. The version stays self-reported while it comes from a header, so a client can still over-report it to clear a floor.

4. Deploy and look

Deploy with `gpilot deploy`, leave `client.unknown` at `"allow"`, and look at what matched. In Insights, each request's **Client** tab shows the **Recognized client** and its version, or that no profile matched, on every plan. From Jet (Starter), the request list can show client name and version as columns. Insights has no aggregated per-client view. The requests that match no profile are the callers you have not described; describe them as in steps 1 to 3.

5. Retire old versions

gpilot.toml

```
[[client.profile]]
name        = "web-app"
when        = "header:gp-client-name == 'web-app'"
version     = "header:gp-client-version"
min_version = "2.1.0"
```

Anything below the floor is refused with `403` and `CLIENT_VERSION_UNSUPPORTED`, with the required version in the message. Versions compare properly: `4.10.0` is newer than `4.9.0`, and `4.2.0+build.7` is the same release as `4.2.0`. A client that reports no version, or one that does not parse, is refused. A `min_version` GraphPilot cannot parse refuses nothing and logs `CLIENT_MIN_VERSION_UNREADABLE`, so a typo costs you the floor, not your traffic.

6. Give a client its own limits

A profile can tighten what the service allows:

Key	Gives that client
`max_body`	A smaller request body limit
`max_query_depth`	A shallower nesting ceiling
`max_document_bytes`	A shorter document limit
`max_selections`	A lower selection limit
`persisted_query = "required"`	Hashes only, with no document in the payload

Each sits under the service-wide setting of the same name, and the stricter one decides; a looser per-client value raises nothing. For plain HTTP requests, only `max_body` applies; the other four need a GraphQL operation. `max_query_depth`, `max_document_bytes` and `max_selections` are not checked on a cache hit, which parses nothing and costs your origin nothing.

`persisted_query = "required"` refuses the APQ registration handshake, because that payload carries a document, so such a client's operations have to be pushed with `gpilot pq push`. It does not apply to subscriptions: operations on a WebSocket travel inside the socket, where the edge never reads them.

None of these limits, and none of the refusals in steps 5 and 7, applies to a browser's CORS preflight: an `OPTIONS` request with an `Access-Control-Request-Method` header and no body. A preflight carries none of the headers a profile is matched on and runs no operation, so refusing it would block every cross-origin request of the very client the profile describes. The request that follows the preflight is checked as usual, and an `OPTIONS` request with a body is checked like any other request.

Two keys relax instead: `cache = "off"` takes the client off the cache, and `max_age` makes it refuse entries older than a number of seconds. Both hand work to your origin, so match such a profile on a verified claim.

7. Refuse what you did not declare

Last, once Insights shows nothing legitimate left unnamed:

gpilot.toml

```
[client]
unknown = "reject"
```

A request that matches no profile is refused with `403` and `UNKNOWN_CLIENT`, before the cache and before your origin. A CORS preflight is the exception, see step 6.

Check that it works

```
curl -sS -o/dev/null -w '%{http_code}\n' "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -H 'gp-client-name: web-app' -H 'gp-client-version: 2.0.0' \
  -d '{"query":"{ __typename }"}'
```

`403`, and the body's `extensions.code` is `CLIENT_VERSION_UNSUPPORTED`. With `gp-client-version: 2.1.0` it passes, and Insights shows **Recognized client** `web-app` with version `2.1.0`. With no `gp-client-name` at all it is `403` with `UNKNOWN_CLIENT` once step 7 is deployed. `UNKNOWN_CLIENT`, `CLIENT_VERSION_UNSUPPORTED` and `PERSISTED_QUERY_REQUIRED` reach the caller; a body over `max_body` reaches it as `PAYLOAD_TOO_LARGE`, and only the **Edge codes** in the request's **Overview** tab say `CLIENT_BODY_TOO_LARGE` and name the client. See error and log codes.

Pitfalls

- **A claim-matched profile plus `unknown = "reject"` means every request must authenticate.** A request without a token carries no claims, so it matches no claim-based profile. Give anonymous callers a profile of their own, or leave `unknown` at `"allow"`.
- `unknown = "reject"` **also applies to WebSocket handshakes.** A browser cannot set `gp-client-name` on a handshake, so a profile matched only on that header refuses your own subscriptions with `UNKNOWN_CLIENT`. Match the handshake on a cookie, a query parameter or the `User-Agent`, or give it a profile of its own; see subscriptions and WebSockets.
- **A missing claim is not a mismatch, it is nothing.** A token without `client_id` leaves the comparison undecided, and `when` reads that as *not this client*; see a request the expression cannot decide.
- **Never write a version floor with `>=`.** Ordering operators compare numbers, `4.2.0` is not one, and the profile silently matches nothing. Use `min_version`, or `semver_gte` inside an expression.
- **Every client shares one cache.** A profile decides whether a request is served and how old an entry it accepts, never what the response contains. A client over its `max_age` goes to your origin and reports `PASS` with `CACHE_SKIPPED_ENTRY_NOT_REPLACED`; the entry stays as it is for everyone else. Different content per client needs a vary rule, at its hit-rate cost.
- **Some combinations refuse the deployment:** `min_version` without `version`; `max_query_depth = 0`, `max_document_bytes = 0` or `max_selections = 0`, which would refuse everything (leave the key out for no ceiling); and `max_age` with `cache = "off"`.
- `CLIENT_MIN_VERSION_UNREADABLE` **and** `CLIENT_PERSISTED_QUERY_UNENFORCEABLE` in your logs mean a setting is being ignored. Fix the configuration.

Related

- Configuration reference: `client`
- Policy expressions
- Authentication
- Insights

GUIDES

Retiring old mobile app versions

Goal: builds of your iOS app older than 4.2.0 call a field your API no longer serves well. Refuse them at the edge with a clear error, so the app can tell the user to update, and leave every newer build and every other client alone.

The rules for client profiles are on telling your clients apart; this recipe applies them to one case, a mobile app that already ships.

The running example is `shop-api` and its iOS app, built with Apollo iOS, which sends `apollographql-client-name: ios-app` and `apollographql-client-version` on every request. The build pipeline appends a build number to the version: build 118 of 4.2.0 sends `4.2.0-118`.

What you need

- A deployed service and its `gpilot.toml`.
- An app that sends its name and version on every request. Apollo iOS sends the two `apollographql-client-*` headers; any other client can be set up to send them, or `gp-client-name` and `gp-client-version`.

1. Declare the app and its floor

gpilot.toml

```
[[client.profile]]
name        = "ios-app"
when        = "header:apollographql-client-name == 'ios-app'"
version     = "split(header:apollographql-client-version, '-', 0)"
min_version = "4.2.0"
```

`version` takes the part before the first `-`. That is not cosmetic: versions are compared as semantic versions, where `4.2.0-118` is a prerelease of 4.2.0 and sorts **below** it. Without the `split`, every build of 4.2.0 itself would be refused. With it, `4.2.0-118` is read as `4.2.0` and clears the floor.

A request from the app below 4.2.0 is refused with `403` and `CLIENT_VERSION_UNSUPPORTED`, without reaching your origin, with the required version in the message. A request that matches no profile (your web app, a partner, a script) is not affected, because `client.unknown` stays at `"allow"`.

> ⚠ **A header is not a security boundary**
>
> Anyone can send `apollographql-client-name: ios-app` with any version. The floor retires old builds; it does not keep out someone who wants past it. Where it has to hold, match the profile on a verified claim, as in match a partner on a verified claim.

2. Deploy and try an old build

```
curl -sS "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -H 'apollographql-client-name: ios-app' \
  -H 'apollographql-client-version: 4.1.3-97' \
  -d '{"query":"query ProductPage { product(id: \"p01\") { id name price } }"}'
```

The answer is `403`, and the body's `extensions.code` is `CLIENT_VERSION_UNSUPPORTED`. Send `4.2.0-118` instead and the request is served as before.

Handle the code in the app: when a response carries `CLIENT_VERSION_UNSUPPORTED`, show the update prompt instead of a generic error. The code is stable; the message text is for people.

3. Read it in Insights

Before you raise the floor, look at who it would refuse. In Insights, each request's **Client** tab shows **Recognized client** `ios-app` and the version the profile read, so `4.2.0` for build `4.2.0-118`. From Jet (Starter), the request list can show client name and version as columns, which is the quickest way to see which versions still call.

After the deployment, the refused requests are in the list with status `403`. The **Auth failures** preset (under **Errors**) lists requests answered `401` or `403`.

Pitfalls

- **Three-part versions only.** A version the profile cannot parse as a semantic version is refused, like one below the floor. An app that reports `4.2` is refused; make it report `4.2.0`. A `min_version` that does not parse refuses nothing and logs `CLIENT_MIN_VERSION_UNREADABLE`.
- **Never write the floor with `≥`.** `header:apollographql-client-version ≥ '4.2.0'` compares numbers, and a version is not one. Use `min_version`, or `semver_gte` in an expression; see versions.
- **A `+` build suffix needs no `split`.** `4.2.0+118` is build metadata, and `4.2.0+118` is the same release as `4.2.0`. Only a `-` suffix reads as a prerelease.
- **Raising the floor does not empty the cache.** It refuses requests; the answers newer builds read stay cached as before.

Related

- Telling your clients apart
- Configuration reference: `client`
- Error and log codes

GUIDES

Migrating to trusted documents

> **GraphQL only** A plain HTTP API has no operations to register; its paths are the allowlist.

With trusted documents, your service runs only the operations you registered, and refuses every other one, however valid. An attacker cannot craft a new query, and you know exactly what runs against your API. Switching it on in one step breaks every client that sends an operation you forgot. This guide gets there in stages, so the switch happens against evidence. How persisted queries work is on persisted queries.

The running example is a web app built with graphql-codegen, which writes `dist/persisted-documents.json`, and a mobile app released every two weeks.

What you need

- Clients that send the Apollo-style `extensions.persistedQuery` hash; see how a client sends one.
- A manifest from your client build, and a CI job that can run `gpilot pq push`; see deploy from CI.

1. Push the manifest before every client release

Register the operations of each client build before that build reaches users:

```
gpilot pq push ./dist/persisted-documents.json --dry-run   # parse and verify only
gpilot pq push ./dist/persisted-documents.json
```

A push only adds, so the operations of a release still in the field keep working. Every entry is checked: its id must be the SHA-256 of its document, and a mismatch is reported with the one configuration line that fixes it. Formats and flags are on registering your operations.

2. Run in audit mode

Let everything run, and record what is not registered:

gpilot.toml

```
[persisted_query]
preset = "trusted_migrating"
```

That preset resolves hashes, lets clients register operations through the APQ handshake, and logs every operation the registry does not hold as `PERSISTED_QUERY_UNTRUSTED_OPERATION`, with its hash and operation name. Nothing is refused yet. If clients should not register anything while you watch, turn the handshake off beside the preset:

gpilot.toml

```
[persisted_query]
```

```
preset = "trusted_migrating"
apq    = false
```

Deploy with `gpilot deploy`.

3. Collect what is missing

Two places show what `trusted` would refuse:

- **The edge codes.** Each request that ran an unregistered operation carries `PERSISTED_QUERY_UNTRUSTED_OPERATION` under **Edge codes** in its **Overview** tab in Insights.
- **The APQ list.** `gpilot apq ls`, or the **APQ** tab on the portal's Queries page, lists the operations clients registered themselves that your manifest does not declare. Read one with `gpilot apq show <hash>`.

For each one, find the client code that sends it and add it to that client's manifest, or decide it should stop running. Keep watching until a full release cycle passes with nothing new; for the mobile app, that means until the oldest version still in use has sent everything it sends.

4. Fold it in and clean up

Push the updated manifests, then remove what clients registered on their own:

```
gpilot pq push ./dist/persisted-documents.json
gpilot apq rm --all
```

`apq rm` revokes those entries and purges everything cached for them.

5. Switch to trusted

gpilot.toml

```
[persisted_query]
preset = "trusted"
```

`trusted` resolves hashes, refuses the handshake and refuses anything the registry does not hold. Deploy.

Check that it works

A registered operation still answers:

```
curl -sS "https://$GRAPHPILOT_HOST/graphql" -H 'content-type: application/json' \
  -d '{"extensions":{"persistedQuery":{"version":1,"sha256Hash":"<a hash from your
manifest>"}}}'
```

A query nobody registered is refused with `403` and `FORBIDDEN`:

```
curl -sS -o/dev/null -w '%{http_code}\n' "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' -d '{"query":"{ __typename }"}'
```

The refused request's log line is `PERSISTED_QUERY_REFUSED`. Every answer the edge can give is on codes.

Pitfalls

- **Cached responses outlive the switch.** The allowlist is checked on a miss, not on a hit, so revoke with `gpilot pq rm` or `gpilot apq rm`, which purge, what must stop at once; see choosing a setup.
- `only_registered` **and** `apq` **together are refused** at deploy time. An allowlist anyone can add to is not one. `trusted` already has `apq` off; do not turn it back on.
- **Ad-hoc queries stop too.** Your own tooling, a GraphiQL session and a partner's scripts are refused unless their operations are registered. Put them on a separate service if they need to stay open.
- **A client released before its push** sends hashes the registry cannot resolve, and under `trusted` that is a `403`. Push in the same pipeline, before the release.
- **Subscriptions are not covered.** Operations sent over a WebSocket travel inside the tunnel, where the edge never reads them; see subscriptions and WebSockets.
- **A client profile can require hashes.** A client profile with `persisted_query = "required"` refuses the APQ handshake too, so its operations have to be pushed; see telling your clients apart.

Related

- Persisted queries: configuration, presets and limits
- Queries in the portal
- `gpilot pq` and `gpilot apq`
- Hardening a public API

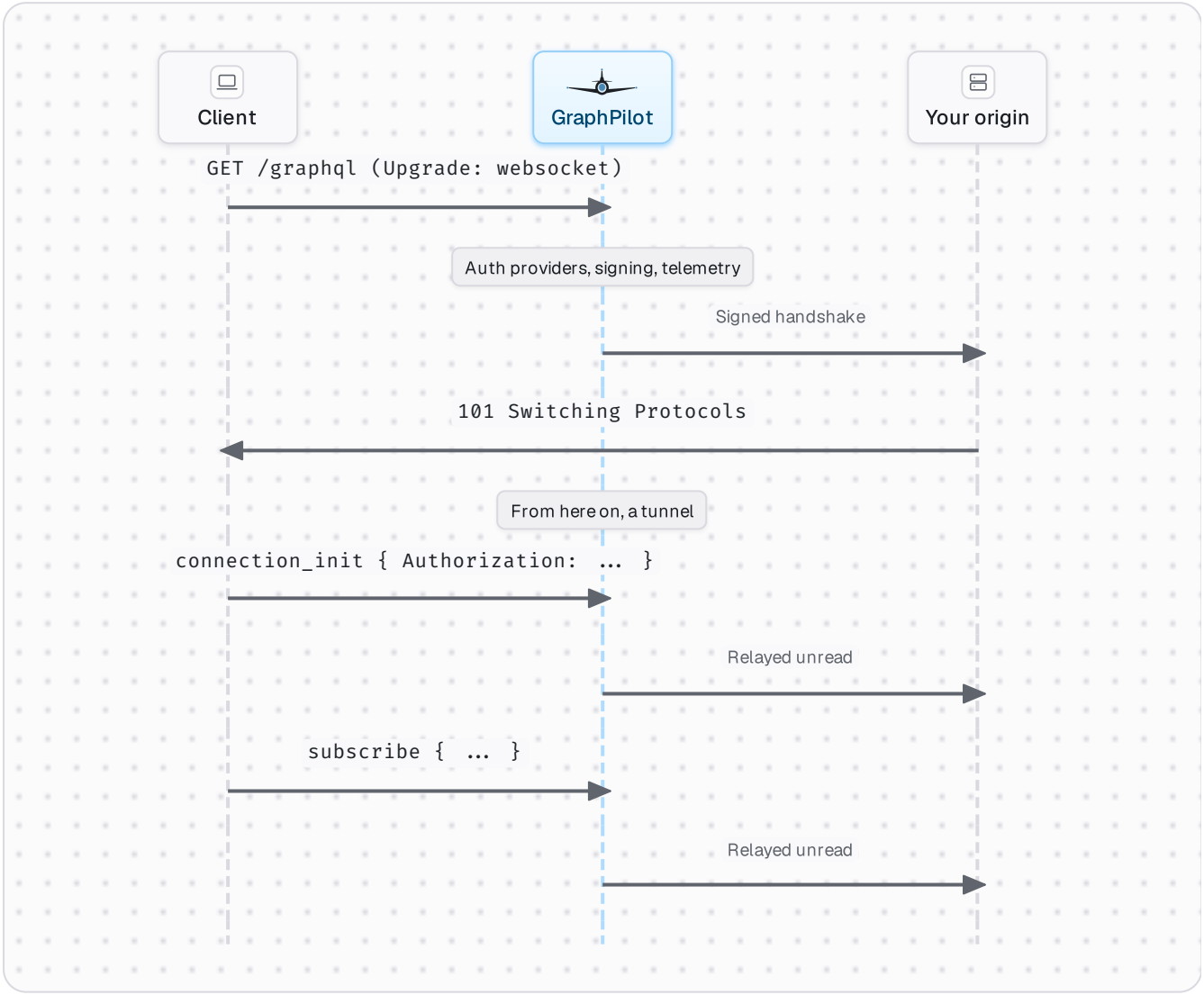
GUIDES

Subscriptions and WebSockets

A GraphQL subscription runs over a WebSocket, and a WebSocket is one HTTP request followed by a long conversation that is no longer HTTP. GraphPilot treats those two halves differently, and the difference decides what your origin has to do for itself.

The handshake is an ordinary request and goes through the whole edge pipeline. Everything after it is a **tunnel**: GraphPilot relays frames between your client and your origin without reading them.

The running example is a GraphQL API that serves subscriptions with `graphql-transport-ws` on `/graphql`, used by a browser app that sends its token in `connection_init`. A WebSocket to a plain HTTP origin goes through the same handshake and the same tunnel; only the GraphQL parts of steps 3 and 5 are specific to subscriptions.



What you need

- An origin that accepts WebSocket upgrades on the path your clients connect to.
- Your service's signing key, to verify the handshake; see the signing key.
- A WebSocket client for the check, such as `wscat` (`npx wscat`).

1. Know where the edge stops

GraphPilot recognizes a handshake by the rules of RFC 6455: a `GET` carrying `Upgrade: websocket` and a `Connection` header whose token list includes `upgrade`. Such a request

- passes your auth providers, so an `Authorization` header sent on the handshake is verified, and a token that does not verify is rejected with **401** before your origin ever sees it (the default; see `[auth.rejection]`),

- is signed like every other forwarded request, with `GP-Timestamp` and `GP-Signature`,
- is matched against your client profiles. With `client.unknown = "reject"`, a handshake that matches no profile is refused with **403** and `UNKNOWN_CLIENT`, and one below a profile's `min_version` with `CLIENT_VERSION_UNSUPPORTED`. A browser cannot set headers on a handshake, so match browser clients on something it does send, such as a cookie, a query parameter or the `User-Agent`,
- never touches the cache. An upgrade is always a pass, reported as `CACHE_PASS_WEBSOCKET_UPGRADE`,
- is never retried: a handshake that fails reaches the client as a `502`,
- appears once in your telemetry, with status `101`.

GraphPilot then hands the connection to your origin and stops interpreting it. Inside the connection:

Edge feature	Inside the tunnel
Authentication	Handshake only. A credential sent as a frame is never seen, so nothing at the edge can verify it.
Cache policy, surrogate keys, invalidation	Do not apply. Nothing that flows through a tunnel is cacheable.
Request signing	Covers the handshake. Individual frames are not signed, and the connection outlives the signature's timestamp.
Persisted queries	Enforced on HTTP GraphQL requests. An operation sent as a frame is not matched against the registry, so a "persisted queries only" configuration does not constrain what a subscription may execute.
Query depth, document size, selections, introspection block	Do not apply. An operation sent as a frame is never parsed, so `request.max_query_depth`, `max_document_bytes`, `max_selections` and `introspection` do not limit it, and neither does a client profile's lower value.
Request size	Covers the handshake, which has no body. Frames have no size limit at the edge, and a connection has no limit on how many it carries.
Telemetry	One event, for the handshake. Operations and messages inside the connection are not recorded.

This is not a gap to be closed later. A tunnel is what makes a WebSocket work through a CDN at all: the edge cannot buffer, parse or replay a bidirectional stream without breaking the protocol it is carrying.

No setting turns WebSocket upgrades off at the edge. A handshake on any path, GraphQL or not, that passes the checks above, client profiles included, is handed to your origin, even on a service that requires persisted queries or blocks introspection. If your API serves no subscriptions, have your origin refuse the upgrade.

2. Verify the signature on the handshake

`GP-Signature` proves the connection was opened through GraphPilot rather than directly against your origin. Verify it on the upgrade request as on any other; see verify the signature on your origin. It says nothing about any frame that follows.

3. Verify the credential from `connection_init` yourself

> ⚠ **The edge cannot verify it**
>
> A token sent in the `connection_init` payload is **not** verified by GraphPilot. It arrives after the handshake, as a frame, so a forged or expired token is indistinguishable from a valid one until your origin checks it.

Browsers cannot set request headers on a WebSocket handshake. That is why clients speaking `graphql-transport-ws` carry the credential in the `connection_init` payload instead, the first message after the `101`. No auth provider reads it, and the rejection behavior documented for HTTP requests does not apply to it.

Check the signature against your issuer's key set, the expiry, the issuer, the audience and whatever claims you authorize on. Treat the connection as anonymous until that check passes, and refuse subscriptions on an unauthenticated connection rather than letting them fall through to a default identity.

4. Re-check expiry for the life of the connection

A subscription can stay open for hours, far longer than an access token stays valid. Without a periodic check, one token accepted at minute zero authorizes the stream indefinitely. Close the connection when the token expires, or require the client to re-authenticate. `4401` is the close code `graphql-ws` clients conventionally read as an authentication failure.

5. Apply your own limits to the connection

The protections you configure at the edge do not reach inside the tunnel, so the limits on a subscription are the ones your origin sets:

- **Message size.** A maximum frame or message size on your WebSocket server, for example the `maxPayload` option of the `ws` library. A message over it closes the connection.
- **Operation shape.** The same depth, alias or field count and cost limits your GraphQL server applies over HTTP, applied to every operation that arrives as a `subscribe` message. The edge defaults (depth 10, 4,000 selections, a 64 KiB document) are a reasonable starting point.
- **Introspection.** Off in production, as it is at the edge by default; a socket is a way around the edge's block.
- **Volume.** Subscriptions per connection, connections per credential and messages per second.

The full list of what the edge does not check, over HTTP and over a WebSocket, is on hardening a public API.

6. Send the credential on the handshake too, where you can

Clients that can set headers, such as server-to-server consumers and CLIs, may send `Authorization` on the upgrade request. The edge then verifies it like any other request and rejects a bad token with 401 before your origin is involved. That is a useful additional filter, and it does not replace step 3: as long as any browser client connects to the same endpoint, your origin still has to authenticate the connection itself.

Check that it works

Open a connection through GraphPilot and send a `connection_init` with a token that is not valid:

```
npx wscat -c "wss://$GRAPHPILOT_HOST/graphql" -s graphql-transport-ws
> {"type":"connection_init","payload":{"Authorization":"Bearer not-a-token"}}
```

Your origin should close the connection, with `4401` if it follows the `graphql-ws` convention. With a valid token it answers `{"type":"connection_ack"}`. In Insights, the handshake appears once with status `101`, and nothing after it. A connection opened straight to your origin, without `GP-Signature`, should be refused at step 2.

Pitfalls

- **A `101` in your telemetry is not your origin's answer.** The handshake is recorded as `101` as soon as GraphPilot has handed it to your origin, even if your origin then refuses the upgrade. Only a handoff that fails is recorded differently, as a bad gateway error. Your origin's logs say whether it accepted the connection.
- **Subscriptions over HTTP do not work through the edge.** GraphPilot delivers an origin response once it is complete, and a Server-Sent Events or `multipart/mixed` response never completes. Use WebSockets for subscriptions.
- **A persisted-queries-only setup does not cover subscriptions**, and neither does a client profile's `persisted_query = "required"`.
- **GraphPilot secures the door, not the room behind it.** Every guarantee about the operations running inside a subscription has to come from your origin.

Related

- Authentication at the edge
- Request signing
- Persisted queries
- Hardening a public API

GUIDES

Promoting from staging in CI

Goal: every merge to `main` goes live on staging by itself, is checked with a real request, and reaches production only after somebody approves it. The same commit is deployed to both, so production gets exactly what staging was tested with.

GraphPilot has no promotion between services: a deployment builds from the files in front of it. So “promote” means deploying the same files from the same commit to a second service, and the approval lives in your CI. The pieces are on deploy from CI and staging and rollback; this recipe puts them into one workflow.

The running example is the service pair `shop-api-staging` and `shop-api`, deployed from one repository that holds `schema.graphql`, `gpilot.toml`, and the client’s persisted query manifest at `dist/persisted-documents.json`.

What you need

- Both services, created as on staging and rollback.
- An organization API key as the repository secret `GPILOT_TOKEN`, with the permissions to deploy and to push persisted queries to both services. See jobs.
- A GitHub environment called `production` with required reviewers. That is the approval gate.

1. The workflow

.github/workflows/graphpilot.yml

```yaml
name: GraphPilot

on:
  pull_request:
  push:
    branches: [main]

concurrency:
  group: graphpilot-shop-api
  cancel-in-progress: false

env:
  GPILOT_TOKEN: ${{ secrets.GPILOT_TOKEN }}

jobs:
  check:
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - name: Install the gpilot CLI
        run: |
          curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
          echo "$HOME/.local/bin" >> "$GITHUB_PATH"
      - name: Check the persisted query manifest
        run: gpilot pq push ./dist/persisted-documents.json --dry-run --service shop-api

  staging:
    if: github.event_name == 'push'
    runs-on: ubuntu-latest
```

```
    steps:
      - uses: actions/checkout@v5
      - name: Install the gpilot CLI
        run: |
          curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
          echo "$HOME/.local/bin" >> "$GITHUB_PATH"
      - name: Deploy staging
        run: gpilot deploy --service shop-api-staging --yes
      - name: Push persisted queries
        run: gpilot pq push ./dist/persisted-documents.json --service shop-api-staging
      - name: Smoke test
        run: ./scripts/smoke.sh https://shop-api-staging.k7m2qp.graphpilot.cloud

  production:
    needs: staging
    runs-on: ubuntu-latest
    environment: production
    steps:
      - uses: actions/checkout@v5
      - name: Install the gpilot CLI
        run: |
          curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
          echo "$HOME/.local/bin" >> "$GITHUB_PATH"
      - name: Deploy production
        run: gpilot deploy --service shop-api --yes
      - name: Push persisted queries
        run: gpilot pq push ./dist/persisted-documents.json --service shop-api
      - name: Smoke test
        run: ./scripts/smoke.sh https://shop-api.k7m2qp.graphpilot.cloud
```

Replace `k7m2qp` with your organization's hostname label.

- `check` runs on every pull request. `--dry-run` parses and verifies the manifest and writes nothing, so a broken manifest fails the pull request instead of the release.
- `staging` deploys on every merge. `gpilot deploy` waits for the deployment and exits `0` only when it completed, so a failed build stops the workflow before the smoke test. The exit codes say why.
- `production` waits for `staging`, then for a reviewer of the `production` environment. It checks out the same commit, so it deploys the same files staging was tested with.
- **Persisted queries are pushed to each service** before your client release, so no client ever sends a hash the registry cannot resolve. `pq push` only adds, so pushing a manifest twice is harmless. Release the client after the `production` job, never before it.

2. The smoke test

scripts/smoke.sh

```
#!/usr/bin/env bash
```

```
set -euo pipefail
host="$1"

body=$(curl -sS --fail-with-body "$host/graphql" \
  -H 'content-type: application/json' \
  -H "GP-Correlation-ID: smoke-$GITHUB_SHA" \
  -d '{"query":"query Smoke { products(first: 1) { id name } }"}')

echo "$body" | jq -e '.errors == null and (.data.products | length) == 1'
```

`--fail-with-body` fails on an HTTP error status, and `jq -e` fails on a GraphQL error, which arrives with `200`. Pick a query that exercises your origin and your schema together: a broken origin fails it, and so does a schema whose new field your origin does not know yet.

The `GP-Correlation-ID` header labels the request. On Airliner (Business) and above, add a step that waits for it in telemetry, which proves the request was recorded on the service you meant:

```
gpilot requests --service shop-api-staging --correlation-id "smoke-$GITHUB_SHA" \
  --wait --timeout 60s -o json | jq -e '.[0].status == 200'
```

3. What you see

- **In GitHub**, a merge shows `staging` green and `production` waiting for review. Approving it runs the production deployment.
- **In the portal**, each service's **Deployments** list shows the new deployment with the API key's name as who made it, and its findings. Name the key after the pipeline so the list reads well.
- **In Insights**, the smoke request is in each service's request list with its correlation ID on the **Overview** tab of the request detail.
- **From a terminal**, `gpilot status --service shop-api` lists the newest deployment as `completed`.

Pitfalls

- **One key for both services is one key that can deploy production.** If the approval must not be bypassable, use two keys, and make the production key an environment secret of `production` rather than a repository secret.
- **The cache survives the deployment.** A `HIT` right after the deploy is an entry stored before it. If the change made cached entries wrong, purge them in the production job; see purge after deploying.
- **Staging's cache says nothing about production's.** The two services share nothing: cache, keys and telemetry are separate.
- **Deploy the schema before the origin.** If the same workflow releases your origin, run the GraphPilot deployment first; see deploy the schema before the origin.

Related

- Deploy from CI
- Staging and rollback
- Migrating to trusted documents
- CLI command reference

GUIDES

Triaging an alert

Goal: an alert email arrives. Within a few minutes you know whether the problem is your origin, your last deployment, or GraphPilot, and you have either fixed it or rolled back.

What each rule watches and who is emailed is on alerts. This recipe walks one incident from the email to the end.

The running example is `shop-api`. A deployment went out at 14:02. At 14:12 the Admins and Members get an email headed **Your origin returned server errors for shop-api**.

What you need

- The `gpilot` CLI, logged in, or the portal.
- `service:view` to read Insights and the Alerts page, and `deployment:edit` to roll back.

1. Read the email

The email names the rule, the service and the measurement, for example the share of requests your origin answered with a server error over the last five minutes. Which rule fired already narrows the search:

Rule in the subject	Where to look first
GraphPilot returned errors	GraphPilot's status page, linked in the email. Requests fail before they reach your origin; nothing to fix on your side.
Your origin did not respond	Your origin's reachability: a firewall rule, a rate limit, an expired certificate or a DNS change.
Your origin returned server errors	Your application and whatever sits directly in front of it.
GraphQL errors in your responses	The errors your origin returned inside `200` responses.
Cache hit rate dropped	The skip reason the email names, and the last deployment's caching changes.
Traffic stopped	Your clients and the DNS record of the service.

Open this incident leads to the service's **Alerts** page, where the incident reads **Ongoing** with the time it started.

2. Find the failing requests

In Insights, set the time range to the last 30 minutes and apply the **Server errors** preset, under **Errors**. Each row shows the status your origin returned, the operation and the path. Open one: the **Overview** tab says whether the request reached your origin and with which status, and the **GraphQL** tab shows the document your origin received.

The same from a terminal:

```
gpilot requests --service shop-api --status 500 --status 502 --status 503 --since 30m --wide
```

`--wide` adds the request id, the cache reason, the cache key and the surrogate keys. To follow a request into your own logs, see correlating with your logs.

Three questions decide what to do next:

- **Did it start with the deployment?** Compare the first failing request with the deployment's time on the **Deployments** page. Insights can also filter by deployment (the **Deployment** group in the **Filters** panel).
- **Is it one operation or all of them?** One operation points at a resolver or a schema change. All of them point at the origin itself.
- **Did the origin answer at all?** An origin status in the request detail means your application answered. No origin status means the request never got an answer, which is the territory of **Your origin did not respond**.

3. Roll back if the release caused it

If the failures start with the deployment, roll back before you debug:

1. Open **Deployments** in the portal and the deployment before the bad one.
2. Choose **Redeploy**. The dialog starts with that deployment's schema and configuration.
3. Confirm. The previous version keeps serving until the new deployment is live.

From the CLI, deploy the previous files again; see roll back to a deployment that worked. If the bad version stored wrong answers, purge them once the rollback is live, because the cache survives a deployment.

If your origin release, not the GraphPilot deployment, caused it, roll back the origin; nothing on the GraphPilot side needs to change.

4. Watch it close

When the condition has not been seen for fifteen minutes, the incident closes and a second email says so, about fifteen to twenty minutes after the last failing minute. The incident then shows its duration on the **Alerts** page. A condition that comes back after that is a new incident with a new email.

Pitfalls

- **Only Admins and Members are emailed.** A Developer on call gets nothing. Give the people who should be woken the Member role; see the emails.
- **Alerts see requests Insights does not.** Alerts count every request, including those past your logged-request allowance and those left out by `telemetry.record`. If Insights shows fewer failures than the email, that is why.
- **Do not switch the rule off to stop the emails.** An incident is one email at the start and one at the end. Switch **Traffic stopped** off for planned maintenance, not the error rules.

Related

- Alerts
- Insights
- Staging and rollback
- Troubleshooting

GUIDES

Keeping request telemetry minimal

Goal: your data protection review asks for request telemetry that keeps only what you need to operate the service. You want Insights to keep working (status, latency, cache decisions, operations), without full client addresses, credentials or health-check noise.

Everything here is per service in `gpilot.toml`, and none of it changes how requests are served, cached or billed. Only what is stored changes.

The running example is `shop-api`, called by browsers in the EU and probed every few seconds by a Kubernetes liveness check.

What you need

- A deployed service and its `gpilot.toml`.

1. The configuration

gpilot.toml

```
[telemetry]
# Keep the network part of the address: 203.0.113.77 is stored as 203.0.113.0.
client_ip = "truncated"
# Store segment values as keyed pseudonyms (the default, stated so a review can see it).
vary_values = "pseudonymized"
# Leave the cluster's health probes out of Insights.
record = "!starts_with(header:user-agent, 'kube-probe/')"

# Which request headers are recorded. Replacing the list replaces the defaults,
# so the two credential rules come first.
[[telemetry.headers.request]]
name = "authorization"
action = "sanitize"

[[telemetry.headers.request]]
name = "cookie"
action = "sanitize"

[[telemetry.headers.request]]
name = "content-type"
action = "include"

[[telemetry.headers.request]]
name = "accept"
action = "include"
```

Key	Stored
`client_ip = "truncated"`	The address with its last IPv4 octet zeroed, or an IPv6 address cut to 48 bits. Country, region, city and the network data are kept; latitude, longitude and postal code are not. `"none"` stores no address and no region or city.
`vary_values = "pseudonymized"`	Each vary rule's value as `hmac-sha256:` and 16 hex digits, under a key of the service. Requests of one segment still group together.
`record`	Only the requests the expression matches. A probe is still served and counted, just not recorded.
`[[telemetry.headers.request]]`	`authorization` and `cookie` as keyed digests (the `Bearer` scheme and cookie names stay readable), `content-type` and `accept` as sent. Every other request header is not recorded. Compared with the default list, `referer` is left out.

Header rules cannot bring back the address `client_ip` leaves out. Unless `client_ip = "full"`, no rule records `gp-client-ip` or a forwarding header that carries the client's address or URL (`forwarded`, `x-forwarded-*`, `x-real-ip`, `true-client-ip` and the like), not even a catch-all rule. `gp-shield-hop`, which carries the address between edge locations, is never recorded.

The other three header lists (`origin_request`, `response`, `client_response`) keep their defaults, which already sanitize `authorization`, `cookie`, `gp-signature` and `set-cookie`. See `[telemetry.headers]`.

> ⚠ **A list ending in .* records everything**
>
> A rule with `pattern = ".*"` and `action = "include"` records every header, credentials included, unless sanitize rules come before it. The list above names each header instead.

2. Deploy and check a request

Deploy with `gpilot deploy`, then send one request with a credential and one as the probe:

```
curl -sS -o /dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -H "authorization: Bearer $TOKEN" \
  -H 'GP-Correlation-ID: privacy-check' \
  -d '{"query":"query ProductPage { product(id: \"p01\") { id name } }"}'

curl -sS -o /dev/null "https://$GRAPHPILOT_HOST/graphql" -A 'kube-probe/1.31' \
  -H 'content-type: application/json' -d '{"query":"{ __typename }"}'
```

3. What Insights shows

Open the first request in Insights:

- **Client** tab: the IP ends in `.0` (shown in IPv6 notation, such as `::ffff:203.0.113.0`), the country and city are there, and nothing finer.
- **Overview** tab, **Headers from the client**: `authorization` reads `Bearer hmac-sha256:…` with a shield icon, and `user-agent` and `referer` are not listed at all.
- **Cache** tab: each segment value is marked **pseudonymized**.

The probe request is not in the list, and the notice below the tabs names the `record` expression. **Only my requests** matches nothing on this service, because your full address is never stored; filter by the truncated address instead.

Pitfalls

- **It applies from the next deployment.** Requests recorded before keep what was stored then, until your plan's retention deletes them. To remove them sooner, an Admin can reset the history.
- **A probe without a User-Agent is still recorded.** The expression has no answer for a request that sent none, and `record` keeps undecided requests. To leave those out too, write `header:user-agent && !starts_with(header:user-agent, 'kube-probe/')`.
- `vary_values = "clear"` **undoes the pseudonyms for every rule.** It is service-wide, and the header sanitize rules do not cover segment values.

- **Your origin is a separate question.** `origin.forward_client_ip` decides whether your origin receives the full address, independently of what telemetry stores.
- **Alerts and billing are unaffected.** They count every request, recorded or not.

For what is recorded, where it is stored and how long, and what your origin receives, see privacy and personal data.

Related

- Privacy and personal data
- Configuration reference: `[telemetry]`
- What Insights records
- Insights

Troubleshooting

TROUBLESHOOTING

Troubleshooting

Find the symptom you see and follow its link. Each page it points to explains the cause and the fix; this page only sends you to the right one.

Where the evidence is

- **Response headers.** `gp-cache` says what the cache did, and a `PASS` carries `gp-cache-reason` saying why. See headers and cache statuses.

```
# GRAPHPILOT_HOST is your service hostname (<subdomain>.<label>.graphpilot.cloud)
or your custom domain.
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"{ products { id name } }"}' | grep -i '^gp-'
```

- **Insights in the portal**, where each stored request shows its cache status, reason, keys and log lines. See Insights.
- **The CLI.** `gpilot requests` lists recent requests; see the command reference.

Insights and `gpilot requests` show only requests whose telemetry was stored. Nothing is stored in lean mode, or for a request your service leaves out with `telemetry.record`. For any other request, the response headers are your evidence.

Nothing is cached

Every response says `gp-cache: PASS`. Read its `gp-cache-reason` and look it up in why is nothing cached?, which goes from each code to its cause and fix.

The same query never hits

The first response was a `MISS`, and the same request again is another `MISS`. The **Lookup** field of the second request in Insights says whether no entry was found or one could not be used. Check, in order:

- **The first response was a `PASS`**, so nothing was stored: see nothing is cached.
- **The two requests differ.** The query text, operation name and variables are all part of the key: see what the entry is keyed by.

- **The entry is partitioned per caller**, which is correct and looks like a low hit rate: see identifying callers and its vary rules.
- **A purge removed it.** A mutation purges every entry containing what it returned: see verifying a purge.
- **Another edge location answered.** Entries warm up per location: see locations.

The hit rate drops to zero, or paid features are paused

Every service of the account goes to its origin at once, and responses carry no `gp-cache` header. The billing account runs in lean mode: it used up its request allowance on Glider (Free), a payment is overdue, or the account was paused. The edge cache steps aside for every service of the account until lean mode ends, and no request telemetry is stored meanwhile. Purges keep running. Every organization page of the portal shows a **Paid features are paused** notice; see quotas and lean mode and paid features are paused for what ends it.

Stale data after a write

A `HIT` served data your origin has since changed. Find the purge that should have removed it with verifying a purge, and the keys the entry carries with surrogate key debugging. A write that did not go through GraphPilot has to be purged explicitly.

A purge did not clear a response

A response you purged still comes back as a `HIT`. Check, in order:

- **The purge was recorded** and named what you expected: see the purge history.
- **The entry carries the key you purged.** See debugging surrogate keys for the keys a response really has.
- **The response you see came from GraphPilot.** A purge removes the entry at every edge location (purging is global), but not a copy held by a browser or another cache in front of your clients. Confirm at the edge with the `gp-cache` header.

Origin rejects requests after going live

Your origin answers requests through GraphPilot with `401` or `403`. Check the signature verification against the known test vector, your origin's clock against the replay window, and your firewall against the current origin allowlist.

The edge refuses a request

The response carries a GraphPilot code: for a GraphQL request in `extensions.code` of a GraphQL error, for any other request in the `title` of an `application/problem+json` body. The request never reached your origin. Look the code up in the request was rejected. The usual causes:

Status	Usual cause	Where the fix is
`400`	The operation is deeper, longer or makes more selections than the service allows, uses a field name with the reserved prefix, or could not be read unambiguously.	Hardening, limits
`401`	A credential your auth providers could not accept.	Authentication
`403`	Introspection is off, the operation is not in the service's persisted query registry, or the client is not declared or below its required version.	Introspection, persisted queries, client profiles
`404`	A GraphQL operation was sent to a path that is not one of the service's GraphQL paths.	GraphQL on other paths
`405`	A mutation or subscription was sent as a `GET`. The `Allow` header names the method to use.	Codes
`413`	The request body is larger than the service, or the client's profile, allows.	Request size
`415`	A body on a GraphQL path is not a single operation GraphPilot can check, such as a batch or a multipart upload.	`request.forward_unchecked_bodies`

A browser reports a CORS error

The browser hides the real answer behind a CORS error when a response carries no `Access-Control-Allow-Origin` for the calling page. A response from your origin, a cache hit included, carries the CORS headers your origin sent. A response GraphPilot produces itself, such as a refusal from the table above, carries none unless you configure them: see edge rejections. Read the status and code in the browser's network panel, not the console message.

A deployment was refused or shows findings

A deployment refused for a value it names: see a deployment can be refused. A deployment that went live with findings: see diagnostics and findings. A deployment that failed with a support reference: contact support and quote the reference (see a deployment can fail with a support reference).

Subscriptions disconnect

A handshake refused with `401` failed your auth providers; one that fails with `502` could not reach your origin and is never retried. A close after the handshake comes from your origin or your client. See subscriptions and WebSockets.

Requests fail with 502 or 504

A `502` with the code `BAD_GATEWAY` means your origin could not be reached in time, failed TLS, or sent a body GraphPilot could not decode. See connection to the origin: timeouts, TLS and ports, compression and retries.

With shielding, a `504` with the code `GATEWAY_TIMEOUT` means no usable answer came back from the shield in time, and a `502` can also mean a mutation could not be sent through the shield (`SHIELD_UNAVAILABLE_NO_FALLBACK` in your logs). A `503` or `504` without a GraphPilot error code came from your origin or something in front of it.

A custom domain stays pending

A hostname that stays on **Waiting for DNS** or **Verifying**, or ends in **Failed**: see what the statuses mean and the custom domain troubleshooting, which covers an unverified domain, a record that never appears and a hostname already in use.

The CLI cannot log in

- **Check what the CLI holds** with `gpilot auth status`: see check your status.
- **In CI or a script,** `gpilot auth login` cannot ask how to sign in and exits `1`. Pass `--api-key`, or set `GPILOT_TOKEN`: see log in.
- **Exit code** `3` is an authentication failure, and **exit code** `6` a CLI too old to run; see staying up to date. Every exit code is on codes.

Responses are slow

Read `gp-cache` first. A `MISS` or `PASS` includes your origin's time, so look at your origin and its timeouts. A `HIT` depends on the distance between the client and the nearest edge location: see latency and how to measure your own service. If the same request misses at many locations, shielding lets one location's cache stand in front of your origin.

No gp-cache header at all

- **The service runs in lean mode**, where the cache steps aside.
- **The edge refused the request** before it reached the cache, for example with a `401` or a `413`. See the edge refuses a request.
- **The headers are switched off** with `response.cache_headers = false`; see switching the headers on and off.
- **The request did not go through GraphPilot.** Check that the client calls your service hostname or your custom domain, and that its DNS record points to GraphPilot.

Reproducing a miss deliberately

Configure a bypass expression naming a header only you send, such as `bypass = "header:x-debug-bypass"` under `[cache.policy]`. Such a request is answered by your origin and reported as `PASS` with `CACHE_PASS_BYPASS_POLICY`.

Still stuck: status page and support

Check the status page. If the problem is on GraphPilot's side, it is posted at graphpilot.instatus.com.

Contact support from the portal. Open **Help and feedback** (the button in the bottom right corner of the portal, or the same entry in your account menu) and choose **Contact support**. Open it from the service you are asking about: the request carries the organization and service you are looking at, so nobody has to ask for them. Include what you already know: the time, the service hostname, and a `gp-request-id` or correlation id if you have one. The answer comes by email; how soon depends on your plan (see contact support).

TROUBLESHOOTING

Why is nothing cached?

Every response says `gp-cache: PASS`, or the hit rate stays at zero. GraphPilot never refuses to store silently: a `PASS` carries `gp-cache-reason` with the rule that refused, and each rule has one usual cause. This guide goes from the code to the fix. The full order of the decision is on when a response is cached.

The running example is one request you expect to be cached, for example `GET /products/42` or a GraphQL `{ products { id name } }` query, sent to your service hostname.

What you need

- The request, and a way to send it with `curl` as your client does (the same method, path, body and credential).
- Access to your origin's response headers, directly or in Insights.

1. Read the reason

Send the request and keep only the GraphPilot headers:

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/products/42" | grep -i '^gp-cache'
```

```
gp-cache: PASS
gp-cache-reason: CACHE_SKIPPED_SET_COOKIE
```

The same reason is in Insights, on the request's **Cache** tab and in the list's column as **Not cached because**, and the filter of that name finds every request with it. A `MISS` means the response was stored; a second `MISS` for the same request is a different problem, covered under the same query never hits. No `gp-cache` header at all is covered under no gp-cache header at all.

2. Match the reason to its cause

No lifetime was stated

`CACHE_SKIPPED_NO_CACHE_POLICY` or `CACHE_SKIPPED_NO_MAX_AGE`. Neither your origin nor your schema said how long the response stays fresh, so it is not stored.

Fix: send `Cache-Control: s-maxage=60` (or `max-age`) from your origin, or give every type the query selects a `maxAge` with `@cacheControl`. A bare `private` or `public` states no lifetime either.

Your origin sets a cookie

`CACHE_SKIPPED_SET_COOKIE`. The most common cause. A framework that touches the session on every request sends `Set-Cookie` on reads too, and a response that sets a cookie is never stored, because a hit would hand that cookie to the next caller. The request's **Overview** tab in Insights lists the code under **Edge codes** and names the cookie.

Fix: set the session cookie only on the requests that change the session. If the cookie carries no session, such as a load balancer's affinity cookie, add it to `cache.cookies.allow`. See refusals that depend only on the response.

Apollo Server sent `no-store`

`CACHE_SKIPPED_NO_STORE` on a GraphQL response from Apollo Server. Its cache-control plugin sends `Cache-Control: no-store` for every response its hints do not make cacheable, and an origin header wins over your schema.

Fix: add `@cacheControl` hints that cover every field the query selects, or switch the header off and let GraphPilot read the schema; see Apollo Server.

When your hosting platform adds `Cache-Control: private`

Some platforms write `Cache-Control: private` onto every response that carries no `Cache-Control` of its own, together with a session-affinity cookie. **Google Cloud Run** does this when session affinity is on: Google Frontend adds `Set-Cookie: GAESA=…`, `Cache-Control: private` and `Expires` to every successful response without a `Cache-Control`, and leaves error responses and a header your application sent alone. Other platforms with session affinity, such as Google Cloud Load Balancing and Azure App Service, are reported to do the same. Check your origin's headers rather than relying on a list.

The cookie is harmless: `GAESA`, `GCLB`, `AWSALB`, `ARRAffinity` and the other common affinity cookies are on the default `cache.cookies.allow`. The `private` is not. A bare `private` states no lifetime, so it yields to your schema, but it still partitions the entry by `cache.policy.private`. A type you scoped `PUBLIC` or to a named scope is then stored once per caller, and without a `private` expression not at all (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`). GraphPilot cannot tell the platform's `private` from one you sent on purpose, so it honors it.

How to recognize it:

- The request's **Segmentation** in Insights is **Private** and lists your `cache.policy.private` source beside what your schema asked for, on a query whose types all declare a shared scope.
- The **origin response headers** show `cache-control: private` with a `set-cookie` for an affinity cookie.
- **Edge codes** in the **Overview** tab list `CACHE_POLICY_ORIGIN_HEADER_IGNORED` with the header verbatim.
- Your origin, asked directly, sends it on a successful response:

```
curl -sS -D- -o /dev/null https://your-service.a.run.app/healthz | grep -iE 'cache-control|set-cookie'
```

Fix: send a `Cache-Control` of your own that decides nothing. The platform only fills in a missing header. `Cache-Control: no-transform` states no lifetime and no audience, so it yields to your schema and adds no partition. It is also safe for every cache after the edge: without a lifetime, and with `Authorization` on the request, a shared cache may not store the response (RFC 9111 §3.5), and browsers do not cache a `POST` on their own. Set it wherever your responses leave your application, replacing what a framework put there, and keep sending a real policy where you mean one (`no-store` on mutations, `private, max-age=…` on per-person responses):

```
// Hono
app.use("/graphql", async (c, next) => {
    await next();
    c.header("Cache-Control", "no-transform");
});
```

```
// Express
app.use("/graphql", (req, res, next) => {
    res.setHeader("Cache-Control", "no-transform");
    next();
});
```

Or turn session affinity off, if nothing needs it. Without the cookie the platform adds no header. On Cloud Run:

```
gcloud run services update YOUR_SERVICE --no-session-affinity
```

A GraphQL subscription over a WebSocket stays on the instance it connected to for the life of the connection, so it does not need session affinity. State kept in instance memory between separate requests does.

> ⚠ **Two settings that do not fix it**
>
> - **Adding the affinity cookie to `cache.cookies.allow`** only affects the cookie. The `private` directive still applies.
> - `cache.ignore_origin_header_without_freshness = false` makes the header decide instead of yielding. A bare `private` has no `max-age`, so the response is then not cached at all (`CACHE_SKIPPED_NO_MAX_AGE`).

The request carried a credential

`CACHE_SKIPPED_CREDENTIALED_REQUEST`. The request had an `Authorization` header, a session cookie, or the header an auth provider reads, and your origin's `Cache-Control` did not say the response may be shared. With no `private` expression to file it per caller, it is not stored.

Fix: if the response is the same for everyone, send `public` or `s-maxage` (`Cache-Control: s-maxage=60`). If it differs per caller, configure `cache.policy.private`, and mark the response `private`. See authenticated requests.

A scope did not resolve

`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`. The response is partitioned, by a `private` from your origin, a `PRIVATE` field or a named scope, and a value it is keyed on was missing for this request. Its entry under **Edge codes** in the **Overview** tab says whether the request lacked the value (`missing`) or nothing configures the name (`unconfigured`). The same code is reported when the request carried a credential and a `[[cache.policy.vary]]` rule resolved to nothing or to the empty string for it; that entry names the rules under `unresolved`.

Fix: for `private`, configure `cache.policy.private` and check that its source is on the request (a verified token, not an expired one). For a named scope or a vary rule, give every name in its `@vary` a `[[cache.policy.vary]]` entry, and check that the claim it reads is in the token, not empty, and a string, a number or a Boolean. See partitioning your cache.

The query selects a field the deployed schema does not know

`CACHE_SKIPPED_UNRESOLVED_SELECTION`. The query selects a field, or a fragment on a type, that the schema of the live deployment does not declare, at any depth. GraphPilot cannot tell who may share that field's value or for how long, so the response is not stored, whatever your origin's `Cache-Control` says. It is almost always an origin released ahead of its schema.

Fix: deploy the schema your origin runs. Responses are stored again from that deployment on. To avoid the gap, deploy the schema first; see deploy the schema before the origin.

The client asked not to store

`CACHE_SKIPPED_REQUEST_NO_STORE`. The request itself carried `Cache-Control: no-store`. The cache was still read, so this request can be a `HIT`, but its own response is never written. An HTTP client library or a browser's developer tools can add it.

Fix: test with a client that does not send it. If your callers send it and you want their responses stored anyway, set `cache.request_directives.no_store = "ignore"`; see the client's own `Cache-Control`.

The response was an error

A `4xx` or `5xx` status is not stored by default, even with a storable `Cache-Control`, and reports `PASS` with `CACHE_SKIPPED_ERROR_STATUS`. Many frameworks attach one `Cache-Control` to every response, errors included.

Fix: usually none, because this is the protection working. If you do want an error cached, such as a `404` for something that does not exist, list its status in `cache.policy.store_error_statuses`; see an error status is not stored unless you opt in. A GraphQL response with an `errors` array is a different rule, `CACHE_SKIPPED_GRAPHQL_ERRORS`, and is never stored.

A GraphQL response could not be parsed

`CACHE_SKIPPED_UNPARSEABLE_GRAPHQL_RESPONSE` means the body of a GraphQL response was not JSON GraphPilot could read. The client still received it unchanged, but it was not stored, because without reading it GraphPilot cannot see errors in it or derive the keys a purge needs.

Fix: check that your origin answers GraphQL requests with a complete JSON body. If it does and one operation always reports this code, look for very large integers: a JSON integer larger than 18446744073709551615 is not read. Return such values as strings, through an `ID` or a custom scalar. A body over 133 MiB is not read either. See refusals that depend only on what the response carries.

Other refusals

Code	Usual cause	Fix
`CACHE_SKIPPED_UNCACHEABLE_OPERATION`	The operation was a mutation or a subscription. Only a query is ever stored.	None; this is correct.
`CACHE_SKIPPED_NO_CACHE_DIRECTIVE`	Your origin sent `Cache-Control: no-cache`, which forbids reuse without asking your origin first.	Send a `max-age` or `s-maxage` instead, if the response may be reused.
`CACHE_SKIPPED_AUTHENTICATION_HEADER`	The response carried `WWW-Authenticate`, `Clear-Site-Data` or another header about one caller's session.	Send that header only on the responses that need it.
`CACHE_SKIPPED_VARY_WILDCARD`	Your origin sent `Vary: *`.	Name the headers the response varies on; see your origin's `Vary` header.
`CACHE_SKIPPED_SHIELD_ENTRY_EXHAUSTED`, `CACHE_SKIPPED_SHIELD_RESPONSE_NOT_STORABLE`	With shielding, the shield had already used up the response's lifetime, or answered with a response that may not be copied.	None for a stale or used-up copy: a later fresh response from the shield is stored. A response the shield passed has a reason of its own on the shield; fix that one.
`CACHE_SKIPPED_ENTRY_NOT_REPLACED`	An entry this request did not use (older than its client's `max_age`, or unreadable) could not be replaced.	None; the entry is unchanged and still serves other requests.
`CACHE_SKIPPED_SERIALIZATION_FAILED`, `CACHE_SKIPPED_INSERT_FAILED`, `CACHE_PASS_READ_DISABLED`	A fault on GraphPilot's side, not in your configuration.	If it persists, check the status page and contact support.

Any other code

A code starting with `CACHE_PASS_` means the cache was not consulted at all: a bypass matched, a client profile sets `cache = "off"`, the method is uncacheable, the client sent `Cache-Control: no-cache` while `cache.request_directives.no_cache` is `"honor"`, or it was a WebSocket upgrade. Every code, with its fix, is in the reference.

Check that it works

Deploy the fix, or change your origin, then send the request twice:

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/products/42" | grep -i '^gp-cache:'
# MISS
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/products/42" | grep -i '^gp-cache:'   #
HIT
```

A `MISS` carries no reason: it was stored. The `HIT` proves it can be read back. For a per-caller response, send the second request with the same credential.

Pitfalls

- **Configuration applies to responses stored after the deployment.** Entries stored before it keep their old partitioning until they expire or you purge them.
- **A `HIT` for you is not a `HIT` for everyone.** Each edge location fills its own cache, and a partitioned entry is shared only within its audience. If a global audience keeps your hit rate low, switch on shielding, so every location fills from one shared cache.
- **Fixing one refusal can reveal the next.** The rules are checked in order, so after removing `Set-Cookie` the same response may report `CACHE_SKIPPED_CREDENTIALED_REQUEST`. Read the reason again after each fix.

Related

- When a response is cached: every rule, in order
- Cache statuses
- Codes
- Troubleshooting

TROUBLESHOOTING

Verifying a purge

A purge returns as soon as the edge has accepted it, and its result counts keys, not cached responses. To know what a purge actually covered, look it up in the service's purge history. To know that a particular response is

gone, request it again and read the `gp-cache` header.

The purge history

Every purge of a service's cache is recorded, as long as request telemetry is stored: the automatic ones a write's response triggers, and the manual ones sent through the GraphPilot API, `gpilot purge` or the portal. Each record holds the cleartext keys the purge addressed.

The history is request telemetry, so it is stored and kept under the same rules; see what Insights needs.

> ⚠ **Unclear what was invalidated**
>
> A keyed purge runs through its keys in order and stops at the first one that fails. A purge recorded with an error therefore removed an unknown part of what it addressed, and the rest of the cache may still hold what you meant to purge. Send it again: purging a key that is already gone removes nothing.

In the portal

Open the service and choose **Purges**. The list, its filters and the detail panel of one purge are described on Purges. A purge takes effect immediately, but it can take up to a minute to appear there.

Through the GraphPilot API

The same history is the `purges` field on a `Service`, looked up here by its subdomain. It takes a required time window, an optional filter, and cursor pagination (`first`, at most 1000 per page, and `after`). Entries come newest first.

```
query {
  services(condition: { subdomain: "my-shop" }) {
    purges(
      first: 50
      timeRange: { from: "2026-09-22T00:00:00Z", to: "2026-09-23T00:00:00Z" }
      filter: { trigger: [AUTO], key: "User:id:1" }
    ) {
      nodes {
        timestamp
        trigger
        soft
        keys
        keyCount
        error
        actor { name email }
        triggeredByRequest { id path method }
      }
      pageInfo { hasNextPage endCursor }
    }
  }
```

```
}
```

Filter field	Type	Keeps
`trigger`	`[PurgeTrigger!]`	`AUTO`, `MANUAL`, or both.
`scope`	`PurgeScope`	`EVERYTHING` (purge-all) or `KEYS`.
`soft`	`Boolean`	`true` for soft purges, `false` for hard ones.
`outcome`	`PurgeOutcome`	`SUCCEEDED` or `FAILED` (an error was recorded).
`key`	`String`	Purges with at least one key containing this text. `%` and `_` are not wildcards.
`actorUserIds`	`[String!]`	Manual purges sent by these users. Automatic purges have no actor, so this filter excludes them. A purge sent with an API key is not recorded under a member's user id.

All filter fields combine with AND; list fields match any of their values. `keys` is empty and `keyCount` is `0` for a purge-all. `error` is `null` unless the purge recorded one.

What an automatic purge records

When a GraphQL mutation response carries keys that automatic invalidation follows, GraphPilot purges them and records one entry:

- **Trigger** is automatic, and there is no actor.
- **Keys** are the ones it purged: the entity keys and origin keys on the response, plus `![Type]` for each type it touched. Keys that ride along without being purged (such as `m:updateUser` or a type key) are not listed.
- **Triggered by** points to the mutation request.
- It is always a **hard** purge.

A plain HTTP write whose response carries a `Surrogate-Key` header is recorded the same way, with the header's keys and the write request as **Triggered by**.

A mutation whose response carried nothing to purge records nothing. If you expected a purge and the history shows none for that mutation, the response did not name the object; see what is not invalidated automatically.

The mutation request's **Overview** tab in Insights also lists `CACHE_PURGED` under **Edge codes** for each automatic purge, with the number of keys (`key_count`) rather than the keys themselves. Edge codes are stored with the request, so they follow the same rules as the history.

Confirming at the edge

The history tells you what was purged. To see that a particular response is really gone, request it before and after, and compare the `gp-cache` header, which is on every response by default.

1. Send the query and check that it is cached. Run it twice if needed:

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"{ user(id: 1) { id name } }"}' | grep -i '^gp-cache'
```

```
gp-cache: HIT
gp-cache-age: 412
```

2. Purge it, for example with `gpilot purge --yes User:id:1`, or run the mutation that should invalidate it.
3. Send the same query again. After a hard purge the entry is gone and the request goes to your origin:

   ```
   gp-cache: MISS
   ```

   After a soft purge of an entry stored with a stale window you see `STALE` instead: the old response is served once more while GraphPilot fetches a fresh one.
4. Send it once more. It is a `HIT` again, with a small `gp-cache-age`, because the response from step 3 was stored as a new entry.

Purges reach the edge locations asynchronously. If the request in step 3 still reports a `HIT` with a large `gp-cache-age`, retry after a moment before looking further.

If it keeps hitting, the purge did not name a key that entry carries. Check, in this order:

- **The entry's keys.** Switch on the `gp-surrogate-key` header or open the request in the portal, and compare its keys with the ones in the purge record. See debugging.
- **A plain HTTP cache key without its prefix.** A cache key taken from telemetry through the API needs `ck:` in front of it. A GraphQL cache key without the prefix is recognized and purged in both forms, so the prefix is not the cause there; a plain HTTP one sent without it matches nothing. See purging one cached response by its cache key.
- **A hostname in a plain HTTP cache key.** By default the key has no scheme or hostname, so `ck:GET:https://catalog-api.k7m2qp.graphpilot.cloud/products/42` matches nothing: purge `ck:GET:/products/42`. If the service sets `cache.http_key.host = "include"`, it is the other way round: the key is the full URL, and each hostname the path was requested through has an entry of its own to purge.
- **A named operation purged by hash.** `purgeByQueryHash` with a `variablesHash` and `operationName: null` purges only the entry of requests that sent no operation name. Pass the name from the request's cache key, after the query hash and a `#` (`graphql.operationName` can differ for a request that sent none), or omit it to have the names looked up; the lookup refuses the purge when telemetry saw no request with that query and variable set.
- **A different service.** A purge applies to one service; a request to another service's hostname reads another cache.

Related

- Purging through the API and CLI: `gpilot purge` and the four purge mutations
- Automatic invalidation: what a mutation purges
- Cache statuses: what `HIT`, `MISS` and `STALE` mean
- Debugging surrogate keys: reading the keys a response carries

---

TROUBLESHOOTING

Debugging surrogate keys

Invalidation only reaches what the keys name, so “which keys did this response get” is the question behind most invalidation problems. You can read the answer off any response.

The `gp-surrogate-key` response header

With `response.surrogate_key_header = true` (it is off by default), every response that carries keys lists them, space separated. The header grows with the response, one key per object with an entity key, so switch it on where you want to inspect keys, for example in a staging service:

```
# GRAPHPILOT_HOST is your service hostname, <subdomain>.<label>.graphpilot.cloud,
# or a custom domain you attached. The portal lists both.
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/products/42" | grep -i '^gp-surrogate-key'
```

```
gp-surrogate-key: ck:GET:/products/42 product-42 catalog
```

For a GraphQL request, the keys GraphPilot derived from the response are listed too:

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/graphql" \
  -H 'content-type: application/json' \
  -d '{"query":"{ products { id name } }"}' | grep -i '^gp-surrogate-key'
```

```
gp-surrogate-key: ck:9f86d0 ck:9f86d0:4e0740 q:products Product Product:id:1
Product:id:2
```

This is the set the entry was tagged with, which is exactly the set a purge can name to remove it. The header stops at 8,192 bytes by default (`response.surrogate_key_header_max_bytes`). If a `gp-surrogate-key-truncated` header is also present, the list is incomplete and that header says how many keys were left out. The request

detail in the portal has the full set. See size limit. Before raising the limit, check what every client in the path accepts, including proxies and load balancers in front of your clients: a header they refuse fails the whole request, not just the header.

> **Note**
>
> The keys are not secret and the header is not gated. They are what a caller would name to purge this response, and purging is authenticated separately, so showing them costs nothing. If that is not what you want for a public service, leave `response.surrogate_key_header` at its default, `false`. The keys are still written to the entry and purging by key still works; only the header goes. See switching the headers on and off.
>
> Your origin's own `Surrogate-Key` header is what GraphPilot reads keys from; it is never passed on to your client.

Reading the keys

The prefix tells you what kind of key it is, and the kind decides what invalidation can do with it: `ck:` is the cache key, `q:` and `m:` name an operation, `![…]` an empty list, and `Type:field:value` one object. The full table, with what each kind lets a purge do, is on surrogate keys.

What to check

Your origin's keys are there. A key your origin sent in a `Surrogate-Key` header appears in its folded form: a character outside printable ASCII is folded (`Müller` as `Muller`), so look for that spelling. See the header's format.

The deployment accepted your directives. A `@surrogateKey` the deployment build refused contributes no key, and the deployment still succeeds. Read its diagnostics after every schema change.

Your entity keys are there at all. If a response holding products carries only `Product` and `q:products` and no `Product:id:…`, GraphPilot found no key field. Either the type has no inferable key or the field is not annotated. Fix it in the schema, not in the query: you do not need to select the key fields yourself, selection set expansion injects them and strips them again.

They name the type you expect. For an interface or union, the key uses the concrete type that was actually returned. A key reading `Product:id:1` where you expected `Item:id:1` means the invalidation you write against `Item` will miss it.

They are stable across requests. The same object in two different queries must produce the same key, or a purge that clears one will not clear the other. If they differ, the key fields being read differ, which usually means one query reaches the object through a different parent field.

When the keys do not fit

No key is ever silently dropped. If a response's keys do not fit on the cache entry, the response is **not cached at all**, reported as `CACHE_SKIPPED_SURROGATE_KEYS_TOO_LARGE`, so a purge can never miss an entry because a key went

missing from it. A response that generates that many keys is usually a very large list; paginating it fixes both problems at once. See how many keys fit on one response.

In the portal

When a request is stored in Insights (part of every plan, within its logged-request allowance), the same keys are recorded with it. The request detail in the portal shows them on every plan, and from Airliner (Business) you can filter the request list by a key. The list has no surrogate key column. From the CLI, `gpilot requests --wide` shows them (from Airliner (Business)), and `--surrogate-key` finds the requests that carried one.

The service's **Purges** page lists every purge with the keys it addressed. Together, the two answer "what did that purge actually remove" after the fact, without reproducing the request. See verifying a purge.

Related

- Surrogate keys: the kinds of key and what they may contain
- The Surrogate-Key header: keys from your origin
- Headers: every header GraphPilot sets
- Troubleshooting: start from a symptom

TROUBLESHOOTING

Correlating with your logs

A request that went wrong leaves a trace in three places: your client, GraphPilot, and your origin. This guide joins them, so an investigation that starts in any one of them can continue in the other two. GraphPilot gives you two ids to join on:

- **The request id**, which GraphPilot mints for every request it handles and records the request under. You can send it to your origin and to the caller.
- **A correlation id**, which your client chooses and sends. GraphPilot records it with the request and on every log line for it.

The running example is a checkout flow whose client already sends `x-correlation-id` on every call, and whose origin logs `x-request-id` beside each request line.

What you need

- A deployed service and its `gpilot.toml`.
- An origin that logs a request header of your choice, and a client that can send one.

1. Send the request id to your origin

Name the header your origin already logs:

gpilot.toml

```
[origin]
request_id_header = "x-request-id"
```

Every request that reaches your origin now carries GraphPilot's id under that name. If a client sends a header of the same name, GraphPilot's value replaces it. A name in the `gp-` namespace, or one that is not a valid header name, refuses the deployment. Details are on request ids.

2. Show it to the caller

gpilot.toml

```
[response]
request_id_header = true
```

Every response now carries `gp-request-id`. A customer opening a support case can quote it, and your client can log it next to its own error.

3. Read your client's correlation id

By default GraphPilot reads a correlation id from `GP-Correlation-ID`. Your client already sends its own header, so name that one instead:

gpilot.toml

```
[telemetry]
correlation_header = "x-correlation-id"
```

GraphPilot records the value with the request and forwards the header to your origin unchanged. Nothing is generated when the header is missing: that request simply has no correlation id. The default `GP-Correlation-ID` is forwarded too, although it is in the reserved `gp-` namespace. See correlation ids.

Keep the two names apart. If `origin.request_id_header` and `telemetry.correlation_header` were the same, GraphPilot's request id would replace your client's correlation id on its way to your origin.

Deploy with `gpilot deploy`.

4. Find a request from either end

From your origin's log line, take the `x-request-id` value. In Insights, the request list's **Request ID** column (every plan; add it under **Columns**, it is hidden by default) shows it, and the request's **Overview** tab shows it as **Entry ID**.

From your client's correlation id, filter by it:

```
gpilot requests --correlation-id checkout-42 --since 1h
```

In Insights, type `correlation.id:checkout-42` into the filter bar. Filtering by correlation id, in the portal and in the CLI, needs Airliner (Business) or above. On every plan, the request's **Overview** tab shows its **Correlation ID** and **Entry ID**, so you can narrow the list by path, status and time and read both ids there.

From a customer's `gp-request-id`, it is the same id your origin logged, so search your logs for it first.

Check that it works

Send one request with a correlation id and read the id back:

```
curl -sS -D- -o/dev/null "https://$GRAPHPILOT_HOST/products/42" \
  -H 'x-correlation-id: checkout-42' | grep -iE '^gp-(cache|request-id):'
```

```
gp-cache: MISS
gp-request-id: <request id>
```

Your origin's log has a line for that request with that `<request id>` as `x-request-id`, and `x-correlation-id: checkout-42`. Insights shows the request with both. Send the request again: now `gp-cache: HIT`, a new `gp-request-id`, and **no** line in your origin's log.

Pitfalls

- **A cache hit never reaches your origin.** A request id GraphPilot reported that your origin has no record of is normally a hit, not a lost request. Read `gp-cache` to tell them apart.
- **Only stored telemetry can be searched.** Insights and `gpilot requests` read request telemetry, which is stored up to your plan's logged-request allowance, not while the account runs in lean mode, and not for a request the service leaves out with `telemetry.record`. For a request that was not stored, the response headers and your origin's log are the evidence.
- **A correlation id is whatever the client sent.** It is for joining logs, not for trusting. Do not partition or authorize on it.
- **The CLI finds a request by its id.** `gpilot requests --request-id <ID>`, and it prints each row's id as `id` with `-o json`. It also finds requests by correlation id, path, method, status, operation, surrogate key or client IP (each filter needs the same plan as in Insights), and `--wide` adds the id to the table. In Insights, the **Request ID** column shows the id of every listed request.

Related

- Request ids
- Correlation ids
- Insights: the request list and its detail panel

- `gpilot requests`

Portal

PORTAL

The portal

The portal is the web app at cockpit.graphpilot.io. It is where you create organizations and services, invite your team, watch your traffic, read what a deployment built, purge the cache, and manage billing. Everything that changes what the edge serves (your `schema.graphql` and `gpilot.toml`) can also be deployed from here, but most teams deploy from the CLI or from CI and use the portal to look.

Getting an account

Sign up at cockpit.graphpilot.io/signup. After you sign in, the portal opens your organization. If you belong to several, the sign-in asks which one to continue with. If you belong to none yet, it opens the **Organizations** page, where you create one with **New organization**.

The same account signs you in to the CLI: `gpilot auth login --browser` opens the sign-in in your browser. See signing in to the CLI.

How the navigation is laid out

The sidebar changes with where you are. At the top of it, the organization switcher lists your organizations and, under each, its services, so you can jump from one service to another without going back to a list.

Organization

Entry	What it holds	Documented in
Dashboard	Every service in the organization as a card, the request allowance used this period, **New service**, and, until you dismiss it, the organization's onboarding checklist. A service's own dashboard has a checklist of its own.	Services and settings
Members	Who belongs to the organization, their roles, and invitations.	Organizations and members
Billing	Your plan, usage, payment method, invoices and billing account.	Billing and usage
Settings → General	The organization's name, URL slug, website and industry, and deleting the organization.	Organizations and members
Settings → Domains	Domains the organization has verified, for custom hostnames.	Custom domains
Settings → API keys	Organization keys and personal keys.	API keys

Service

Entry	What it holds	Documented in
Dashboard	Requests, cache hit rate, errors and latency for the last 24 hours, open alerts and top operations.	Services and settings
Insights	Your request telemetry: **Requests** and **Operations**.	Insights
Alerts	Every time one of GraphPilot's alert rules fired for this service.	Alerts
Deployments	What is live, what each deployment built and found, comparing two, and **New deployment**.	Deployments
Purges	The **Purge** button, and the history of every purge, automatic and manual.	Purges
Queries	**Persisted queries** (your manifest) and **APQ** (what clients registered).	Queries
Schema	The live schema: **Explorer**, **Cache rules**, **Surrogate keys**, **Operations**, **Findings** and **schema.graphql**.	Schema and config findings
Config	The live configuration: **Values**, **Findings** and **gpilot.toml**.	Schema and config findings
Settings	**General** (name, origin, signing key, origin allowlist, introspection, deleting the service), **Domains**, **Alerts** and **Runtime**.	Services and settings

A service that has never gone live shows a short getting-started panel on its Dashboard, Insights, Alerts, Purges, Queries and Schema pages instead of empty charts. It names the command to run next (`gpilot deploy`) and links to the quickstart.

Account

Open the menu under your name at the bottom of the sidebar. It lists the three account sections.

Section	What it holds	Documented in
Profile	Your personal details, and **Delete account** at the foot of the page.	Your account
Security	Password and multi-factor authentication.	Your account
Sessions	The browsers and devices signed in to your account.	Your account

Hidden or disabled

What you see depends on your permissions and your plan, and the portal shows them differently:

- **A control you lack the permission for is not shown.** Without `deployment:edit` there is no **New deployment** button, and without `cache:purge` there are no purge buttons in Insights or on the Schema pages. Ask an admin of your organization for a role that carries it. Every permission is on permissions.

- **A page you may not read at all says so. Deployments** without `deployment:view`, for example, says that you do not have permission.
- **A control your plan does not include is shown disabled, with the reason.** Point at it to see which plan includes it, for example an Insights filter or column, a latency percentile, or **New service** once the plan's services are in use.

A few controls are exceptions and say so where they are documented: the **Purge** button on the **Purges** page stays visible and disabled without `cache:purge`, because the page would read as broken without it, and **Delete organization** is disabled rather than hidden without `organization:delete`.

When something does not load

A page whose data cannot be loaded says so in place of its content, for example **Could not load billing**, with **Try again**.

When the page loads but a single value on it cannot be read, the rest of the page is shown and the value's place reads **This value could not be loaded.**, with **Try again** where there is room for it. The portal does not show a missing value as an answer: a usage figure that failed is not **Currently unavailable**, a payment method that failed is not **No payment method** (and offers no **Add**), and a requester that failed is not **Not recorded** or **Automatic**. In the request panel of Insights, the panel says **Some details of this request could not be loaded.** above its tabs.

A value that decides what you can do is never shown as missing. When the plan, the account lock, a quota, a signing key, a runtime pin, a deployment's schema or the name typed to confirm a deletion cannot be read, the page shows its error instead. Insights figures, charts and tables also fail as a whole rather than show a partial answer.

A value already on screen stays when a later refresh cannot read it again.

Getting help from the portal

The help button in the bottom corner of every page, and the **Help and feedback** group in the account menu, open the documentation, the FAQ, **Contact support** and **Send feedback**. See contacting support.

Related

- Quickstart: create a service and deploy it
- Services and settings
- Insights
- Deployments and Purges

PORTAL

Organizations and members

An organization is the unit everything else belongs to: its services, its members, its API keys and its verified domains. Permissions are held per organization, so the same person can be an admin in one organization and hold only a few permissions in another.

Create an organization

1. Open the organization switcher at the top of the sidebar and choose **Add organization**, or choose **New organization** on the **Organizations** page.
2. In the **Create a new organization** dialog, enter the **Organization name**. The **URL slug** is filled in from the name as you type; change it if you want a different one. It is what appears in the portal's addresses.
3. Optionally add the **Website URL** and **Industry**.
4. Tick the box to accept the **Terms of Service**, then select **Create organization**. The button stays disabled until the box is ticked.

Creating an organization enters a contract on the free plan, so it requires the terms in force to be accepted. GraphPilot records each acceptance with the version of the terms, the time, the user who accepted (with the email address and name of their login at that moment), the organization and its billing account. The **Privacy Policy** is linked below the box for information; it is not something you accept. When the terms change, the next organization you create asks for the new version.

You become the new organization's admin. It starts with no services: create one from its Dashboard, see Services and settings.

Organizations per billing account

Every organization you create is funded by your billing account, which shares its plan and allowances among them (see billing and usage). A billing account funds at most **3 organizations**, on every plan. When it already funds 3, the dialog shows **Organization not created** with the message "A billing account can fund up to 3 organizations, and yours already does. Delete an organization you no longer need, or contact support to fund more organizations." Nothing is created. Deleting an organization frees its place as soon as the deletion starts. To fund more than 3, contact support: GraphPilot can raise the number for one billing account.

Switch between organizations

The organization switcher at the top of the sidebar lists every organization you belong to, and under each one its services. Pick an organization to open its Dashboard, or a service to open that service directly. The portal switches your session to the organization you picked, so what you see and what you may do always match the organization in the address bar.

The CLI keeps its own current organization. See switch organizations.

Members and roles

Open **Members** in the organization's sidebar. The page lists everyone in the organization with their role, and is where you invite people, change a member's role, and remove a member. An invited person receives an email with a link to accept. Admins, Developers and Members can manage members; Viewers cannot. You can only invite someone with, or change someone to, a role whose permissions you hold yourself: a Member can invite a Member or a Viewer, but not a Developer or an Admin. Only an Admin can give someone the Admin role.

A role is a set of permissions. The permissions are the same ones an API key carries. The GraphPilot API checks them on every action. In the portal, a control you lack the permission for is not shown; see hidden or disabled for the rule and its exceptions.

Every permission, and which ones each job needs, is on permissions.

An organization has four roles: **Admin**, **Member**, **Developer** and **Viewer**. Whoever creates an organization becomes its Admin. Admins receive the billing emails; Admins and Members receive the alert emails. If a member needs a permission they do not have, someone whose role holds every permission of the new role gives it to them, for example an Admin.

Each role holds these permissions:

Permission	Admin	Developer	Member	Viewer
`service:view`	Yes	Yes	Yes	Yes
`service:edit`	Yes	Yes	No	No
`service:delete`	Yes	No	No	No
`deployment:view`	Yes	Yes	Yes	Yes
`deployment:edit`	Yes	Yes	Yes	No
`cache:purge`	Yes	Yes	Yes	No
`organization:edit`	Yes	No	No	No
`organization:delete`	Yes	No	No	No
`billing:view`	Yes	Yes	Yes	Yes
`billing:edit`	Yes	No	No	No

Admins, Developers and Members can also create and revoke organization API keys; Viewers cannot. Every role can create personal keys, limited to the permissions it holds, and only Admins can revoke another member's personal key.

> **Note**
>
> Your own account is not governed by these permissions. Anyone can delete their own account, see Your account.

Organization settings

Open **Settings** in the organization's sidebar. It has three sections.

General

The **Organization name**, **URL slug**, **Website URL** and **Industry**. Select **Save changes** to apply them. Saving needs the `organization:edit` permission; without it the change is refused. Changing the URL slug changes the portal's addresses for the organization, so bookmarks to the old one stop working.

Domains

The domains your organization has verified. A verified domain is what lets a service answer on a hostname of your own. See Custom domains.

API keys

Organization keys and **Personal keys**, for the CLI in CI and for scripts that call the GraphPilot API. See API keys.

Delete an organization

Delete organization sits at the foot of **Settings** → **General**. It needs the `organization:delete` permission; without it the button is disabled.

Deleting an organization deletes every service in it, removes every member's access, revokes every API key, and releases its hostnames and verified domains, which someone else can then claim. The telemetry, logs and stored files of its services stop being readable at once and are deleted within minutes, once each service is off the edge; the organization's remaining telemetry goes with it. Some records, such as the terms the organization accepted and its billing history, are kept on purpose; see what a deletion keeps. Deletion starts immediately and cannot be undone.

If the organization pays for an annual plan, the plan is not refunded: the contract ends with the deletion, fees paid in advance for the rest of its term are not credited, and the usage up to the deletion is invoiced on a final invoice.

The deletion completes in the background. Its services stop accepting requests at once. Until the deletion is complete, the organization stays in the organization list but accepts no changes, and any change is refused with "This organization is being deleted". When the organization has services, this takes at least 15 minutes, and at most 72 hours, without further action.

1. Select **Delete organization**. The dialog lists what will be deleted and what will be released.

2. Type the organization's name to confirm, then select **Send confirmation code**.
3. Enter the 6-digit code from the email the portal sends you. The code is good for ten minutes and for one deletion. Each code allows five tries; after that, send a new one. **Send a new code** replaces the earlier code, so enter the one from the newest email. You can request at most three codes a minute, across every deletion.
4. Review the summary and select **Delete organization**.

Once the deletion has started, a dialog says ***name* is being deleted**. Select **Continue**: the portal moves you to another organization you belong to, or to the **Organizations** page if there is none, while the deletion completes in the background.

> **Portal only**
> The CLI does not delete organizations. Deleting one is done in the portal only.

Related

- API keys
- Custom domains
- Services and settings
- Billing and usage: how a billing account funds organizations

---

PORTAL

API keys

An API key authenticates you **to GraphPilot**, without a browser. It is what the CLI uses in CI, and what a script calling the GraphPilot API presents, for example to purge. You create and manage keys in the portal, under your organization's **Settings** → **API keys**. Who can do what with keys depends on the role:

Role	Organization keys	Own personal keys	Other members' personal keys
Admin	Create and revoke	Create and revoke	See and revoke
Developer, Member	Create and revoke	Create and revoke	No
Viewer	No	Create and revoke	No

A key can only be given permissions its creator holds, so a Viewer's personal key can only read.

> **Note**
>
> These are not the credentials your clients send **through** GraphPilot to your own API. Those never reach GraphPilot as an identity. If you want the edge to recognize them so they can share a header with a JWT, see opaque API keys.

Two kinds of keys

GraphPilot issues keys at two levels. They differ in who owns the key, not in what it can do: either kind acts with the permissions chosen for the key, never with the role of the member who created it.

Organization keys

Recommended for most integrations. The key belongs to the organization and is shared by its members for programmatic access. It is not tied to any single person, so it keeps working as people join or leave the team. That makes it the right choice for services, CI/CD, and server-side integrations.

Personal keys

Linked to an organization, but **owned by your user account**. Use these for your own ad-hoc access. Because the key is tied to your account, revoke it yourself when you leave that organization; it stops working when your account is deleted.

	Organization key	Personal key
Owned by	The organization	You (linked to an organization)
Acts with	The key's own permissions	The key's own permissions, not your role
Shown in the portal as	The key's name and redacted value	The key's name and redacted value
Shared by the team	Yes	No
Survives a member leaving	Yes	No: an Admin revokes it when the member leaves
Best for	Services, CI/CD, integrations	Your own ad-hoc access

Rule of thumb: reach for an organization key by default. A personal key does not make a request act as you: where the portal names who changed something, it names the key, not the person who owns it.

Permissions

A key can do exactly what its permissions allow, and nothing else; you choose them when you create the key. You can choose only permissions your own role holds by name, so a role meant to create a read-only key lists

`service:view` next to `service:edit`. Which permissions a key can carry, and which each job needs, is on permissions.

Some permissions are never available to a key, whatever was selected for it: organization settings, billing changes, the signing key and deleting a service among them. They are dropped from the key when it is used, and the key keeps the rest. A key left with none of the permissions a key may hold is refused with `401`, "API key not accepted: every permission it carries is one an API key may not hold." See when a credential is refused.

A client that presents many different unknown keys in a short time is answered `429`; see API key validation.

Creating a key

1. Open the portal and go to your organization's **Settings** → **API keys**.
2. Choose the section for the kind of key you need (**Organization keys** or **Personal keys**) and create a new key. Name it after what uses it, and select its permissions.
3. **Copy the key immediately.** The full secret is shown only once, right after creation. Store it in a secret manager; you cannot retrieve it again later.

Treat the key like a password: never commit it to source control, and inject it through your platform's secret store rather than hard-coding it. To use it with the CLI, see signing in to the CLI.

Revoking a key

Revoke a key from the same **Settings** → **API keys** screen. Once revoked, the key can no longer be used to authenticate. Note the delay below before assuming it is fully disabled.

How long key changes take to take effect

To keep request latency low, GraphPilot **caches the result of API key authentication for up to 60 seconds**. Because of this cache, revoking a key is *eventually consistent* rather than instant:

- **Revoking a key**: a revoked key may continue to be accepted for **up to 60 seconds** after you revoke it, until its cached entry expires.
- **A newly created key**: works immediately the first time it is used. (Edge case: if that exact key value was sent and rejected *before* the key existed, the rejected result can remain cached for up to 10 seconds.)

> ⚠ **Revocation is not instantaneous**
>
> Allow **up to 60 seconds** for a revoked key to take full effect everywhere. For a compromised key, revoke it **and** rotate any dependent integrations, and account for this window in your incident response rather than assuming the key is cut off the instant you revoke it.

Best practices

- Prefer **organization keys** for services, CI/CD, and shared integrations.
- Give each key only the permissions its job needs, and use separate keys for separate jobs.
- Store keys in a secret manager; never commit them to source control.
- Rotate keys periodically, and immediately if one may have been exposed.
- Remember the **up-to-60-second** window when revoking or rotating.

Related

- Signing in to the CLI: use a key with `gpilot`
- Deploy from CI
- Purging through the API, the most common reason to hold one
- GraphPilot API reference
- Authentication if you meant the keys **your** clients send to **your** API, which are a different thing entirely

---

PORTAL

Custom domains

Every service is reachable on the service hostname GraphPilot issues for it, `<subdomain>.<label>.graphpilot.cloud` (see the hostname label; older services keep `<subdomain>.graphpilot.cloud`). You can add one of your own, so requests go to `api.your-company.com` instead. Both keep working: adding yours does not switch GraphPilot's hostname off. The portal calls that one the **platform hostname**.

Plans Jet (Starter) and above **Permission** `service:edit`

What you need

- **A paid plan.** Glider (Free) includes no custom domains. Paid plans include an allowance of them, counted across every service and every organization your billing account funds, not per service. See pricing, and Billing and usage for how many you use.
- **Permission to edit the service.** Ask an administrator of your organization if the page shows the hostnames but offers no actions.
- **A domain your organization has verified.** This is the part people miss, so it has a section of its own below.

Verify the domain once per organization

Before any service can answer on `api.your-company.com`, your organization has to prove it controls the domain. That happens once, at the organization level, by creating a TXT record.

1. Open **Settings** → **Domains** in your organization, or start from the service's **Settings** → **Domains**: select **Add domain**, and if nothing is verified yet the dialog offers **Verify a domain**, which opens the same dialog, so you do not have to change pages.
2. Enter the domain and create the TXT record it shows you with your DNS provider.
3. Wait for the state to become **verified**. Until then the domain appears in the service's domain picker but cannot be chosen.

Verifying covers everything below it. Verify `your-company.com` once, and every service in the organization can use `your-company.com` itself, `api.your-company.com`, `api.eu.your-company.com`, and anything else under it. You do not verify a hostname per service.

The reverse also holds: verifying only `api.your-company.com` covers that hostname and what sits under it, and nothing else. If you expect to add more hostnames later, verify the registrable domain rather than one subdomain.

> ⚠ **A verified domain is what stops a hijack**
>
> A hostname that points at GraphPilot but belongs to nobody could otherwise be claimed by whoever noticed it first. That is why the check is on ownership rather than on whether the DNS record happens to exist, and why `evil-your-company.com` is not covered by a verified `your-company.com`: the match is on a label boundary, not on how the name ends.

Attach the hostname

1. Open the service's **Settings** → **Domains** and select **Add domain**.
2. In the **Add a custom domain** dialog, pick the verified domain and type the subdomain in front of it. The full hostname is shown as you type.
3. Leave the subdomain empty to use the verified domain itself, including a registrable domain such as `your-company.com`.
4. Select **Add domain**.

If your allowance is used up, the page says so. To use a different hostname, remove one you no longer need first: **Remove** is in the hostname's menu.

Create one DNS record

The hostname appears in the list as **Waiting for DNS**, and shortly after, the records you can create appear underneath it. They take a moment: the certificate has to be requested before the records are known.

You will usually be offered more than one record. **Create one of them**, not all of them. Which one fits depends on where the hostname sits in your zone:

Record	Use it when
`CNAME`	The hostname is a subdomain, such as `api.your-company.com`. Usually the one you want.
`A` / `AAAA`	The hostname is a zone root, such as `your-company.com`. DNS does not permit a `CNAME` there, so this is the only option.

An `A` record comes with several values. Those are anycast addresses for the same hostname, not alternatives to each other: create one record per value.

Check what your provider published:

```
dig +short CNAME api.your-company.com
dig +short A your-company.com
```

What the statuses mean

Status	Meaning
Waiting for DNS	GraphPilot cannot see your record yet. Nothing happens until it appears.
Verifying	GraphPilot can see your record, and the certificate is being issued.
Active	Serving.
Failed	Verification did not succeed; the reason is shown on the hostname, and **Retry verification** starts it again.
Disabled	Switched off, not serving.

Checks run in the background, so the page updates on its own while a hostname is still settling. A record that was only just created can take a while to become visible everywhere, which is normal and needs nothing from you.

Make it the primary hostname

A new hostname never arrives as the primary one, because it has no certificate yet and promoting it would take the service off the air. Once it is **Active**, use **Set as primary** in the hostname's menu.

Switch off the platform hostname

Once another hostname is active and primary, choose **Disable** in the platform hostname's menu and confirm with **Switch it off**. After that the service answers only on your own domains, so make sure everything that calls it has been moved over first.

The platform hostname cannot be removed, only switched off. To bring it back, choose **Enable** in its menu. It is set up again, with nothing to do on your side, and answers once it is **Active**. **Disable** is unavailable while it is the primary hostname or the only active one, and the menu says which.

Troubleshooting

“Your organization has not verified a domain that covers …” The domain, not the hostname, is what gets verified. Check **Settings** → **Domains** in your organization: the entry has to be *verified*, not *pending*, and it has to be the hostname itself or something it sits under.

A domain shows as verified in the portal but cannot be selected. Domains verified before the identity provider required proof of control are listed as needing verification again. Verify the domain once more and it becomes selectable.

”… is already attached to a service” Hostnames are unique across all of GraphPilot. If you used this one on another service, remove it there first.

The record was created but the hostname stays on “Waiting for DNS”. Confirm with `dig` that the record is published, and that it is one of the records the page offered. A `CNAME` on a zone root is refused by DNS itself and will never appear.

A hostname was removed but still answers. Removal starts a deployment that takes the hostname and its certificate off the edge. It keeps answering until that deployment finishes.

Related

- Edge locations: a custom domain is served from the same network and changes nothing about routing, caching or invalidation.
- Request signing, which your origin should verify whichever hostname the request arrived on.

PORTAL

Billing and usage

Billing belongs to the billing account that funds your current organization, not to the organization itself. You reach it through the organization: choose **Billing** in the organization’s sidebar, next to **Members**. The organization Dashboard’s **View billing** button and every **See plans** link in the portal lead there too. The note at the top of the page says how many other organizations share the plan and quota.

Permission `billing:view`

Without `billing:view` the page shows **Could not load billing**, with “You do not have permission to do this. Ask an admin of this organization.” The request meter on the organization Dashboard, the Paid features are paused notice and the lock notice stay visible to every member.

Plans and pricing

GraphPilot has four plans: **Glider** (Free), **Jet** (Starter), **Airliner** (Business) and **Jumbo** (Enterprise). What each includes and costs is on the pricing page.

One billing account, several organizations

A billing account pays for one or more organizations, at most 3 on every plan (see organizations per billing account). Its plan and its allowances apply to all of them together: the services, custom domains and requests of every organization it funds are counted against one allowance. The billing page shows the billing account of the organization you opened it from, so it changes when you switch organizations. The **Account** tab lists the organizations the account funds when there is more than one.

Everything the plan unlocks follows from the billing account too, for example which Insights filters and columns an organization's members see.

The billing page

The top of the page shows your current plan and its state, with what you can do about it. Below it are four tabs.

The billing page shows its buttons to everyone who can open it. Changing the plan, canceling, the payment method, the billing address and paying an invoice need `billing:edit`; without it, the action is refused with "You do not have permission to do this. Ask an admin of this organization."

Plan states

State	The banner says	What you can do
Active	"Renews automatically on" the renewal date.	**Change plan**, or **cancel my subscription**.
Scheduled change	"You are switching to" the new plan on the date it takes effect.	**Cancel plan change**, or **Change plan**.
Past due	"Payment failed. Update your payment method if it changed, then pay the open invoice to keep your plan." The payment is retried automatically, and until it is paid the services run in lean mode; see Overdue payments for what follows if the payment does not arrive.	**Update payment method**, or **Pay open invoice**, which opens the **Invoices** tab.
Incomplete	"Your subscription setup is incomplete." The paid plan applies once its first payment goes through.	**Complete payment**, which reopens the checkout.
Canceled	"Your subscription ends on" the date, after which the account moves to Glider (Free).	**Keep subscription** takes the cancellation back; **Change plan**.
Inactive	"Your subscription is currently inactive."	**Upgrade** or **Change plan**.

On Glider (Free) the banner offers **Change plan**. An account without any subscription, for example after its paid plan was terminated for non-payment, sees **Upgrade** instead.

Tab	Shows
Overview	**Current cycle**: a preview of the upcoming invoice, and **Usage**. When a coupon applies, the preview lists the **Subtotal**, a **Discount** row with the coupon's name and the amount it takes off, and the total after it.
Payment	Your **Payment method** and billing address, each with a button to change it.
Invoices	**Payment history**: previous and upcoming invoices.
Account	**Account details**: your billing account ID, the organizations it funds, and **Copy support details** and **Contact support** for a billing question.

Overdue payments

When a renewal payment fails, the plan stays and the payment is retried automatically. From the first failed attempt until the payment arrives, every service the billing account funds runs in lean mode: requests are still served, without the edge cache and without stored request telemetry, and a Paid features are paused notice says why. The services run in full mode again once the payment arrives.

1. **Reminder.** One payment reminder goes out by email, with the amount due and the last day to pay: the 14th day after the reminder. Days end at midnight UTC. The reminder names the day the account is locked.
2. **Lock.** If the payment has not arrived by the end of that day, the account is locked within the first hour of the next day (UTC), and its services are suspended: the API is no longer served, lean or full, and only billing can be changed and the plan canceled. Every member sees a lock notice, which takes the place of the lean-mode notice. The amount stays due, and nothing is deleted: the services' hostnames and certificates are released at the edge, while every organization, service, deployment, configuration and domain is kept. The lock email names the last day before the paid plan is terminated.
3. **Termination of the paid plan.** If the payment has still not arrived by the end of the 30th day after the lock, the paid plan is terminated for non-payment: the subscription ends and the account moves to Glider (Free). It stays locked until the amount is paid.

Paying what is outstanding ends this at any point, and nothing else is needed: no new plan. The account unlocks on its own and its services are redeployed, usually within minutes: on its plan before the termination, on Glider after it, from where a paid plan can be chosen again on the billing page. Custom domains get a new certificate with that redeployment, which can take a few minutes; if you removed their DNS records while the account was locked, add them again.

While the account is locked, the billing page opens with **Unlock your account**: the amount still open, with a **Pay** button (or **View invoices** when several invoices are open). **Change plan** is disabled. The only changes accepted are paying, the payment method and the billing address; contacting support and deleting your account also still work. Canceling the plan also still works: it takes effect at the end of the current period, so the plan does not renew. It does not pay or cancel the open invoices; they stay due and the account stays locked until they are paid. Taking the cancellation back is not possible while the account is locked.

GraphPilot can also lock an account that misuses the service. The notice then reads **Account suspended**: "This account is locked and its services are not served. Nothing is deleted. Please contact support to have it

reactivated." Only support lifts that lock.

Fair use

Glider (Free) is offered under fair use: an account whose traffic goes far beyond what Glider is meant for is warned, and can then be locked until the start of the next month or a move to a paid plan. Two notices sit directly below the toolbar on every page of every organization the account funds, the billing page included. Neither can be dismissed, and every member sees them, including members whose role cannot open the billing page.

State	The notice says	Action
Warned this month	"Traffic beyond fair use: the services may be suspended. By *date*, this account had sent traffic far beyond what Glider is meant for. Glider is offered under fair use: if this goes on, the account is locked and its services are suspended. Such a lock lifts at the start of the next month, on *date*, or at once on a paid plan. Nothing is deleted." While the account is also in lean mode over its request allowance, the notice says so and takes that notice's place. While it runs lean for another reason (a suspension or a payment), the lean mode notice stays, with its own action, and this one is not shown.	**Upgrade** opens the plan selection.
Locked this month	"Services suspended: traffic beyond fair use. This account sent traffic far beyond what Glider is meant for. Glider is offered under fair use, so the account is locked and its services are not served. Nothing is deleted, and only billing can be changed. The lock lifts at the start of the next month, on *date*, or at once on a paid plan, and the services are then redeployed."	**Upgrade** opens the plan selection.
Locked this month, with a payment overdue	"Services suspended: traffic beyond fair use. This account sent traffic far beyond what Glider is meant for. Glider is offered under fair use, so the account is locked and its services are not served. Nothing is deleted, and only billing can be changed. The fair-use lock lifts at the start of the next month, on *date*. A payment is also overdue: please pay the outstanding invoice by *date*. If it is still unpaid then, the account is locked for the overdue payment, and that lock does not lift at the start of a month. A paid plan can be chosen once the invoice is paid." The second date is the last day to pay that the payment reminder named.	**Go to billing** opens the billing page, where the invoice is paid.

The lift date is the 1st of the next month (UTC). The warning notice goes away at the start of the next month, once the account is locked, or once it is on a paid plan. After the start of the month, until the lock has lifted, the lock notice says the lock is lifting now. A fair-use lock does not end through paying an invoice, and a lock for an overdue payment does not end at the start of a month: see fair use.

Payment method

The **Payment** tab shows the payment method on file, or **No payment method**, with **Add** or **Change**:

- **A card**: its brand, last four digits and **Expires** date.
- **SEPA Direct Debit**: the account holder, the last four digits of the IBAN, the **Bank** and the country.

- **Link**: the **Connected Account** of a Link (by Stripe) wallet, for one-click checkout. A card saved through Link shows as a card with the Link mark.

Invoices

Payment history lists each invoice with its **Status** (for example **Paid** or **Open**), **Date**, **Due date** and **Amount**. Expand a row to see its line items, in the order the invoice PDF lists them. Line amounts are before discounts: an invoice with a coupon, or with tax, also shows its **Subtotal**, a **Discount** row per coupon, **Tax** and the **Total**. **PDF** downloads the invoice. An open invoice has a **Pay** button, which opens the **Pay invoice** page: it charges the saved payment method, or asks for **Confirm payment** when the bank wants a confirmation. To pay with another method, change it on the **Payment** tab first. An invoice that could not be collected has a **Pay** button that opens the payment provider's hosted invoice page.

An invoice is addressed to the name and address on the billing address. Changing the billing address on the **Payment** tab, or from the order step, asks for this name as an organization's name, not a person's. An order with a new payment method saves the billing address entered with it in the same way. An invoice keeps the name it was issued with: a later change applies to the invoices that follow. The same dialog takes an optional **VAT ID**.

Usage

Usage shows these figures against your plan's allowance:

Figure	What happens at the limit
Request units	Counted per billing period, which the row shows, in request units: one per started 10 KiB of uncompressed response body, at every edge location the request passed through (a request through the shield counts at two), so the figure is higher than the request count in Insights when responses are larger than 10 KiB or go through the shield. On Glider (Free), edge caching and stored request telemetry are paused until the plan changes or the period resets, and requests served meanwhile still count; traffic that continues beyond fair use, as set out in the terms, can have the services suspended. On paid plans, further request units are billed as overage, per started million (see the request quota); after a plan change within the period, the row shows the current plan's allowance against the whole period, while the invoice applies each plan's allowance to its own part of it.
Logged requests	Paid plans only. Counted per billing period, which the row shows: the requests kept for Insights, each counted once, against the plan's allowance of logged requests. Further logged requests are billed as overage, per started million. The count is read by the hour, so the hour the period starts in counts whole; after a plan change within the period, the invoice applies each plan's allowance to its own part of the period, while the row shows the current plan's allowance against the whole period. Glider (Free) has no such row: recording stops with lean mode when the request units are used up.
Custom domains	Further ones are refused. Glider (Free) includes none.
Services	Further ones are refused.

When the **Request units** figure is higher than the number of requests clients sent, because request units count a response once per started 10 KiB and a request through a shield at two edge locations, the row names that number below the period, for example **200K client requests. Large responses and requests through the**

shield use more than one request unit. Insights shows the same client request count. When the two are equal the line is not shown.

When a figure cannot be read, its row shows **This value could not be loaded.** with **Try again**, and the rest of the page loads as usual. The **Request units this period** meter on the organization Dashboard, the payment method and the preview of the next invoice do the same. See when something does not load.

Paid features are paused

While the billing account runs in lean mode, a **Paid features are paused** notice sits directly below the toolbar on every page of every organization the account funds, the billing page included. It cannot be dismissed: it goes away when lean mode ends.

Cause	The notice says	Action
The allowance is used up	"The allowance on this plan is used up, so edge caching and stored request telemetry are paused across every service this account funds until the plan changes or the period resets on *date*."	**Upgrade** opens the plan selection on the billing page. Otherwise the period resetting on the date shown restores it. On Glider (Free) this is the request allowance.
GraphPilot paused the account	"Edge caching and stored request telemetry are paused across every service this account funds. Please contact support to have them restored."	**Contact support**. No upgrade is offered, because no plan lifts it.
A renewal payment failed	"A payment failed and is being retried, so edge caching and stored request telemetry are paused across every service this account funds until the payment goes through. Check the payment method on the billing page."	**Go to billing**, where the payment method is updated and the open invoice can be paid. Lean mode ends when the payment arrives; see Overdue payments.
Any other reason	"Edge caching and stored request telemetry are withdrawn across every service this account funds."	None. See lean mode.

Every member sees the notice, including members whose role cannot open the billing page.

While the notice shows, requests are still counted against the allowance but not recorded: they do not appear in Insights or `gpilot requests`, and changing the plan does not bring them back. Recording resumes for requests served after the plan changes.

The pages that go quiet say why instead of showing their usual empty state:

- **Insights**: on the **Requests** and **Operations** tabs, an empty request log reads **No requests are recorded while this account is in lean mode**, with the same action as the notice, and an empty top list reads **Nothing is recorded in lean mode.**
- **Purges**: an empty history reads **Nothing is recorded while this account is in lean mode**. Purges you run still take effect; they are only not listed.

- **Service dashboard**: the service shows **Lean: served without edge caching** beside its name, linking to the notice.

The **Usage** figures keep counting in lean mode. At the limit on Glider (Free) the **Request units** row says so.

What else lean mode changes is on lean mode.

The organization **Dashboard** shows the same figure as **Request units this period**, with the same client request line.

Billing emails

Billing emails come from `billing@graphpilot.io`.

- **To the billing account's email address**: receipts for paid invoices, with the invoice PDF attached, a failed payment (except for a renewal, which the overdue payment emails below cover), refunds, and requests to confirm a payment with your bank. A request to confirm links to the invoice's **Pay invoice** page in the portal.
- **To everyone with the Admin role in each organization the billing account funds**: a plan change, a scheduled plan change and its withdrawal, a cancellation and its withdrawal, the renewal of a yearly plan, and every overdue payment email (the reminder, the lock, the termination and the unlock). Members with any other role receive none of these.

GraphPilot sends these emails itself, receipts included; Stripe sends none of its own.

The billing account's email address is the email address of the person who created the billing account's first organization, as it was when the billing account was set up. Nothing in the portal changes it later: not the billing address, not a new payment method, and not a change to the email address of that person's login. To have receipts and invoices sent to a different address, contact support.

Change your plan

Permission `billing:edit`

1. On the billing page, select **Change plan**, which Glider (Free) shows too. An account without any subscription shows **Upgrade** instead.
2. Pick a plan and a billing interval. Glider, Jet and Airliner are always shown. Each plan is marked **Current**, **Recommended**, **Upgrade** or **Downgrade**. A plan that cannot be ordered here is grayed out with the reason: Glider (Free) is never ordered here. From a paid plan, it is reached by canceling the subscription, and an account without any subscription is put on it automatically. Jumbo (Enterprise) is not offered here.
3. Select **Continue**. The next step lists the plan's usage-billed items with their price per block of requests; they come with the plan and are not chosen. Without a saved payment method, it also asks for a payment method and a billing address, with the organization's name; your invoices are addressed to this name and address, and a billing address already on file is offered there. The summary lists the charges before

discount, the **Subtotal**, a **Discount** row for each coupon that applies, tax, and what is **Due today** after them. With a 100 % coupon, that is €0.00. On a change in the middle of a period, Stripe prices the discount into the prorated charge (“Remaining time on Jet (with 100.0% off)” on the invoice); the summary shows that charge before discount and the discount as its own **Discount** row, named after the coupon. Below the summary, enter a **Promotion code (optional)** if you have one and select **Apply** (see Promotion codes).

Billing period shows when the plan renews and the amount of the renewal before tax. With a coupon in effect, that amount is the renewal as Stripe previews it with the coupons that still apply then, and each coupon says how long it lasts: it applies to every renewal (a coupon for ever), until the date Stripe ends it (a coupon for a number of months), or to this order only (a coupon for one invoice). When the renewal cannot be previewed (a coupon that can no longer be redeemed still applies to the plan, or Stripe does not answer in time), no amount is quoted: the line shows the plan’s price before discounts. Without a coupon, the renewal is at the plan’s price. Below the summary, the checkout states when the plan can be canceled: a monthly plan at any time, effective at the end of the billing period; a yearly plan to the end of its term, after which it renews for another twelve months. The date shown is the end of the period Stripe bills for the plan.

4. With the saved payment method, **Invoice to** shows the name, city and country of the billing address the invoices are addressed to. Select **Edit** to change it in the **Billing Address** dialog. Without a billing address on file, the step asks you to add one first (**Add billing address**), and the order waits for it.

5. Under **Business details**, tick **I am ordering for a company or as a self-employed professional, not as a consumer.** It is required: paid plans are available to businesses (§ 14 BGB) only. With a new payment method, a **VAT ID (optional)** field follows, which asks for the VAT ID only.

6. Tick the box to accept the **Terms of Service**, a separate box, and select **Order now**. The button stays disabled until the business confirmation, the acceptance and the billing address are given.

Every paid order accepts the terms in force again, including a change from one paid plan to another, so an account created before a change of the terms accepts the new version with its next order. GraphPilot records each acceptance with the version of the terms, the time, the user who ordered (with the email address and name of their login at that moment), the organization and the billing account, before the order is sent. The **Privacy Policy** is linked below the box for information; it is not something you accept. A change to Glider (Free), which in the portal is **cancel my subscription**, asks for none of this.

The order sends no name of its own: the invoice of the order and every later one, a reverse-charge invoice included, are addressed to the name on the billing address.

Promotion codes

A promotion code is entered on the order step, below the summary, and checked when you select **Apply**, before anything is ordered. An accepted code is shown as applied and the summary is calculated again: the code shows as a **Discount** row with its coupon’s name, and the order is charged that amount. **Order now** waits until

the summary is up to date. A code that cannot be used for this order is not applied and the summary stays as it was: the reason shows under the field (for example, the code does not exist, has expired, has been redeemed as often as it may be, is for a first order only, needs a higher order amount, or does not apply to the chosen plan). **Remove** takes an applied code off and calculates the summary again without it. The order uses the code the summary shows, also when a new payment method is confirmed first. A code is checked again when the order is placed. Going back to the plan selection starts the order without a code.

A code entered on a plan change replaces a discount the subscription already has. Glider (Free) takes no code.

A discount the subscription already has carries over to a plan change that takes effect at the end of the period, unless that change brings a code of its own. The order summary shows it as well.

A plan change that takes effect at the end of the period shows as scheduled at the top of the page, and **Cancel plan change** withdraws it.

An upgrade, the switch from monthly to yearly billing, and a first paid plan take effect at once and are charged at once. A change to a lower plan or to a shorter interval (yearly to monthly) takes effect at the end of the current period, also when it moves to a higher plan at the same time. The checkout's **Starts** line says which: **Immediately** or **At next renewal**. An upgrade replaces a scheduled plan change or a scheduled cancellation, but only once its payment has gone through: if the payment fails, both stay as they were.

A change to a plan that keeps Insights history for a shorter time deletes the older history. From the day the change takes effect (today for a change that applies at once, otherwise the end of the period), request telemetry older than the new plan's retention is deleted, and a later upgrade does not bring it back. Canceling counts as a change to Glider (Free), which keeps one day. The checkout and the cancel dialog show this as **Insights history is deleted**, with the date. To keep a copy, ask support for an export before that date.

When your bank asks you to confirm the payment

Your bank can ask you to confirm a payment (3D Secure), which is common for cards issued in the EU. When it does for your first paid plan (ordered from Glider (Free), paying by card or Link), the checkout closes and your bank's confirmation opens on the billing page.

- **You confirm it**: the order shows as complete, and the new plan applies once the payment has gone through.
- **You cancel it, or the bank declines it**: **Payment not confirmed** says why, and that your plan has not changed. Nothing is charged. Choose **Change plan** to order again.

Until the payment is confirmed you stay on your current plan, with its features and its scheduled changes. Scheduling another plan change in that time is refused with "A plan change is waiting for your bank to confirm its payment.", until the payment is confirmed or the change is discarded. A plan change that is still waiting for a confirmation after about 23 hours (sooner when the billing period ends first) is discarded, together with its invoice. Until then its invoice is listed as **Open** on the **Invoices** tab, and paying it there completes the change.

A declined card (one that needs no confirmation) fails the order at once, and the plan stays as it is.

Ordering again while a confirmation is outstanding replaces the earlier order, whose invoice is then discarded. The exception is a billing account without any subscription, for example one whose subscription ended after it was locked. There a new order is refused with “An earlier order is still being completed.” while the earlier one waits for its confirmation or its payment is still being processed. Confirm it from its invoice on the **Invoices** tab, or wait until it has gone through or been discarded.

A change from one paid plan to another (an upgrade, or monthly to yearly billing) cannot be confirmed this way: when the bank asks for a confirmation there, the order fails with “The payment did not go through”, and the plan, its scheduled changes and its cancellation stay as they were. The message then points to **Contact support** in the same view: support completes the change with you, and you confirm the payment on the invoice’s payment page. Paying with a card that needs no confirmation, or with SEPA Direct Debit, also completes it.

To end a paid plan, use **cancel my subscription** under the plan. It runs to the end of the current period, and **Keep subscription** takes the cancellation back until then.

Renewal of a yearly plan

A plan billed yearly renews automatically for another twelve months at the end of its term, and can be canceled until the last day of the term. One month before it renews, the members with the Admin role in every organization the billing account funds receive a reminder by email: the renewal date, the plan’s price before tax, and a link to the billing page where the plan can be canceled. The reminder goes out on the day one calendar month before the renewal date (UTC), or on the last day of that month when it is shorter (a renewal on 31 March is announced on the last day of February). Each renewal is announced once.

No reminder goes out for a plan billed monthly, for a plan that is already canceled or has a plan change scheduled for the end of its term (a downgrade, or a switch to monthly billing), or while a payment is overdue: the overdue payment emails apply then. None goes out either once the deletion of the account has started, or while every organization the billing account funds is being deleted. When only some of them are being deleted, the reminder goes to the organizations that stay. A plan whose cancellation is withdrawn, or whose overdue payment arrives, before the renewal still gets its reminder, within a day.

> **Moving a billing account**
>
> The portal cannot move an organization to a different billing account, or hand a billing account to someone else. Contact support for that, see contacting support.

Related

- Quotas and lean mode
- Custom domains
- Your account: deleting an account that pays for organizations

PORTAL

Services and settings

A service is one API behind GraphPilot, plain HTTP or GraphQL: one origin, one configuration (and, for GraphQL, one schema), and one cache. Each service gets its own service hostname, `https://<subdomain>.<label>.graphpilot.cloud`, and its own signing key.

The hostname label

`<label>` is your organization's hostname label, six characters such as `k7m2qp`.

- **Generated, not chosen.** GraphPilot draws it at random when the organization is created.
- **Fixed per organization.** Every service of the organization is under the same label, and it never changes, also when you rename the organization or a service. Renaming a service changes only the first part of its hostname. The old hostname stops answering once the new one is set up, with no grace period, so move every client to the new hostname (or to a custom domain) before renaming.
- **Never reused.** After your organization is deleted, its label is not given to anyone else, so a hostname that once was yours cannot later answer for somebody else.
- **Shown before you create.** The **Create a new service** dialog shows the full hostname, label included, as you type the subdomain. **Settings** → **General** shows it for an existing service.

Existing hostnames keep working. A service created before labels existed keeps `https://<subdomain>.graphpilot.cloud`, and nothing about it changes until GraphPilot moves it onto the label. A move adds `https://<subdomain>.<label>.graphpilot.cloud` beside the old hostname, and once the new one serves, it becomes the service hostname the portal and the CLI show. The old hostname keeps answering beside it for a transition, so running clients are not interrupted, until GraphPilot switches it off. While both exist, renaming the service changes only the labelled hostname.

DNS and custom domains. Nothing changes for custom domains: the record you create for your own hostname is the one the portal shows for that domain, as before. GraphPilot manages the DNS of the service hostname itself; there is nothing to set up for it.

When the hostname starts answering. The service hostname resolves as soon as the service is created. HTTPS needs the organization's certificate, one for all services under the label (`*.<label>.graphpilot.cloud`):

- **The organization's first service** waits for that certificate to be issued, which can take up to an hour. Until then the hostname resolves, but HTTPS requests to it fail with a certificate error.
- **Every later service** uses the certificate that is already there and serves HTTPS right away.

Whether a service is plain HTTP or GraphQL is not a setting. It follows from the live deployment: one deployed without a schema makes the service plain HTTP, with GraphQL features off, and the **Schema** pages say **No schema stored yet**, or **A schema was uploaded but is not live** when one was uploaded (see a service without a schema). The next deployment with a schema turns them on again. Where a deployment takes its schema from when it names none is the service's schema source.

Create a service

Permission `service:edit`

A service's signing key is shown once, when the service is created: in the portal, in the dialog below, or in the output of `gpilot service create`. After that, the only way to get a key you can copy is to rotate it.

1. Open the organization's **Dashboard** and select **New service**, or choose **Add service** under the organization in the organization switcher.
2. Fill in the **Create a new service** dialog:

Field	What to enter
Service name	A display name, for example `Checkout API`.
Origin URL	The URL of your API, GraphQL or plain HTTP, for example `https://api.example.com`. Give only the scheme, host and port: a URL with a path, query or fragment is refused (`ORIGIN_URL_PATH_NOT_ALLOWED`), because GraphPilot forwards the path the client called. Which paths are GraphQL is `request.graphql_paths`. Without a port, GraphPilot connects to port 443 for `https://` and 80 for `http://`.
Subdomain	The first label of the service hostname: `checkout-api` gives `https://checkout-api.k7m2qp.graphpilot.cloud`, under your organization's label. Lowercase letters, digits and hyphens, 2 to 63 characters, not starting or ending with a hyphen. No other service of the organization may use it. Any such name is allowed, `api` and `www` included; only a service whose hostname is directly under `graphpilot.cloud` has reserved names.
Environment	**Development**, **Staging**, **Production** or **Custom** (with a name of your own), and a color. It labels the service in the portal and changes nothing about how it behaves.

3. Keep **Deploy right away (no schema, default configuration)** ticked (the default) to have GraphPilot deploy the service once as a plain pass-through to your origin and then look for a GraphQL API behind it; see set up a new service. Clear it to create the service without deploying anything. If the origin URL is on a private or local address (for example `10.0.0.5`, `192.168.1.10`, `localhost` or a name ending in `.internal`), the dialog says that the edge cannot reach it: the service is still created, but nothing is deployed for it.
4. Select **Create service**.
5. The **Copy your signing key** dialog opens. Copy the key, store it in your secret manager and give it to your origin, then select **I have copied the key**. With **Deploy right away**, the dialog says that GraphPilot deploys the service while you copy the key, and the service's dashboard then shows the setup.

> ⚠ **The signing key is shown once**
>
> The dialog cannot be dismissed by clicking outside it or pressing Escape, and closing it is final. Everywhere else the portal shows only the key's prefix and its last four characters. If you lose it, generate a new one under **Settings** → **General** → **Signing key** and activate it once your origin accepts it. See request signing.

Reserved subdomains

A service under your organization's label may have any valid subdomain, such as `api.k7m2qp.graphpilot.cloud` or `www.k7m2qp.graphpilot.cloud`: the label keeps it apart from every name GraphPilot uses itself.

The names below are reserved only for a service whose hostname is directly under `graphpilot.cloud` (`https://<subdomain>.graphpilot.cloud`, a service created before labels existed): names for infrastructure and tooling (`api`, `www`, `mail`, `cdn`, `docs`, `status`), and the environment and stage names people often reach for first (`dev`, `staging`, `prod`, `test`, `demo`, `beta`, `sandbox`). Renaming such a service to one of them is refused under **Settings** → **General**, on the **Subdomain** field, as reserved. A reserved word inside a longer name is fine.

> › **All 72 reserved subdomains**

The subdomain is also the service's slug: the portal's addresses use it, and so do the `gpilot` commands that take a service.

If the plan's service allowance is used up, **New service** and **Add service** are disabled, say that all of your plan's services are in use, and link to **See plans**. Services are counted across every organization your billing account funds. See Billing and usage.

A service created without **Deploy right away** serves nothing until its first deployment goes live. Until then its dashboard shows its checklist, which starts with the command to run, `gpilot deploy`, and its other pages show a short panel with the same command. If the last deployment was canceled, the panel says so and offers the command again; if one failed, it points to the deployment that says why. See the quickstart.

Set up a new service

A service created with **Deploy right away** is set up in five steps, shown as a **Setting up <service>** card at the top of its **Dashboard**: **Create**, **First deployment**, **Detection**, **Configuration** and **Done**. The state is stored by GraphPilot, not in your browser, so you can leave the page and come back, or open it on another device, and the card shows the step the setup is on. Every member who may see deployments sees the card; the buttons need the permissions named below.

1. **First deployment.** GraphPilot deploys the service once, without a schema and without a configuration file: every path is forwarded to your origin as plain HTTP, every setting is at its default, and only responses your origin marks cacheable with `Cache-Control` are cached. The card shows the build steps while it runs (usually one to two minutes). While the certificate for the service hostname is still being issued, which is the case for an organization's first service, the card then shows **Waiting for the certificate**, usually for a few minutes; the service hostname answers once the certificate is issued. The deployment is listed on the **Deployments** page as requested by **GraphPilot**.

2. **Detection.** GraphPilot sends `{ __typename }` through the service's own hostname to `/graphql`, `/api/graphql`, `/query` and `/`, in that order, and asks the first path that answers as GraphQL for its schema with an introspection query. The requests leave from the edge, like all traffic to your origin, and are signed with the service's signing key. With a runtime that recognizes them, they are not billed and not counted as traffic: see platform probes.

3. **Configuration.** What the card offers depends on what was found:

Found	The card shows
GraphQL on a path, with a schema	**GraphQL found**: the endpoint path, whether introspection answered, the number of types, and the `gpilot.toml` it would deploy (none when the path is `/graphql`, the default). **Apply suggested configuration and schema** starts a deployment with them, as your own deployment. **Keep as plain HTTP** leaves the service as it is.
GraphQL on a path, no schema (introspection is off or needs credentials at your origin)	The findings, and no apply button: read the schema yourself with `gpilot schema pull` and deploy it with `gpilot deploy`.
No GraphQL	**No GraphQL endpoint detected, the service runs as plain HTTP**. If your API answers GraphQL on another path, set `request.graphql_paths` and deploy it with your schema. **Keep as plain HTTP** finishes the setup; **Retry detection** probes again.
No answer from the origin	**The edge could not reach your origin**, with GraphPilot's IP ranges to allow at your firewall and **Retry detection**. The setup waits in this step until detection succeeds or you choose **Keep as plain HTTP**.

   A schema read by introspection carries no applied directives, such as `@cacheControl`, `@surrogateKey` or `@vary`. To use them, deploy your own `schema.graphql`.

4. **Done.** The card states the service hostname, that requests reach your origin, and how to deploy your own schema (`gpilot deploy`, or **Deploy from the portal**). It stays for a day and then disappears.

Apply suggested configuration and schema and **Retry detection** need `deployment:edit`; **Keep as plain HTTP** needs `service:edit`. A deployment of your own that goes live during the setup replaces it, and the card disappears. If a schema or configuration upload lands while **Apply** writes the suggestion, nothing is written and the message offers **Try again**. **Apply** never replaces a `gpilot.toml` or schema you uploaded yourself: if the newest upload of either is yours, it is refused with a message saying to set `request.graphql_paths` in your own `gpilot.toml` and run `gpilot deploy`.

If the setup cannot go on, the card says why and what to do:

- **Not deployed: your origin is on a private address.** The edge cannot reach a private or local address, so nothing was deployed. Deploy once the origin is reachable from the internet, or change it under **Edit origin URL**.

- **The setup stopped.** A deployment or the certificate failed, a step was interrupted before it could finish, the member who created the service may create services but not deployments, or the first deployment was not requested because the billing account is locked or the organization is being deleted. The card links to the failed deployment, to **Edit origin URL** and to **Deploy from the portal**, and offers **Retry detection** and **Keep as plain HTTP** where they apply.

The same setup runs from the CLI with `gpilot service create`.

The organization dashboard

The organization's **Dashboard** is headed **Services**. It shows a card per service with its status (**Live**, **Not deployed**, or **Alert** while an alert is open), its requests and cache hit rate over the last 24 hours, and a link to the service hostname. Above the cards, a meter shows the **Request units this period** against your plan's allowance, with **View billing**.

The onboarding checklist

Until you dismiss it, a checklist above the meter shows the organization's steps to going live, and explains the next open one:

1. **Create a service.**
2. **Get a service to its first cached query**: a deployment of your own (requested by a member or an API key, not the automatic first deployment of the setup) of one of the organization's services completed, a request reached a service hostname, and the edge answered a request from its cache. The step shows which of the three are done (**Own deployment**, **First request**, **First cache hit**) and links to the first service's dashboard, where its checklist walks through them.
3. **Read the go-live checklist**: open the go-live checklist and tick **I have read it**.

The checklist is the same on every device and for every member of the organization. The steps that depend on traffic tick themselves within minutes of the request, and stay done once they have. The last step is the only one you tick yourself.

Under the steps, **Also worth doing** links to **Invite your team** (the **Members** page) and **Create an API key for deploying from CI** (**Settings**, **API keys**). They are not steps and are never ticked.

Dismiss appears once every step is done, for members with `organization:edit`. It hides the checklist for the whole organization, and it does not come back. Ticking **I have read it** needs no permission beyond membership.

The service dashboard

The service checklist

Until its steps are done, the top of a service's **Dashboard** shows the steps that take this service to a first cached query, and explains the next open one:

1. **Deploy your own schema and configuration**: a deployment of this service that a member or an API key requested is live. The automatic first deployment of the setup does not count: while it is the only one, the

step says the service is live as a plain pass-through and shows `gpilot deploy` and **Deploy from the portal**. Otherwise the step shows `gpilot deploy`, or, after a failed deployment, links to the deployment that says why.

2. **Send a first request through the edge**: a request reached the service hostname.
3. **See a first cache hit**: the edge answered a request from its cache. For a service without a schema, only responses your origin marks with `Cache-Control` are cached.

Steps 2 and 3 can be done before step 1, since the automatic first deployment already makes the service reachable. They keep their place in the list.

GraphPilot records the first request and the first cache hit for the organization, not for each service. The checklist therefore shows steps 2 and 3 only while the service is the organization's only one, and only if neither was recorded before the service was created (a service that replaces a deleted one does not take over its traffic). Otherwise it shows the deployment step alone. Steps 2 and 3 tick themselves within minutes of the request.

The checklist has no **Dismiss**: it disappears once its steps are done. If the service's origin URL starts with `http://`, it says so and links to the service's settings, where you can register the origin with `https://` instead. That warning stays after the steps are done. On a plan with custom domains, members who may manage the service's hostnames also get a link to **Serve it on your own domain**.

While nothing of the service is live, the checklist is all the dashboard shows.

Traffic and performance

The service's **Dashboard** covers the last 24 hours:

- **Requests**, **Cache hit rate**, **Origin offload**, **Error rate**, **GraphQL error rate** and **p95 latency**, each compared with the previous 24 hours. Each figure links into Insights for the same 24 hours, filtered to the requests behind it where one applies (cache hits, errors, responses with GraphQL errors, the slowest requests).
- **Error rate** and **GraphQL error rate** count different failures. **Error rate** is the share of requests answered with an HTTP status of 400 or higher. **GraphQL error rate** is the share of GraphQL operations whose response carried at least one GraphQL error, whatever the HTTP status, so it sees the errors that arrive in a `200`, which **Error rate** does not. Both open Insights on the **Errors** column preset.
- **Requests, last 24 hours**, as a chart.
- **Open alerts**, linking to Alerts.
- **Top operations**: the operations called most often. Select one to open it in Insights.

Top operations is available on Jet (Starter) and above; on Glider (Free) the card says so. **p95 latency** also needs Jet (Starter) or above, like the p95 in Insights; on Glider (Free) its card shows a dash and says which plan includes it. The other figures are there on every plan. Like everything built from request telemetry, they count only requests that were stored; see Insights.

Service settings

Open **Settings** in the service's sidebar. It has four sections: **General**, **Domains**, **Alerts** and **Runtime**.

General

Permission `service:edit`

The same four fields as the create dialog: **Service name**, **Origin URL**, **Subdomain** and **Environment**. Select **Save changes** to apply them.

The **Origin URL** is where GraphPilot forwards requests, whether your origin speaks GraphQL or plain HTTP. The change reaches the edge with the next deployment. Only its scheme, host and port are used; see TLS and ports.

Origin allowlist

Under the form, **Origin allowlist** lists the IP ranges GraphPilot's edge calls your origin from, as **IPv4** and **IPv6**, with **Copy all**. Open these at your origin's firewall. The set changes as the network changes, and nothing warns you when it does, so read it on a schedule rather than once. See Origin allowlist.

Introspection

Introspection shows whether the edge answers schema introspection for the live deployment: **Blocked** (the default), **Allowed**, or **Unknown** when the portal cannot tell. It cannot be changed here: set `request.introspection` in your `gpilot.toml` and deploy, and the change takes effect with the next deployment. While it is blocked, queries that select `__schema` or `__type` are refused with `INTROSPECTION_DISABLED`, so GraphiQL, Apollo Studio and codegen pointed at the service hostname stop working against it. The deployed schema is always on the service's **Schema** page.

Schema source

Permission `service:edit`

Schema source decides where a deployment takes its schema from when it names none, such as `gpilot deploy` without a schema file in a scripted run or **New deployment** with the default choice:

Setting	A deployment that names no schema
Latest upload (default)	uses the newest uploaded schema.
No schema	deploys without a schema, as plain HTTP, even if a schema was uploaded.
Introspect at deploy	reads the schema from your origin through the service's own hostname when it builds, and deploys what it read.

Select **Save** to apply it from the next deployment on. A deployment that names a schema, or deploys without one, is not affected.

With **Introspect at deploy**:

- Introspection returns no applied directives, such as `@cacheControl`, `@surrogateKey` or `@vary`. Uploading your `schema.graphql` stays the way to deploy directives.
- Deprecated fields and enum values are read, but deprecated arguments and input fields are not, so a schema read at deploy lacks them. `gpilot schema pull --url` reads them too.
- If introspection fails, the deployment goes live with the newest stored schema (or without one, if none was stored), and its page says **Schema from <date>, introspection failed**. See a deployment's page.
- Your origin has to answer introspection on the GraphQL path for requests from the edge. The requests are platform probes.

Danger zone: reset history

Permission `service:delete`

Reset history deletes the history a service has recorded and leaves the service running: it keeps serving, and its deployments, configuration, hostnames, signing key and alert rules stay as they are. Choose what goes:

Group	Deletes
Request telemetry	Every recorded request of the service, and with them what Insights shows from them: operations, clients, figures, GraphQL errors and error groups, logs, and the suggestions the filters offer.
Purge history	Every entry of the purge history, automatic and manual.
Alert history	Closed incidents in the incident history. An incident that is still open is kept, and so is one that closed less than an hour ago, so its all-clear email is still sent.

Kept in every case:

- The request counts that billing, the plan's quota and the usage figures are based on, and every invoice. A reset never changes what is billed.
- The measurements alerting compares against, so alert rules keep working as before.
- The raw telemetry copy and the backups, for as long as a deletion keeps them.

The reset runs in the background. The dialog says when it was requested, and the history is usually gone within a few minutes. Requests that arrive in the meantime are recorded as usual, so the newest of them can survive the reset. Persisted operations show no usage until they are called again, so do not judge which ones are unused right after a reset. A reset cannot be undone.

1. Select **Reset history** in the danger zone of **Settings → General**.
2. Choose what to delete (all three are selected to start with), type the service's name, and select **Reset history**.

A reset needs `service:delete`, which only Admins hold, and a signed-in member: API keys cannot reset history. Through the API it is the `resetServiceHistory` mutation. It is not in the CLI.

Danger zone: signing key

The **Signing key** block, in the danger zone below **Delete service**, shows the active key (prefix and last four characters) and runs a staged rotation: **Generate a new pending key**, deploy it to your origin so it accepts both keys, then **Activate pending key**. **Cancel rotation** discards a pending key. See rotating the signing key. Seeing and rotating the key needs `service:edit`; without it the block says so.

Danger zone: delete the service

Permission `service:delete`

Deleting a service takes it off the edge at once: requests to it stop being served, its deployments and configuration are removed, and its hostnames are released, so someone else can take them. Its telemetry, logs and stored files stop being readable at once and are deleted within minutes, once the service is off the edge. Some records are kept on purpose; see what a deletion keeps. It cannot be undone.

1. Select **Delete service** in the danger zone of **Settings** → **General**.
2. Read what will be deleted and released, type the service's name, and select **Send confirmation code**.
3. Enter the 6-digit code from the email the portal sends you. It is good for ten minutes and for one deletion. Each code allows five tries; after that, send a new one. **Send a new code** replaces the earlier code, so enter the one from the newest email. You can request at most three codes a minute, across every deletion.
4. Review the summary and select **Delete service**.

> **From the CLI**
>
> `gpilot service delete` deletes a service with the same emailed code, when you are signed in as a user rather than with an API key. To take a service off the edge but keep it, see taking a service down.

Domains

Hostnames lists every hostname the service answers on, including the one GraphPilot issued, and is where you add a hostname of your own. See Custom domains.

Alerts

The alert rules GraphPilot evaluates for the service, each with a switch to turn it off. See Alerts.

Runtime

The GraphPilot proxy runtime the service runs on (**Running**), and the one its next deployment gets (**Selection**): the stable version, or a version you pin here, with its changelog. Changing the pin needs `service:edit` and

`deployment:edit`.

When the choice resolves to the version the service already runs, **Save** stores it without asking and says **Saved. The service already runs `<version>`.** Otherwise **Save** asks first:

- **Deploy now with `<version>`** saves the pin and starts a deployment that rebuilds the service with its current schema and configuration on that version: **Saved. A new deployment with `<version>` has started.**
- **Later (applies to the next deployment)** saves the pin alone: **Saved. The next deployment will use `<version>`.** For **Automatic (stable)** the note names the stable version, because the next deployment takes whatever stable is then.

Closing the question saves nothing. A service that was never deployed has nothing to rebuild: **Saved. The service's first deployment will use `<version>`.** If a deployment is still in progress when the choice matches the running version, GraphPilot starts one anyway, so the newest build carries the pin, and the note says so. See runtime versions.

Related

- Quickstart
- Request signing
- Deployments: the service's deployments in the portal
- Custom domains

PORTAL

Insights

Insights is where you see what reached your service: every request and every GraphQL operation, what the cache did with it, how long it took, and what went wrong. Open a service in the portal and choose **Insights**.

Plans Every plan

What you need

Insights is part of every plan, Glider (Free) included. Plans differ in how many list columns and filters they offer, and in how far back you can look.

Request telemetry is stored on every plan. Paid plans include a monthly allowance of logged requests, and logged requests past it are billed as overage, per started million, from the first day of the billing period. Each logged request counts once against this allowance, including one that went through a shield, and **Usage** on the billing page shows the count (see usage). Glider (Free) has no separate allowance of logged requests: recording stops only when the billing account switches to lean mode, which happens when its request units are used up.

Nothing is stored while the billing account runs in lean mode, and an upgrade does not fill that window in later: see nothing is recorded in lean mode. The allowance for each paid plan is on pricing.

A service can leave some or all of its own traffic out of Insights with `telemetry.record` in its `gpilot.toml`. Those requests are still served and counted. When a service records nothing, every Insights tab says so in place of empty lists; when it records only part of its traffic, a note below the tabs names the expression. While the billing account runs in lean mode, only the lean mode notice is shown, since lean mode is then the reason nothing is recorded.

A request that was not stored does not appear in Insights, in the figures on the service Dashboard or in the purge history. Alerts do not depend on stored telemetry.

How far back you can look depends on your plan:

Plan	Request telemetry is kept for
Glider (Free)	1 day
Jet (Starter)	7 days
Airliner (Business)	30 days
Jumbo (Enterprise)	90 days

Older requests are deleted automatically within a day, at the retention of the plan the organization has at that moment. After a move to a lower plan, what it no longer keeps is deleted within a day, and a move to a higher plan does not bring back what was already deleted. The GraphQL errors recorded with those requests go with them. The error groups and the values the filters suggest are kept as long as they keep occurring within the plan's retention, and are deleted within a day once they have not. If you pick a time range that reaches further back than your plan keeps, the time range picker says so, and that part of the range comes back empty.

An Admin can also delete a service's request telemetry at any time, without waiting for retention: see reset history. The request counts billing and the quota are based on are kept.

The tabs

Tab	Shows
Requests	Every HTTP request to the service, GraphQL or not.
Operations	Every GraphQL operation, named or anonymous.

Requests and **Operations** share one layout, top to bottom:

- **Key figures** for the selected range: total requests (or operations), **Cache hit**, **Latency**, **Error rate** and **Traffic** (bytes in and out). Most figures have shortcuts that filter the list, for example to failed requests or to cache hits. See latency and response size and the two error rates.

- **A chart** over time. **Requests** can be grouped by **Status code**, **Method** or **Cache status**; **Operations** by **Operation**, **Status code** or **Cache status**. See charts.
- **A top list** (Jet (Starter) and above): **Top paths** on Requests, **Top operations** on Operations. Selecting an entry filters by it. On Glider (Free) the list is shown locked, with the plan that includes it.
- **The list** of individual requests or operations, newest first. Select a row to open its detail panel.

Hide analytics folds the figures, chart and top list away so the list gets the room.

The Cache hit figure

Below its rate, **Cache hit** breaks the requests down: **Offload** (requests answered without calling your origin), **Stale** (served from a stale entry; these also count as hits) and **Masked** (your origin answered with an error, 400 or higher, but the client still got a status below 400).

For a service with a shield, a **Shield** section follows whenever requests were routed to it: **Hit** (the share the shield answered from its own cache, fresh or stale), **Offload** (the share of all requests the shield answered, so your origin never saw them), **Fallback** (the share where the shield could not be used: unreachable, too slow, or unavailable for a mutation) and **Shielded** (how many requests went to the shield). Selecting **Hit** or **Fallback** filters the list by **Shield result**.

Latency and response size

The **Latency** figure shows one statistic of one part of the time a request took. The menu in its header picks both:

Part	Measures
Total	Client to response.
Edge	Edge processing (total minus origin).
Origin	Edge to origin and back.

Each part offers the average and the p50, p90, p95 and p99 percentiles, with the value of each shown in the menu. The default is **Total** at p50. The average and p50 are on every plan; p90, p95 and p99, and the **Edge** and **Origin** parts, need Jet (Starter) or above, the same plan as the edge and origin latency columns and filters. An entry your plan does not include is shown disabled, with the plan that includes it.

In the **Traffic** figure, **Response size** shows the size of a single response at a statistic you pick on that line: the average, p50, p90, p95 or p99. All five are on every plan.

The two error rates

The **Error rate** figure is the share of requests that failed: the client got an HTTP status of 400 or higher, or the response carried at least one GraphQL error. A GraphQL error usually arrives in a `200`, and it counts all the same, including an error the edge removed before the client saw it. **4xx** and **5xx** beneath it are the shares of requests with each status class, so they add up to the error rate only when no GraphQL error arrived in a `200`. **GraphQL errors**, on the line below, is the share of GraphQL operations whose response carried at least one GraphQL error,

whatever the HTTP status: it counts over GraphQL operations only, where the error rate counts over every request. It reads 0% when there were no GraphQL operations.

An origin error that the client never saw, because the cache answered or the edge served a stale entry, is not a failed request. The **Masked** figure on **Cache hit** counts those, and the **Masked origin errors** preset lists them.

Select a figure to filter the list: **Error rate** to failed requests (the **Request failed** filter), **4xx** and **5xx** to their status class, **GraphQL errors** to responses that carried a GraphQL error. Its small chart draws the error rate. The alert rule **GraphQL errors in your responses** watches the same failures as **GraphQL errors**.

Charts

Every figure carries a small chart, and the main chart sits below them. Each chart has its own type buttons (**Area**, **Bar**, **Line**), and **Chart type for all charts** at the top of the page sets every chart at once. The legend beside the main chart names each series; selecting a series filters the list by it.

The slider under the main chart narrows the page to part of the range on screen: drag its handles and let go, and the whole page (figures, chart and list) follows. It only narrows; widen the range with the time range picker.

Time range, live and refresh

The controls at the top right apply to every tab:

- **The time range picker** offers relative ranges and a custom range, and the time zone the page shows times in. On **Requests** and **Operations** it also offers **Live**: new entries are added to the top of the list as they arrive, without reloading.
- **Granularity** sets the chart's buckets. The default, **Auto**, picks a bucket size that fits the range. **Per hour** and **Per day** are on every plan, **Per minute** and **Per month** from Airliner (Business). A granularity that would make too many buckets for the range is shown disabled.
- **Auto-refresh** refreshes the data on an interval you pick. Beside it, the page says when its data was last updated.

A relative range such as the last 24 hours ends shortly before the page opened; refresh and auto-refresh move its end to the current moment. A custom range is used exactly as it is.

Filters

Type a filter into the bar above the list, for example `path:/graphql -method:GET`. A leading `-` excludes. As you type, the bar suggests fields and the values seen on this service; **How to write a filter** explains the syntax. The **Filters** panel offers the same filters as form fields, grouped (**Request**, **Response**, **Cache**, **GraphQL**, **Latency**, **Network**, **Deployment**, **Tracing** and more), with **Reset all filters**.

A leading `-` or `!` excludes, on every plan. Several exclusions are combined into one: a request is hidden only when it matches all of them. A field inside an exclusion still needs its own plan, so excluding by a field your plan does not include is refused as that field.

Only my requests narrows the list to requests sent from your browser's IP address, which is the quickest way to find a request you just made. It filters by client IP, so like that filter it needs Jet (Starter) or above. On a service that stores client IPs truncated or not at all (`telemetry.client_ip`), it matches nothing, since your full address is never stored there; a request without a stored address shows **Not stored** as its IP.

Filters are where plans differ. Method, path, hostname, status, operation name and type, latency and sizes are available on every plan, and so is what the cache and your origin did: cache status, cache lookup, **Shield result**, **Not cached because**, **Segmentation** (**Public**, **Private** or **Vary**), origin status, **Origin outcome**, **Masked origin error** and **Request failed**. Other filters open up on Jet (Starter) and above (for example client IP, client ID, device type, country, cache age and TTL, **Segment source** and **Segment**, edge and origin latency) or on Airliner (Business) and above (for example surrogate key, cache key, cache hits, retries, edge location, trace ID). A filter your plan does not include is shown disabled, and says which plan includes it.

Not cached because covers every reason nothing was stored, whether the cache was never consulted (a `CACHE_PASS_*` code) or the response was not stored (a `CACHE_SKIPPED_*` code). Its choices are grouped by who can act on them: **Your configuration**, **The request**, **Your origin's response**, **Your schema or partitioning**, **Shield**, **Follows from another outcome** and **GraphPilot (internal)**. In the filter bar it is `cache.reason` ; `cache.pass.reason` and `cache.skip.reason` are read as the same filter.

Origin outcome says what became of your origin for a request: **Called**, **Call failed**, **Called via shield** (the shield called your origin), **Not called (shield cache)**, **Not called (cache)**, **Shield failed** (the shield gave no usable answer, and whether your origin received the request is not known) or **Not called (refused by GraphPilot)**. On a shielded request it is the reading to trust: the edge records a request it sent on to the shield the same way as a cache hit, even when the shield then called your origin. **Request failed** selects the requests that failed for the client, the same rule as the **Error rate** figure, and **Masked origin error** the ones whose origin failed while the client still got a status below 400.

Segment takes `source=value` pairs, and every pair must hold on the same request. Type the value as it was recorded: a pseudonym (`hmac-sha256:…`) or digest (`v1=…`) is best copied from a request's Cache tab, or reached through **Requests in this bucket** there. A value in clear only matches on a service that stores vary values in clear, and never matches its pseudonym.

Presets

Presets (the button beside the filters) applies a set of filters in one click. Applying one replaces the filters you had; the time range stays. It comes with built-in presets grouped under **Errors**, **Cache**, **Performance**, **GraphQL**, **Origin** and **Client**. Star a preset to keep it under **Favorites**. **Save current filters** stores what you have applied under **Saved**, and a saved preset can be deleted from there.

A preset uses the same filters you could set by hand, so it needs the plan those filters need.

Favorites and saved presets are kept in your browser, not in your account: they do not follow you to another browser or device, and other members do not see them. To hand a set of filters to someone, share the page address, which carries the filters.

> **Every built-in preset**

Columns

Columns, the button at the right of the list's header, chooses which columns the list shows and in which order. Its menu has three parts:

- **Views**: **All presets** holds the built-in views (**Overview**, the default, then **Errors**, **Cache diagnosis**, **Performance**, **Origin**, **GraphQL** and **Clients and geo**). Star one to keep it under **Favorites**. **Save current view** stores your columns, in order, under **Saved**.
- **Columns**, grouped (**Request**, **Operation**, **Response**, **Latency**, **Origin**, **Cache**, **Client**, **Geo**, **Network**, **Edge**, **Other**), each with a count of how many are shown. The timestamp column cannot be hidden. A column your plan does not include is shown locked, with the plan that includes it.
- **Arrange columns**, to change the order.

The column choice is part of the page address, so a link you share opens with the same columns. Favorite and saved views are kept in your browser, like saved presets.

List columns are the other place plans differ. Timestamp, method, path, status, operation, latency, sizes, the cache **Segment** and the **Request ID** (the same id the detail panel shows as **Entry ID**) are available on every plan. So is what the cache and your origin did with each request: cache status, **Not cached because**, **Shield result**, **Origin status** and **Origin outcome**. Columns such as client name and version, client IP, geography, TTL, and edge and origin latency need Jet (Starter) or above; **Cache key**, **Hits**, **Retries**, **Edge location** and connection details need Airliner (Business) or above.

The list shows a code by its words, the same words the filter uses, with the code itself in the tooltip. A few fields are not list columns, because the detail panel shows them on every plan: surrogate keys, the cache **Lookup**, the **Policy source**, the cache entry's max age and stale-while-revalidate window, and the provider and servers under **Diagnostics**.

> **Plans limit the list, not the data**
>
> A column your plan does not include is still in the request's detail panel. The detail panel shows every field of the one request you opened, on every plan.

The request detail panel

Select a row to open the request, or operation, in a panel beside the list. It has four tabs:

Tab	Shows
Overview	The request (method, path, host, size, client, device, the **Entry ID** and the **Correlation ID**, both with a copy button) and the response (status, size); what was sent to your origin (**Request** says whether it was sent at all), with its status, latency and retries; for a request routed through a shield, the **Shield** set: the shield location with its city and country, how the hop ended (**Result**, with what it means) and, when the shield called your origin, that call's status, latency and retries; the edge's location, latency and version; then **Diagnostics**, for a support case: the **Provider**, the **Edge server** within the location that answered and, for a shielded request, the **Shield server**. Then four header sets: **Headers from the client**, **Headers back to the client**, **Headers sent to the origin** and **Headers from the origin**. Which headers are recorded is set in `[telemetry.headers]`. A sanitized value is shown as the digest the edge recorded, `hmac-sha256:` followed by 16 hex digits, marked with a shield icon: it is a pseudonym of the value, not the value itself, and two requests of one service that sent the same value show the same digest. Last, for a request that has any, **Edge codes**: GraphPilot's codes for the request (what it decided, such as `CACHE_STORED`, or why it refused it) with the detail it had, such as the header or cookie that kept a response out of the cache.
GraphQL	The operation name (the `operationName` the request sent or, when it sent none, the name the document gives its only operation) and type, the query and variables hashes, and a persisted query hash if the client sent one. **Document** shows the operation three ways: **Origin**, the document your origin received; **Client**, the document the client sent; and **Diff**, the two against each other, which marks the fields GraphPilot injected. The heading says **GraphPilot touched this document** when fields were injected and **Sent to the origin as written** when none were, and the **Diff** reading then says so instead of a comparison. A request answered from the cache at the edge location says **Served from cache, not sent to the origin**; a request GraphPilot rejected says **Answered by GraphPilot, not sent to the origin**. **Errors** lists the GraphQL errors your origin answered with (at most ten are kept per request; the heading says **Showing 10 of** the total when there were more), each with its code, message and path. An error marked **Stripped before client** was removed by the edge, so the client never saw it. Only for GraphQL requests.
Cache	The cache status; the **Lookup**, which says what the cache lookup found (an entry, no entry, an entry that could not be read or was older than the client's `max_age`, or a failed lookup); **Not cached because** for a request nothing was stored for, with the reason's code under it (the **Decision** section above says whether the cache was consulted first); for a shielded request the shield answered without storing, the shield's own reason, marked **Decided at the shield**, in place of `CACHE_SKIPPED_SHIELD_RESPONSE_NOT_STORABLE` (filtering by the row still matches that code, which is the request's own); **Policy source**, **Hits**, the freshness timeline, the **Segmentation** (**Public**, **Private** or **Vary**, and what it was split by; each value marked **pseudonymized**, **anonymized** or **clear**; a vary rule's value is a keyed pseudonym unless the rule sets `pass_to_origin = "anonymized"` or the service sets `telemetry.vary_values = "clear"`, see what Insights records), the **Cache key**, and every **surrogate key** the response carries.
Client	**Recognized client** (the client profile the request matched, with its version), **Client ID**, IP, the user agent with its **Device** type (for example desktop, mobile, tablet, bot or AI agent), location and connection details.

The freshness timeline draws the entry's life with two markers: **At this request** and **Now**. The figures under it are read at the request, not at the moment you look: **Status** (**Fresh**, **Stale** or **Expired**), **Age**, and **TTL left**, or for a request served stale, **Stale for**, how far into the `stale-while-revalidate` window it was served, out of the whole window. A `MISS` that stored the entry reads **Fresh** with an age of `0s`. Then the instants (**Stored**, **Served**, **Stale from**, **Expires**) and the policy. A `STALE` request also says what happened to the entry: that this request

refreshed it in the background, from your origin or through the shield, or that it started no refresh of its own, which is the case when another request is already refreshing the entry.

Several fields carry an action: filter the list by this value, or purge. The panel header offers **Ask support**, which opens a support request with the entry's ids attached, and **Purge this request**, on every tab. The **Cache** tab offers **Purge this cache key**, and each surrogate key (or a whole level of them) can be purged on its own. The **GraphQL** tab purges the operation, with these variables or with every variable set. Without the `cache:purge` permission the purge controls are not shown. See Purges.

The panel does not show the warnings and errors GraphPilot writes as log lines. Read them through the API's `Service.logs` with the request's **Entry ID** as `requestId` (see the API).

A cache hit links to the request whose miss stored the entry it was served from.

With shielding, request telemetry records how the hop to the shield went (a hit, miss, stale entry, pass, fallback, timeout, or unavailable for a mutation) alongside the entry's own status, so you can tell a hit the requesting location already had from one it fetched from the shield. The hop to the shield is part of the same request: Insights records it once and counts it as one request. Usage is counted in request units instead: by response size, at every edge location a request passed through, with a background refresh through the shield as a request of its own, so the request units for a period are higher than the request count Insights shows. The **Overview** tab shows the hop in its **Shield** set, and the list can be filtered by **Shield result** and show it as a column. When the shield sent the request on to your origin, the **GraphQL** tab shows the document your origin received as the shield recorded it, and says so; the **Diff** reading compares it with the document the client sent. When the shield answered from its own cache, the tab says **Answered from the shield's cache** and names the shield. A request the shield sent on to your origin (shield result `MISS` or `PASS`) carries the hash of the document your origin received, so when GraphPilot changed that document the list marks the request (**GraphPilot touched this document**) and the **Rewritten by GraphPilot** filter finds it, the same as a request that was not shielded. A shield `HIT` or `STALE` sent nothing to your origin and is never marked. Shielded requests recorded before runtime 1.6.0 stay unmarked and do not match the filter. All of this reads the same on a request opened from the **Requests** list and on an operation opened from the **Operations** list.

Related

- Cache statuses: what `HIT`, `MISS`, `PASS` and the reasons mean
- Debugging surrogate keys
- Telling your clients apart: name your clients so Insights can
- Alerts

PORTAL

Alerts

GraphPilot watches each service for a fixed set of conditions. When one holds, it opens an **incident**, emails the organization's admins and members, and emails them again when the condition ends. There is nothing to set up: every rule is on for a new service.

The rules

Rule	What it watches
GraphPilot returned errors	Requests that fail inside GraphPilot, before they reach your origin. Measured over five minutes.
Your origin did not respond	Requests that cannot reach your origin at all. Measured over five minutes.
Your origin returned server errors	Requests your origin answers with a server error. Measured over five minutes.
GraphQL errors in your responses	Successful responses that carry a GraphQL error, which nothing at the HTTP level can see. Measured over fifteen minutes.
Cache hit rate dropped	Cache hit rate against its usual level at the same hour of day over the previous seven days. Measured over thirty minutes, and silent until the service has at least three days of history.
Traffic stopped	Traffic stopping at an hour of day that usually has some, judged against the previous seven days. Measured over fifteen minutes, and silent until the service has at least three days of history.

The first three mean something is broken, the last three that something is degraded. The thresholds are set by GraphPilot, not by you, and a rule needs a minimum number of requests in its window before it can fire, so a service with a handful of requests does not alert because one of them failed.

To see what **GraphQL errors in your responses** is watching, look at the **GraphQL error rate** on the service's Dashboard, or **GraphQL errors** under **Error rate** in Insights. Both count the responses that carried a GraphQL error, whatever the HTTP status, and lead to the list of those responses. The **Error rate** beside them counts HTTP statuses of 400 or higher and does not see these.

Alerts do not depend on stored request telemetry. They are computed from every request, including requests served in lean mode and requests past your plan's logged-request allowance. The one exception is **Cache hit rate dropped**: requests served in lean mode are left out of it, because lean mode does not cache.

The emails

When a condition starts to hold, everyone in the organization whose role is **Admin** or **Member** gets an email headed with the rule's name, as in the table above. A condition that goes on for an hour is one email, not one per check. The condition counts as over once it has not been seen for fifteen minutes, so the second email, saying it stopped, reaches the same people about fifteen to twenty minutes after the last bad minute. If the condition comes back after that, it is a new incident with a new email.

The recipients follow the role, not its permissions. Members with the **Developer** or **Viewer** role, or any other role, get no alert email. With `service:view` they can follow incidents on the service's **Alerts** page. To have a person emailed, give them the Admin or Member role, see members and roles. There is no per-person setting to opt in or out.

Email is the only channel today.

The incident history

Open the service and choose **Alerts**. The page lists every incident: which rule, when it started, how long it went on, and the measurement that crossed the threshold (for example the share of requests that failed). An incident that is still open reads **Ongoing**, and the service's Dashboard shows it under **Open alerts**. The list shows the most recent hundred incidents and says when it is capped.

The page is read-only. What you can change is which rules are evaluated, under the service's settings. An Admin can delete the closed incidents with reset history.

Turn a rule off or on

Permission `service:edit`

1. Open the service's **Settings** → **Alerts**.
2. Each rule is listed with what it watches, when it last fired, and who last changed it. Use the switch beside a rule to turn it off or back on.

   Last fired links to the incident history narrowed to that rule; **Show every rule** there lifts the filter. A rule that has never opened an incident reads **Never fired**.

   Last changed by names the person (point at the name for their email), or an API key by its name and redacted value. It reads **Former member** for someone no longer in your organization, **Deleted user** for a deleted account, and **System** for a threshold GraphPilot tuned for your service.

A switch takes effect within about two minutes. Off means off: a rule that is switched off is not evaluated at all, so no incident opens and nobody is emailed. If you switch a rule off during an incident you were already emailed about, you still get the email when that incident ends.

Members without `service:edit` see every rule and its state, with the switch disabled.

> **Tip**
> Switching **Traffic stopped** off for a planned maintenance window, and back on afterwards, is what the switch is for. Leave the first three on: they are the ones that tell you your API is down.

Example: server errors after a deployment

Ten minutes after a deployment of `shop-api`, the Admins and Members get an email headed **Your origin returned server errors for shop-api**, with the share of failed requests in it. **Open this incident** leads to the service's **Alerts** page, where the incident reads **Ongoing**. From there:

1. In Insights, apply the **Server errors** preset (under **Errors**) for the last 30 minutes. Each row carries the status your origin returned and the operation.
2. Or from a terminal: `gpilot requests --service shop-api --status 502 --status 503 --since 30m --wide`.
3. If the failures start with the deployment, open **Deployments**, choose the deployment before it and **Redeploy**. See roll back to a deployment that worked.

About fifteen to twenty minutes after the last failing minute, a second email says the condition stopped. The whole walk-through is the recipe triaging an alert.

Related

- Insights: find the requests behind an incident
- Troubleshooting
- Cache statuses: why the hit rate moves

PORTAL

Deployments in the portal

Deployments lists a service's deployments in the portal: what is live, everything that came before it, and why a deployment that did not go live did not. Open a service in the portal and choose **Deployments**. What a deployment is, and what happens to traffic and the cache while one runs, is on deployments.

Permission `deployment:view`

Who sees what

Permission	Without it
`deployment:view`	The list, every deployment's page and its two files say "You do not have permission to see this service's deployments." The **Schema** and **Config** pages need it too.
`deployment:edit`	**New deployment**, **Redeploy** and **Stop this build** are not shown.
`service:edit`	**New deployment** and **Redeploy** offer no **Upload a file**, because uploading writes the service's files; every other source stays. A deployment's **gpilot.toml** tab says **The configuration file is not shown to you** instead of the file, with a link to the service's **Config** values, which show every setting with secrets redacted; there is no download and no comparison of the file. A configuration can hold a provider's shared secret, so the file itself needs this permission.

These follow the portal's hidden or disabled rule: a control you lack the permission for is not shown. See permissions.

Running deployments in the header

While a deployment of any of the organization's services is building, and for a while after it ends, the header shows it on every page: **Deployment running**, **Deployment live**, **Deployment failed** or **Deployment canceled**, with a count when there are several. Open it for the list, headed **Deployments**. Each entry links to its deployment's page and disappears once opened; a failed one stays until you open it or select **Dismiss**. A deployment GraphPilot started on its own reads **Automatic**. A deployment that a newer deployment of the same service replaced before it went live ends canceled, so it shows as **Deployment canceled** (see the header for what its page says).

The list

The page opens on the live deployment and, below it, the history, newest first. It loads the 50 most recent deployments and refreshes on its own while one is building.

The Live card

The card at the top names the deployment that is serving traffic: **Live** since when, the **Schema** and **Config** versions it was built from, the **Proxy version** and who deployed it (**Deployed by**). **View schema** and **View config** open its two files. A deployment without a schema shows **No schema: plain HTTP** as its **Schema** and has no **View schema** link, and one without a configuration file shows **Default configuration** as its **Config**. A service with nothing live yet says **Nothing is live yet**.

Deployed by names the person who signed in (point at the name for their email), or, for a deployment made with an API key (from CI, for example), the key's name followed by its redacted value, such as `ci-deploy (sk_…9W8H)`. The value shows when a key was replaced under the same name. A person whose account was deleted reads **Deleted user**, usually followed by a short code that stays the same for all of that person's deployments without identifying them. The first deployment GraphPilot makes when a service is created with **Deploy right away** reads **GraphPilot** (point at it for **Started automatically when the service was created**); see set up a new service. The history rows and a deployment's header use the same wording.

Search and filters

The chips above the list narrow it:

Filter	Keeps
All	Every deployment loaded.
Failed	Deployments that failed. Canceled deployments are not included.
With findings	Deployments whose build recorded at least one error or warning; notices alone do not count, so a deployment without a schema is not listed for its notice.

Filter	Keeps
Schema changed	Deployments whose schema differs from the deployment before them, including a schema added or removed.
Config changed	Deployments whose configuration differs from the deployment before them.

Search deployments (“Search reason, author, version”) matches the deploy message, who deployed it, and the schema and config versions; `no schema` finds the deployments without one, and `default configuration` the ones without a configuration file. The count beside it says how many of the loaded deployments are shown.

One row

Each row shows, left to right:

- **The state**: **Building**, **Live**, **Superseded** (it was live, and a later one took over), **Failed**, or **Canceled** (the build was stopped).
- **The message** given with the deployment, or `Schema <version>, config <version>` without one (`No schema, config <version>` for a deployment without a schema, `default configuration` in place of `config <version>` for one without a configuration file), with who deployed it and when. Where the schema changed against the one that was live before, the line adds `Schema: <summary>`, such as `Schema: +3 types, -1 field`, and `Introspection failed` when the schema was to be read by introspection and could not be.
- **Findings**: the number of errors and of warnings the build recorded. A deployment that never got as far as checking the schema shows `?` for both, which is not the same as zero.
- **The proxy version** it was built on, with an arrow when it moved against the deployment before: up for an upgrade, down (highlighted) for a rollback.
- **Change markers**: `SCH` and `CFG`, highlighted when the schema or the configuration changed against the deployment before, dashed when unchanged, and marked as having nothing to compare with on the oldest deployment loaded. When a schema was removed, `SCH` is highlighted as a warning and reads **Schema removed: GraphQL features off**; when one was added to a service that had none, **Schema added: GraphQL features on**.

Select a row to open the deployment’s page.

A service that has never been deployed shows **No deployment yet** with the command to run, `gpilot deploy`, next to your `schema.graphql` and `gpilot.toml`, and says that **New deployment** can also deploy without either file.

A deployment’s page

The header

The header gives the state, the message, who deployed it, when, and how long the build took. **Rebuild forced** appears when the deployment was started with a forced rebuild. A deployment that failed shows **This deployment did not go live** with the reason: what was refused and why (see a deployment can be refused), or,

for a failure on GraphPilot's side, a support reference. A canceled deployment shows **This deployment was canceled** instead: it was stopped before it went live, nothing it built serves traffic, and whatever was serving before it is unaffected. When GraphPilot stopped it, the alert gives the reason first, for example **Replaced by a newer deployment of this service before it went live.** A deployment stopped on request has none.

Below that, **Schema**, **Config** and **Proxy version** name what it was built from. The schema and config versions link to the files, and the download button beside each saves the file as it was deployed, named after its version (for example `<version>-schema.graphql`). A deployment without a schema shows **No schema: plain HTTP** instead, and one without a configuration file **Default configuration**, with no link and no download. **Proxy version** reads **Resolved during the build** while a deployment is building and has not recorded its version yet.

When another deployment is picked under **Compare against**, each of the three is marked **changed**, **upgraded** or **rolled back** where it differs. **Schema** is marked **removed** or **added** when one of the two deployments has no schema, and **Config** the same way when one of them has no configuration file.

Above them, two lines say where the schema came from when that is news:

- **Schema changes: <summary>**, such as `+3 types, -1 field`, against the schema that was live when the deployment was requested. Open it for each added and removed type and field (`+ type Review`, `- Product.legacyId`). It is not shown when either side has no schema.
- For a service whose schema source is **Introspect at deploy**: **Schema read by introspection on `<path>`** when it worked. When it did not, **Schema from <date>, introspection failed** with the reason: the deployment went live with the newest schema stored before it, uploaded on that date. With no schema stored before, it reads **No schema, introspection failed** and the deployment has none.

Tabs

Tab	Shows
Findings	What the build recorded about your schema and configuration.
Build	Every step of the build pipeline, with a line above it naming the current step (for example **Step 3 of 5: Provision**) or where the deployment ended (**Every step finished.**, **Failed at Build.**, **Canceled during Provision.**). See the build steps.
schema.graphql	The schema this deployment was built from. For a deployment without one: **No schema: this deployment is plain HTTP**.
gpilot.toml	The configuration this deployment was built from. For a deployment without one: **No configuration file: this deployment runs on the defaults**.

The build steps

A deployment runs these steps in order. Open a step for what it does, when it started and when it finished.

Step	What it does
Prepare	Loads the service and checks that it can be deployed.

Step	What it does
Build	Builds the service from the schema and configuration.
Provision	Sets up the new version at the edge, with its hostnames, certificates, origin and logging, without serving traffic yet.
Activate	Switches traffic to the new version.
Finalize	Records the deployment as live.

Each step shows one of these states:

State	Means
Waiting	The deployment is still building and has not reached this step yet.
Running	The step is running now.
Done	The step finished.
Reused	Only for **Build**: an identical build of the same schema, configuration and proxy version already existed, so it was reused.
Skipped	The pipeline passed over the step.
Failed	The step failed. It opens on its own and says **This step stopped here** with the reason.
Canceled	The step was running when the build was stopped.
Not run	The deployment failed or was canceled before it reached this step.

Findings

The findings are grouped by the check that reported them. **Filter findings…** searches the messages, the severity buttons (**All**, **Errors**, **Warnings**, each with its count, and **Notices** when there is one) and the source buttons narrow the list. A notice asks for nothing: a deployment without a schema carries one saying GraphQL features are off. One finding comes from your plan rather than from the build: a configuration that sets `origin.shield` on a plan without shielding gets a warning on `origin.shield`. The shield is deployed anyway, stays inactive, and turns on by itself after an upgrade, with no redeployment. A finding's line links into **schema.graphql**, and the type it sits in opens in the **Explorer** of the live schema. What each finding means is in deployment diagnostics.

Against the compared deployment, a box headed **Since the deployment being compared with** counts the findings that **appeared** and those that were **resolved**, and lists them. It says so instead when there is nothing to compare with, or when one of the two deployments recorded no findings.

Compare against

Compare against, at the top right of the page, picks the deployment everything on the page is measured against. It starts on the deployment just before this one. On the **schema.graphql** and **gpilot.toml** tabs, choose

This version to read the file, or **Side by side** or **Unified** for a diff against the compared deployment. When the two files are identical, the diff buttons are disabled and the page says **Identical to the deployment being compared with.** When only one of the two has a schema, there is no diff, and the **schema.graphql** tab says which one had none.

Start a deployment from the portal

Permission `deployment:edit`

Upload a file also needs `service:edit`, which uploading a schema or configuration needs everywhere (`gpilot deploy` too); without it the option is not offered.

Most teams deploy with `gpilot deploy` or from CI. To start one here, select **New deployment** on the list, or **Redeploy** on a deployment's page, which starts with that deployment's files selected. **Deploy from the portal** and **Deploy it** elsewhere in the portal open the same dialog.

1. For **Schema** and for **Configuration**, pick where the file comes from: **Latest upload** (the last one uploaded, for example by `gpilot deploy`), **From a deployment** (the file an earlier deployment used), **Upload a file** from your machine (with `service:edit`), or none of it: **No schema** or **Default configuration**.

   - **No schema** deploys the service as plain HTTP: every path is forwarded to your origin and GraphQL features are off.
   - **Default configuration** deploys without a `gpilot.toml`: every setting is at its default, and the **Config** page lists each value after the deployment.
   - Both together deploy a pass-through: requests reach your origin unchanged, and only responses your origin marks with `Cache-Control` are cached.
   - A schema with **Default configuration** reads GraphQL on `/graphql` only, and the dialog says so: if your API answers on another path, use a `gpilot.toml` that sets `request.graphql_paths`, or every operation there is refused with `NOT_A_GRAPHQL_PATH`.

   Latest upload is disabled, with **Latest upload: nothing uploaded yet.**, while nothing was ever uploaded for that file. Each half starts on **Latest upload** when there is one, and on **No schema** or **Default configuration** when there is not. Taking a half **From a deployment** that had none deploys without it again; this is also what **Redeploy** does for a deployment without a schema or without a configuration file.

   For a service whose schema source is **Introspect at deploy**, **Schema** also offers **Introspect at deploy**, and starts on it: the build reads the schema from your origin through the service's hostname. Introspection carries no directives such as `@cacheControl`, `@surrogateKey` or `@vary`; upload your `schema.graphql` to use them. With **No schema** as the schema source, **Latest upload** still deploys the newest uploaded schema.

2. Under **Proxy version**, keep the default unless you have a reason to build this deployment on a specific version. The default is the service's runtime pin, else the stable version, and is labelled with what it resolves to now: **Service pin (`<version>`)** or **Stable (`<version>`)**. Below it, every version that can still be deployed is listed, newest first, with the stable one marked **(stable)** and deprecated ones **(deprecated)**. Withdrawn versions are not listed. On **Redeploy**, the version the deployment ran on is marked **same as**

this deployment; if it was withdrawn, or is not in the runtime catalog, it is not offered and the dialog says why. A version picked here applies to this deployment only and does not change the service's pin.

If the service is pinned to a version that was withdrawn or is no longer in the runtime catalog, the default would be refused with `RUNTIME_VERSION_WITHDRAWN`, so the dialog says so and **Start deployment** stays disabled until you pick a version. If the list of versions cannot be loaded, only the default is offered.

3. Tick **Force a rebuild** only if a build itself went wrong. Without it, GraphPilot reuses an existing build for the same two files. If the two files you picked and the proxy version are exactly what is live, the dialog says so: the deployment would succeed and change nothing. If only the proxy version differs from the one the live deployment was built with, the dialog says **Only the runtime changes:** `<live version>` → `<chosen version>` instead. While the default's version cannot be resolved (the list of versions is loading or failed to load), or the live deployment recorded no version, the dialog says neither.

4. Select **Start deployment**. The portal opens the new deployment's page. If the service's schema or configuration changed while the deployment was being prepared, nothing is started and the dialog that says so offers **Try again**, which sends the same deployment again.

Rolling back with **Redeploy** is walked through in staging and rollback.

Stop a build

Permission `deployment:edit`

While a deployment is building, its page offers **Stop this build**. **Stop this build?** confirms: the build ends where it is, nothing it produced goes live, and what is serving traffic now is unaffected. **Keep building** closes the question. Starting again means a new deployment.

> **Note**
>
> Stopping is the portal's `gpilot deployment cancel`. Taking a service off the edge while keeping it is a different thing, see taking a service down.

Related

- Deployments: what a deployment does, and deploying from the CLI
- Schema and config findings: what the live deployment built
- Staging and rollback

PORTAL

Purges in the portal

Purges is where you throw cached responses away by hand and see every purge that has happened, automatic or manual. Open a service in the portal and choose **Purges**. What a purge is, and which key to reach for, is in invalidation; the same purges from a script are in purging through the API and CLI.

Permission `cache:purge`

A service that is not serving yet shows a short panel instead: with nothing built, there is nothing to purge.

Who can purge

Every purge needs the `cache:purge` permission. `service:edit` does not include it. Without it:

- the **Purge** button at the top of the Purges page stays visible but is disabled, and says why when you point at it;
- every other purge control in the portal (in the Insights request detail and beside the keys on the **Schema** pages) is not shown at all.

The page is the exception to the portal's hidden or disabled rule: a page called Purges without its button would look broken. Reading the history needs only `service:view`. See permissions.

Purge by key or hash

1. Select **Purge** at the top of the page. The **Purge cache** dialog opens.
2. Under **Kind**, choose **Hard** ("Dropped now. The next request waits for your origin.") or **Soft** ("Served while it refreshes, so nobody waits."). A soft purge marks entries stale rather than removing them; see soft and hard.
3. Under **What to purge**, enter what to invalidate. The field takes surrogate keys and query hashes, one or many:
    - **Type a key** and press Enter, a comma or a space to add it. Keys cannot contain spaces, so a space always ends one.
    - **Paste a list.** Pasted text is split on spaces, commas and line breaks, and each part is added on its own.
    - **Pick a key pattern.** As you type, the field suggests the keys your live schema declares, with the type each belongs to. A pattern with parts, such as `User:id:{id}`, opens as a line to fill in: type each part, move between them with Tab, and press Enter to add the key. The pattern stays loaded, so the next key of the same shape is one value away.
    - **Reuse a recent key.** The clock button in the field lists keys you purged before from this browser (the last 20 per service). While you type, up to three of them are suggested alongside the patterns.

Each entry becomes a chip marked **key** or **operation**. A value of exactly 64 hexadecimal characters is read as a query hash (an operation); anything else is a surrogate key. A query hash purges the operation with every variable set. To purge a single cached response by its cache key, enter the key with its `ck:` prefix; see purging one cached response by its cache key.

4. Select the button, which names what you entered (for example **Purge 3 keys** or **Purge 1 operation**). Nothing is sent yet.

5. **Purge this cache?** lists every key or hash, with whether it is a hard or a soft purge. Select **Yes, purge immediately** (hard) or **Yes, mark stale** (soft), or **Back** to change it.

The dialog refuses these before you send them:

- keys and query hashes in the same purge (purge them separately),
- more keys than one purge may name,
- a key with a space or a character outside printable ASCII, such as a non-ASCII letter. The portal accepts only printable ASCII keys without spaces; to purge another key, use the API or the CLI.

The limits are on limits.

When it is done, **Purged** says what was invalidated and lists the keys. The count is a count of keys, not of cached responses: a key with nothing cached behind it still counts. The purge takes effect immediately; it can take up to a minute to appear in the history.

Purge all

Purge all, in the other corner of the **Purge cache** dialog, empties the service's whole cache. Until the cache fills again, every request reaches your origin.

1. Select **Purge**, then **Purge all**.

2. **Purge everything?** asks you to type the service's slug (its subdomain) to confirm.

3. Select **Yes, purge everything**.

A purge-all is always hard. It cannot be undone and cannot be aimed at part of the cache.

Purging from other pages

Where the portal already shows what to purge, it offers the purge there. The same dialog opens, on its confirmation step when the target is complete:

- **Insights**, in a request's detail panel: **Purge this request**, the cache key, each surrogate key, and the operation with or without its variables. See the request detail panel.

- **Schema**, wherever a surrogate key is listed (the **Surrogate keys** view, a type in the **Explorer**, an operation). A key with parts, such as `User:id:{id}`, opens the **Purge cache** dialog with its pattern loaded, ready to fill in. See findings.

The purge history

The list shows the purges in the time range picked at the top of the page (the last 24 hours by default), newest first. It is built from request telemetry, so it follows the same storage and retention rules; see what Insights needs. An Admin can delete it with reset history.

Column	Shows
When	The time of the purge.
Trigger	**Automatic** for a purge a write's response caused, **Manual** for one somebody sent.
Scope	The first key and how many more (`User:id:1 and 2 more`), or **Entire cache** for a purge-all.
Outcome	What was done, for example *Invalidated 3 keys* or *Marked 2 keys stale*. A purge that recorded an error reads **Unclear what was invalidated**.
Actor	The member who sent a manual purge. Empty for an automatic purge, and for a purge sent with an API key.

The refresh button beside the time range reloads the list.

Filters

The filter panel beside the list narrows it down:

Filter	Keeps
Trigger	**Automatic** or **Manual** purges.
Scope	**Purge all** (the whole cache) or **Keys** (keyed purges).
Kind	**Soft** or **Hard** purges.
Outcome	Only purges marked **Unclear what was invalidated**.
Key	Purges with at least one key containing this text, matched literally.

Reset all filters clears them.

One purge

Select a row to open the purge. The detail panel lists every key it addressed and, depending on the purge:

- **Triggered by**: for an automatic purge, the request that caused it, linked to its request detail. Once that request is older than your telemetry retention, the panel says so instead.
- **Addressed request**: for a manual purge sent with `purgeByRequest`, the request it named.
- **Requested by**: for a manual purge sent by a signed-in member, that member.

- **Outcome**: when the purge recorded an error, *Unclear what was invalidated* and a note to send the purge again.

What each record means, what an automatic purge records, and how to check at the edge that a response is gone are on verifying a purge.

> ⚠ **Unclear what was invalidated**
>
> A keyed purge stops at the first key that fails, so a purge marked this way removed an unknown part of what it addressed. Send it again; see verifying a purge.

Related

- Invalidation: which key to purge
- Purging through the API and CLI
- Verifying a purge
- Permissions

PORTAL

Queries in the portal

Queries shows what your service is allowed to run: the operations you declared, and the ones your clients registered for themselves. Open a service in the portal and choose **Queries**. It has two tabs.

How persisted queries work at the edge, and how to configure them, is in Persisted queries.

When the live deployment's configuration does not turn persisted queries on, or that is not known yet, both tabs open with a warning: the edge then answers every request that sends a hash with `PERSISTED_QUERY_NOT_SUPPORTED`, whatever the registry holds. `gpilot pq push` prints the same warning.

Persisted queries

Your manifest: the operations you pushed with `gpilot pq push`. You can search it by operation name. The list has these columns:

Column	Shows
Operation	The operation's name, or that it is anonymous. **adopted** marks one a client first registered through APQ.
Type	The operation type, or `unparsed` for a document that could not be read.
Size	The size of the stored document, in bytes.
Requests	How often it was called over the last 30 days, or over your plan's telemetry retention if that is shorter (1 day on Glider (Free), 7 days on Jet (Starter)).

Column	Shows
Last seen	When it was last called.
Cache hits	The share of those calls answered from the cache, as a percentage.
Hash	The start of the hash clients send instead of the document.

The usage figures come from stored request telemetry, so they count only requests that were stored; see what Insights needs. On a plan with a short retention, an operation called a week ago already shows no requests, so check before you revoke it. The same is true after a history reset: every operation shows no requests until it is called again.

If you have not declared anything yet, the list says so and names the command to run.

APQ

Operations your clients registered themselves through Automatic Persisted Queries, and that your manifest does not declare. These are exactly the operations that `trusted` mode would start refusing, so read this list before you switch to it.

To keep one, open it to read its document, then declare it with `gpilot pq push` from where your client code lives. It then joins your manifest. The portal does not add operations to the manifest itself.

The list loads a page at a time; **Load more** fetches the next one. It shows no total.

Reading one operation

Select an operation on either tab to open it. The panel shows whether it is **in your manifest** or **registered by a client**, its usage (**Requests**, **Last seen**, **Cache hits**, **Errors**), and the GraphQL **Document**, which you can copy or open larger. On Jet (Starter) and above, it also links to the operation's requests in Insights.

Revoke an operation

Permission `service:edit`

Revoke in the operation's panel removes it from the registry, on either tab. After that, no client can run it by its hash. The dialog then reports what the revocation did for each operation: revoked; revoked but not yet recorded in the manifest (try again, or a push of the same manifest registers it once more); removed but still served from the cache, or not removed at all (try again in both cases); or not registered in the first place.

> ⚠ **A revocation cannot be undone from the portal**
>
> Any client still sending the hash starts failing. Reinstating an operation means pushing its document again with `gpilot pq push`.

Related

- Persisted queries
- CLI commands: `gpilot pq` and `gpilot apq`
- Insights

PORTAL

Schema and config findings

The **Schema** and **Config** sections of a service show what is live: the schema and the configuration of the deployment that is serving traffic, what GraphPilot made of them, and what it found wrong. They answer “what does my service do right now” without you having to find the right deployment first.

Every deployment’s own page under **Deployments** shows the same findings for that deployment. What each finding means is in deployment diagnostics. In the portal, both sections need the `deployment:view` permission; through the API, the live deployment’s analysis on the service needs only `service:view` (see permissions). `gpilot deploy` prints the deployment’s diagnostics (schema and configuration) when it finishes, unless run with `--detach`. The **Keys that do nothing** list under **Config → Findings** is shown only in the portal and on the GraphPilot API.

Schema

Open a service and choose **Schema**. It has six views.

View	Answers
Explorer	What is in your API: every type and field, with the cache rules and surrogate keys that apply to each.
Cache rules	What will be cached and for how long. Each rule shows **Where** it is declared, how long it is **Cached for** and **How it applies**. A schema with no hints says so, with the default freshness lifetime (TTL) every response gets.
Surrogate keys	Which keys responses are tagged with, where each comes from, and which mutations can throw a cached answer away. Keys can be copied, and purged from here.
Operations	What one call does to the cache: for a query, the keys its cached answer carries and which mutations can throw it away; for a mutation, what it purges and which cached reads that reaches.
Findings	What the deployment build refused or warned about. See below.
schema.graphql	The exact schema that was deployed. Select a line number to copy a link to that line.

The sidebar shows how many items each view holds, and a badge on **Schema** when the live build has findings.

Wherever a surrogate key is listed, it can be copied and, with the `cache:purge` permission, purged; see purging from other pages.

Explorer

The list on the left holds every definition, grouped by kind (each group folds away). Above it:

- **Filter types** (“Filter types and fields…”) narrows the list by name.
- **Sort types**: **In the document**, **A to Z**, **Annotated first** or **Problems first**.
- Three filters, one per mark a row can carry: **Cache rules**, **Surrogate keys** and **Findings**. Each keeps the types that have it, or those that do not; an active one shows as a chip such as “no cache rules”. Selecting a mark on a row shows only the types with that mark.

Select a type to read it. Its detail shows the definition, as **Type** (the structured view) or as its source in **schema.graphql**, with a copy button and **Previous type** and **Next type**. Below it:

- **Cache rules**: how long a response holding this type is cached, and where that comes from. Without a rule, it says the schema’s default applies.
- **Surrogate keys**: which keys this type puts on a response, from your annotations or from GraphPilot’s defaults, including the keys it inherits from its interfaces and unions, each under the interface or union and under the type itself (`Medium:isbn:{isbn}` and `Book:isbn:{isbn}`). Without one, a cached response holding it can only be thrown away by waiting out its TTL or by purging everything, and the panel says so. Each key carries a mark: a bolt for a key a mutation can purge (dynamic), a hand for a key nothing purges on its own (fixed). A dynamic key that no mutation in the schema can return also gets the hand.
- **Can be purged by**: shown when the type has a dynamic key. It names each mutation that can reach this type and how: it returns the type (“returns it”), the client selects the path to it (“when the client selects `products`”), or GraphPilot adds that path to every call. A mutation purges only the keys its own response carries, so each call purges the objects it actually returned. When no mutation returns the type, directly or through anything it returns, the panel says so: purge its keys by hand, through the purge API, or by naming them in a `Surrogate-Key` header on a mutation’s response.
- **What this mutation purges**: shown on a type a mutation returns (a mutation payload type), in the same blocks as on a mutation in Operations.
- **Referenced by**: the types and fields that point to this one.

On the query and mutation root types, **Keys at runtime** points to **Open Operations** instead, where each root field is answered as its own operation.

Operations

The list on the left holds every operation, grouped into **Queries**, **Mutations** and **Subscriptions**; **Filter operations** narrows it by name. A query or mutation that reaches nothing on the other side carries a red mark: a mutation whose keys no cached read is tagged with, or a query no mutation can throw away. Before you select one, the page counts how many operations are marked.

Select an operation to see what one call does to the cache. Its header names the type it **Answers with**, which opens that type in the **Explorer**. Below it, a summary strip:

- for a query, **Carries** (how many keys its cached answer carries), **Cleared by** (how many mutations can throw it away) and **Kept for** (the cache rule on the field or its return type, or the schema default);
- for a mutation, **Purges** (how many keys it purges) and **Reaches** (how many cached reads that can reach).

A count of zero under **Cleared by** or **Reaches** shows in red.

What the cached answer carries (for a query) or **What this mutation purges** (for a mutation) lists those keys in blocks:

- **Always**: carried by every call, whatever the client selected.
- **From the selection**: carried only where the client selected the type that declares them, one selection step from what the operation returns.
- **Injected** (mutations only): keys the proxy adds whatever the client asked for, so the entity the mutation changed stays purgeable even from a response that only acknowledges the write. A note gives the `surrogate_key.mutation_payload_depth` the descent ran at, and how many more paths one level deeper would reach.

Then, for a query, **What can throw this away** lists the mutations that produce a key its cached answer carries; for a mutation, **What this reaches** lists the cached reads tagged with a key it produces. Each row shows how many keys the two share, expands to list them, and links to that operation. A shared key still only throws away the entries whose values also match, so the list says which operations can be reached, not what every call reaches. With more than six rows, **Filter by operation or key** narrows the list.

When the list is empty, the panel says what that means. For a query: “No mutation in this schema produces a key this answer is tagged with. Once it is cached it stands until its age runs out, whatever happens to the data behind it.” For a mutation: “Nothing this API caches is tagged with anything this mutation produces.” A client reading the same data after the mutation then gets the answer from before the write until that entry expires.

A subscription is not cached, so its page says **Nothing to cache, nothing to purge**.

Findings

Schema → Findings lists the diagnostics the build of the live deployment recorded. Each finding has a severity (error, warning or notice), the part of the build that reported it, a message, and the line and column in your schema. Filter by severity and by source, or search the messages. A finding’s line links into **schema.graphql**, and the type it sits in links into the **Explorer**.

An error means the build refused something, for example a `@surrogateKey` whose selection path does not exist. A warning means it built, but probably not the way you intended. Fix the schema and deploy again. What each diagnostic means is in deployment diagnostics.

On **Findings**, a service that has never gone live says **Nothing has been built yet**.

A service without a schema

When the live deployment has no schema, the service is plain HTTP and GraphQL features are off. **Explorer**, **Cache rules**, **Surrogate keys**, **Operations** and **schema.graphql** then show one of two states instead of an

empty page, decided by whether a schema was ever uploaded:

- **No schema stored yet**: the service is serving plain HTTP, so there are no types, cache rules, surrogate keys or operations to show. It offers **New deployment** (with `deployment:edit`) and the command `gpilot deploy` to run next to your `schema.graphql`, and, to read the schema from your running API, `gpilot schema pull`.
- **A schema was uploaded but is not live**: a schema was uploaded on the date shown, but the service is serving plain HTTP until a deployment with it goes live. **Deploy it** opens the **New deployment** dialog.

The header still names the deployment that is live and links to it. A service with nothing live at all, such as one created without **Deploy right away**, shows the same two states, saying that nothing is deployed yet; after a failed or canceled deployment it shows the panel that points to that deployment instead. Deploy with a schema to fill the views in.

Findings still lists what the build recorded, because a deployment without a schema is still checked against its configuration: the notice that it has no schema, and configuration findings such as the error for a `request.graphql paths` that is set but ignored. A notice is a third severity beside error and warning. It asks for nothing, the **Notices** button appears only when there is one, and it is never counted in the badge on **Schema**.

Config

Open a service and choose **Config**. It has three views.

A deployment without a configuration file (see **Default configuration** in starting a deployment) runs on the defaults. The header then reads **Default configuration** in place of the generation, and a note above the views says **This service runs on the default configuration**: every value is a default, and a `gpilot.toml` of your own changes them. With nothing live, the section shows the defaults the service's last deployment without a configuration file recorded, under **Nothing is deployed yet**; a service that never had one says **Nothing is deployed yet** and that a configuration appears once a deployment has gone live.

Values

Effective configuration: every setting the live deployment runs with, as a tree, whether you set it or it is a default. **Only what I set** hides the defaults, and **Filter by key** narrows the tree. Each value can be copied; secrets, such as a provider's shared secret, are shown redacted here. The stored document itself is not redacted: `gpilot config get` returns it as it was deployed, secrets included, to members and API keys with `service:edit`. Each section links to its entry in the configuration reference.

Beside `origin.shield`, the tree names the shield's city and country. On a plan that includes shielding it adds **A request sent to the shield counts at two points of presence, so it uses twice the request units; a background refresh through it is a request of its own.** (see request units); on Glider (Free) it says the shield is inactive on the plan and turns on by itself after an upgrade.

Beside the three origin timeouts, the tree adds **your plan allows up to** the plan's ceiling (see what your plan allows). The ceiling is your plan's today, not the one the deployment was built under. A key that is accepted but that nothing reads is marked **read by nothing**; see the findings below.

Findings

Keys that do nothing: keys in your `gpilot.toml` that change nothing. There are two kinds:

- **Unknown keys.** Nothing reads them. Unknown keys are ignored almost everywhere in a service configuration, so a misspelled section deploys cleanly and changes nothing. This is where that becomes visible. Check the spelling against the reference.
- **Accepted keys without an effect.** The key is spelled correctly and accepted for compatibility, but has no effect. There is nothing for you to fix.

Each finding names the line in the file, linked to the **gpilot.toml** view.

> **Tip**
> Look here when a setting seems to have no effect. A misspelled key never fails a deployment, so this list is where it shows up.

gpilot.toml

The configuration file exactly as it was deployed, for members with `service:edit` (a configuration can hold a provider's shared secret); without it the view says so and links to **Values**, where every setting is shown with secrets redacted. Select a line number to copy a link to that line. For a deployment without a configuration file it says so: there is no `gpilot.toml` to show, and every setting is at its default.

Related

- Deployments: each deployment's own findings
- Deployments
- Configuration reference
- The caching decision
- Surrogate keys

PORTAL

Your account

Your account is you, the person who signs in. It is separate from the organizations you belong to: you can be a member of several, and leaving one does not end your account. Open the menu under your name at the bottom of the sidebar to reach your account settings.

Profile

Profile shows your personal details and lets you update them. **Delete account** sits at the foot of the page, see below.

Security

Security is where you change your password and set up multi-factor authentication.

Sessions

Sessions lists the browsers and devices signed in to your account. Revoke any you do not recognize; that device is signed out.

Signing in to the CLI with `gpilot auth login` uses the same account. See signing in to the CLI.

Delete your account

Anyone can delete their own account; no organization permission is needed.

> **This deletes organizations, not only your login**
>
> Deleting your account deletes **every organization your account pays for**, with all of its services, whoever else is a member. Making a colleague an admin first does not save such an organization: it follows the billing account that funds it, not its admins. If an organization or a service should carry on, contact support **before** you delete your account, so its funding can be moved. The portal cannot move it.

What deleting your account does:

- **Your login ends.** Once the deletion has started, the portal shows **Your account is being deleted**; select **Sign out**. The login itself is deleted when the deletion completes.
- **Your subscription is canceled**, if you pay for one, and the usage up to the deletion is invoiced on a final invoice.
- **A running annual plan is not refunded.** The contract ends with the deletion; fees paid in advance for the rest of its term are not credited.
- **Organizations your account pays for are deleted**, with every service in them. Their services are taken off the edge, with their deployments and configuration.
- **Organizations somebody else pays for stay.** You only leave them, and what you did there, for example who made a deployment, is anonymized. Your email address is also removed from their alert history, which records who each alert email was sent to. If you are the only admin of one, it is left without an admin, so hand the role over first if that matters.
- **Your personal API keys stop working**, because the account they belong to is gone.
- **Your invoice history is kept**, because GraphPilot is required to keep it.
- **The record of the terms you accepted is kept** as evidence of the contract, with the email address and name it was recorded with.
- **Telemetry, logs and stored files** of the deleted services stop being readable at once and are deleted within minutes, once each service is off the edge. Some records are kept on purpose; see what a deletion keeps.

Deletion starts immediately and cannot be undone. It runs in the background and is complete within 72 hours at the latest, with no further action from you. While it runs, the organizations your account pays for accept no changes: any change by one of their members is refused with “This account is being deleted”. A confirmation email arrives once the deletion is complete.

1. Open **Profile** and select **Delete account**.
2. The first step, **What gets deleted**, lists by name each organization that will be deleted (with its number of services, if it has any) and each organization you will only leave. It lists only what applies to your account: if your account pays for no organization, there is no list of deleted organizations. Read it, type your email address to confirm, and select **Send confirmation code**.
3. Enter the 6-digit code from the email the portal sends you. The code is good for ten minutes and for one deletion. Each code allows five tries; after that, send a new one. **Send a new code** replaces the earlier code, so enter the one from the newest email. You can request at most three codes a minute, across every deletion.
4. Review the summary and select **Delete my account**.

Contacting support

The help button in the bottom right corner of every page, and **Help and feedback** in the account menu, offer **Documentation**, **FAQ**, **Contact support** and **Send feedback**.

Contact support opens a form:

Field	What to enter
What is this about?	**Billing**, **Technical problem**, **Account or access** or **Something else**. It decides who reads your request first.
Which parts does this involve?	Any of **General**, **Proxy** (caching, invalidation, deployments), **Portal**, **Command line** and **API**.
Subject	A short summary.
Your question	What you expected to happen, what happened instead, and when you first saw it.

Select **Send request**. The portal adds the context of the page you sent it from (the organization, service, plan, billing account, page and portal version), so you do not have to look those up. The organization and the billing account are always those of the organization you are signed in to, and the service is attached only if your session can see it.

A person reads every request, and the answer comes by email to your account’s address. You also get a receipt by email with a request reference; replying to it reaches support too. There is no ticket page to watch: the answer arrives in your inbox. Quote the reference if you write about the same request again.

How soon the first reply comes depends on your plan, and the dialog states the figure for the plan you are on:

Plan	Channel	First reply
Glider (Free)	Documentation; email without a commitment to scope or handling, much more limited than on the paid plans	No commitment
Jet (Starter)	Email	Best effort, usually within 2 working days
Airliner (Business)	Email, handled with priority	Best effort, usually within 1 working day
Jumbo (Enterprise)	As agreed	As agreed

Support hours are Monday to Friday, 9:00 to 17:00, excluding public holidays in North Rhine-Westphalia. A response time is the time to the first substantive reply, not to the fix.

If the form cannot send your request, it offers to open it as an email to info@graphpilot.io with everything you typed.

Send feedback is a shorter form for praise, ideas and things that got in your way. It goes to the team, and it is not a support request.

For a billing question, the **Account** tab of the billing page has **Copy support details**, which copies the identifiers support needs.

Related

- Organizations and members
- Billing and usage
- API keys: personal keys end with your account

CLI

CLI

CLI overview

`gpilot` is the command-line interface for GraphPilot. You use it to authenticate, manage organizations and services, deploy your schema and configuration, list and cancel deployments, purge cached responses, read request telemetry, and manage persisted queries. Everything it does fits in a script or a CI job.

A few tasks live only in the portal: deleting an organization, for example. Deleting a service works in both: `gpilot service delete` asks for the confirmation code sent to you by email, as the portal does.

To install it, see installing the CLI.

Command groups

Group	Aliases	What it does
`gpilot auth`	None	Log in and out, and inspect the current session
`gpilot profile`	`p`, `prof`	List, switch, inspect, rename, and delete profiles
`gpilot organization`	`o`, `org`	List, create, edit, and select organizations
`gpilot service`	`s`, `svc`	Manage the services in the active organization, read request telemetry, and purge the cache
`gpilot deployment`	`d`, `dpl`	Create, list, and cancel deployments
`gpilot persisted-query`	`pq`	Manage the operations you declared for a service
`gpilot apq`	None	List and revoke what clients registered through APQ
`gpilot config`	`c`, `cfg`	Read and convert the service configuration
`gpilot runtime`	None	List proxy runtime versions and read their changelog (no login needed)
`gpilot completion`	None	Generate shell completions

A few common actions also have top-level shortcuts, such as `gpilot deploy` for `gpilot deployment create`; they are listed under shortcuts.

> **`gpilot status` vs `gpilot whoami`**
>
> `gpilot status` is the deployment-list shortcut. To inspect your login/session, use `gpilot whoami` (or its full form `gpilot auth status`, see signing in).

See the full command reference for every subcommand and flag.

Global options

These options are accepted by every command:

Option	Description
`--profile <name>`	Act as this profile instead of the active one. Accepted before or after the subcommand.
`-o, --format <pretty|json|jsonl|yaml>`	Output format for data display. Defaults to `pretty`.
`--no-interactive`	Force non-interactive mode (no prompts or spinners). Usually unnecessary: the CLI auto-detects CI and non-terminal contexts. See interactive vs. non-interactive.
`--no-color`	Disable colored output. Color is on by default when stdout is a terminal; also respects the `NO_COLOR` environment variable.
`-q, --quiet`	Suppress non-essential output.
`-h, --help`	Print help for any command (append `--help` to any subcommand).
`-V, --version`	Print the CLI version.

The machine-readable formats differ in how they print a list:

- `json` prints one JSON document. A list is a single array.
- `jsonl` prints JSON Lines: one compact JSON value per line. Use it to pipe into `jq`, `awk` or a `while read` loop, since each line can be processed as it arrives.
- `yaml` prints one YAML document.

In `json`, `jsonl` and `yaml`, stdout carries only the document. Every status, progress and hint line goes to stderr, including the device URL and code of `gpilot auth login`, which still reach you there. That includes the confirmation line of `deployment cancel`, `auth logout`, `profile use`, `profile rename`, `profile delete`, `organization select` and `service select`. In `pretty` nothing changes.

```
gpilot auth status --format json
gpilot requests --status 500 -o jsonl | jq -r '.path'
```

Interactive vs. non-interactive

The CLI runs interactively (prompts, spinners, colored output) when it detects a terminal. It automatically switches to non-interactive mode when any of the following is true, so you rarely need to pass `--no-interactive` yourself:

- `--no-interactive` is set.
- The `CI` environment variable is set to any value other than empty, `0` or `false`, as most CI systems do (GitHub Actions, GitLab CI, CircleCI, and others).
- `GPILOT_NO_INTERACTIVE` (or `GPILOT_NONINTERACTIVE`) is set to any non-empty value.
- `TERM` is `dumb`.
- stdin or stdout is not a terminal.

In non-interactive mode, spinners become plain log lines and prompts cannot be shown. What a command does when it would have asked for confirmation depends on the command:

Command	Without `-y, --yes` in non-interactive mode
`gpilot deploy` (`deployment create`)	Proceeds, so deployments run unattended in CI.
`gpilot purge` (`service purge`)	Refuses and exits `1`.
`gpilot pq rm` (`persisted-query rm`)	Refuses and exits `1`.
`gpilot apq rm`	Refuses and exits `1`.
`gpilot profile delete`	Refuses and exits `1`.
`gpilot service destroy`	Refuses and exits `1`, naming the command to re-run with `--yes`.
`gpilot service delete`	Refuses and exits `1`, naming the two commands that finish the deletion: one with `--yes` to request the code, one with `--code` to spend it.
`gpilot cancel` (`deployment cancel`)	Refuses and exits `1`, naming the command to re-run with `--yes`.

Pass `-y, --yes` to run any of these from a script or a CI job. A “no” typed at a real prompt still exits `0`.

`gpilot organization create` asks a different question: whether you accept the GraphPilot terms. `--yes` does not answer it, `--accept-terms` does. Without that flag the command creates nothing and exits `1` in non-interactive mode, and also after a “no” at the prompt. See `organization create`.

When the keys come from a pipe (`cat keys.txt | gpilot purge`) but you are at a terminal, `purge` still asks you there before it purges anything.

Exit codes

Exit codes are stable, so scripts can branch on them: `0` success, `1` a general error, `2` a usage error, `3` an authentication failure, `4` a network or GraphPilot API error, `6` a CLI too old to run, `7` a deployment whose outcome is unknown. The full table is on codes.

Environment variables

The CLI reads its configuration from `GPILOT_`-prefixed environment variables. Settings grouped into a section follow the pattern `GPILOT_<SECTION>_<KEY>` (for example `GPILOT_API_URL`); the others have a name of their own.

Variable	Purpose
`GPILOT_TOKEN`	An API key (`sk_…`) used for non-interactive authentication. When set, it takes precedence over the stored profile. See signing in.
`GPILOT_PROFILE`	Selects the active profile without changing the on-disk default.
`GPILOT_NO_INTERACTIVE`	Force non-interactive mode regardless of terminal detection (alias `GPILOT_NONINTERACTIVE`). The generic `CI` variable has the same effect. See interactive vs. non-interactive.
`GPILOT_CONFIG_DIR`	Stores profiles and credentials in this directory instead of the default location.
`GPILOT_OPEN_BROWSER`	Set to `0` or `false` to stop `auth login` from launching a browser. The CLI still prints the URL to open.
`GPILOT_SKIP_VERSION_CHECK`	Set to `1`, `true`, `yes` or `on` to skip the minimum-version check for this run. A temporary escape hatch, see Staying up to date.
`GPILOT_INSTALL_DIR`	Read by the install scripts, not by `gpilot`: the directory they install to. See installing the CLI.
`GPILOT_VERSION`	Read by the install scripts, not by `gpilot`: the release to install instead of the latest. See install a specific version.
`GPILOT_REQUIRE_SIGNATURE`	Read by the install scripts, not by `gpilot`: set to `1` to refuse an install whose signature cannot be checked. See verify a download.

Overrides you normally do not set

These point the CLI at other endpoints, and are for support use only: set them when GraphPilot support asks you to. With them unset, the CLI talks to GraphPilot's production services.

Variable	Overrides	Default
`GPILOT_API_URL`	The GraphPilot API endpoint.	The production GraphPilot API.
`GPILOT_UPDATE_MANIFEST_URL`	Where the minimum-version check reads the published version policy.	`https://raw.githubusercontent.com/GraphPilot/gpilot/main/version.json`

From gpilot 1.1.0, `GPILOT_API_URL` must be an `https://` address, because the requests sent to it carry your credential. Plain `http://` is accepted only for a loopback host (`localhost`, or a loopback address such as

`127.0.0.1` or `[::1]`). Any other address is refused before a request is made, with an error that names the variable, and the command exits with `1`.

> **Tip**
> For CI/CD, set `GPILOT_TOKEN` to an organization API key. Non-interactive mode is auto-detected (via `CI` and friends), so `gpilot deploy` runs without a browser, profile, or prompt, and without needing `--no-interactive` or `--yes`. Commands that purge, revoke or tear something down still need `--yes`.

Where configuration is stored

The CLI keeps configuration and credentials in your platform's standard config directory:

- **macOS:** `~/Library/Application Support/io.graphpilot.cli/`
- **Linux:** `~/.config/gpilot/`, or `$XDG_CONFIG_HOME/gpilot/` when `XDG_CONFIG_HOME` is set
- **Windows:** `%APPDATA%\graphpilot\cli\config\`

gpilot 1.0 kept the Linux store in `~/.config/cli/`; newer versions move it on their first run, so an existing login keeps working.

Set `GPILOT_CONFIG_DIR` to use a different directory. With it set, nothing is moved.

```
<config dir>/
├── config.toml                # active profile + per-profile settings
└── credentials/               # one file per profile (permissions 0600)
    ├── default.json
    └── work.json
```

`config.toml` records the active profile and, for each profile, its selected service and any per-profile `api_url` override. The organization is not copied here: it is carried by the credential itself. The `credentials/` files hold the OAuth tokens or API keys and are written with owner-only (`0600`) permissions. You rarely edit these by hand; the `gpilot auth` and `gpilot profile` commands manage them for you, and `gpilot profile show` prints what a profile holds.

The context line

Commands that act on an organization or a service print a dimmed context line on stderr before their output, so the profile, organization and service being acted on are never a guess:

```
profile acme · org acme-corp · service checkout-api
```

The organization is the one the credential is scoped to, as the credential names it: its slug for an API key profile, its organization id (`org_…`) for a browser or device login. A field with nothing in it reads as a dash, and a service the CLI cannot determine reads `unknown`. Under `GPILOT_TOKEN` there is no profile, and the line names the token instead:

```
profile — (GPILOT_TOKEN) · org org_01H… · service checkout-api
```

It is read from local state before the first request, and goes to stderr so that `gpilot deployment list -o json | jq` stays a stream of JSON. It is suppressed by `--quiet` and by any machine-readable `--format`, but not in CI, where which profile a pipeline deployed as is worth having in the log.

CLI

Installing the CLI

`gpilot` ships as a single self-contained binary for macOS, Linux, and Windows. Once it is installed, log in and follow the quickstart. Pick the method that matches your platform. All of them install the same binary, so you can switch later without uninstalling first.

> macOS
>
> **Homebrew** (recommended):
>
> ```
> brew install graphpilot/tap/gpilot
> ```
>
> **Shell script:**
>
> ```
> curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
> ```
>
> The script installs to `~/.local/bin`. Set `GPILOT_INSTALL_DIR` to choose a different location.

> Linux
>
> **Homebrew** (recommended):
>
> ```
> brew install graphpilot/tap/gpilot
> ```
>
> **Shell script:**
>
> ```
> curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
> ```
>
> The script installs to `~/.local/bin`. Set `GPILOT_INSTALL_DIR` to choose a different location.

> **Windows**
>
> **Scoop** (recommended):
>
> ```
> scoop bucket add graphpilot https://github.com/GraphPilot/gpilot
> scoop install gpilot
> ```
>
> **PowerShell script:**
>
> ```
> irm https://get.graphpilot.io/install.ps1 | iex
> ```
>
> The script installs to `%LOCALAPPDATA%\gpilot\bin`. Set `GPILOT_INSTALL_DIR` to choose a different location.

Both scripts download the release archive from GitHub over HTTPS and install it only if its SHA-256 matches the release's `SHA256SUMS.txt`. When cosign v2.5.0 or later is on your `PATH`, they first check the signature of `SHA256SUMS.txt` and refuse to install if it does not match (see Verify a download). Keep the `https://` in the command: without it, curl and PowerShell fetch the script itself over plain HTTP before any redirect.

Neither script changes your `PATH`. When the install directory is not on it, the script says so at the end; add the directory to your `PATH` to run `gpilot` by name.

Install a specific version

The scripts install the latest release. To install another one, set `GPILOT_VERSION` to its version number (with or without the leading `v`):

```
curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | GPILOT_VERSION=1.0.1
sh
```

```
$env:GPILOT_VERSION = "1.0.1"; irm https://get.graphpilot.io/install.ps1 | iex
```

Check the installation

```
gpilot --version
```

Manual download

Every release publishes prebuilt archives for all platforms on the releases page. Download the archive for your target, extract the `gpilot` binary, and place it anywhere on your `PATH`. The archives are named `gpilot-cli-<version>-<target>`:

Platform	Target	Archive
macOS, Apple silicon	`aarch64-apple-darwin`	`.tar.gz`
macOS, Intel	`x86_64-apple-darwin`	`.tar.gz`
Linux, x86-64	`x86_64-unknown-linux-musl`	`.tar.gz`
Linux, ARM64	`aarch64-unknown-linux-musl`	`.tar.gz`
Windows, x86-64	`x86_64-pc-windows-msvc`	`.zip`

Each release carries a `SHA256SUMS.txt` and, from v1.0.1 on, its signature `SHA256SUMS.txt.sigstore.json`, so you can verify the download.

> **macOS: clearing the quarantine flag**
>
> A binary downloaded through a browser is quarantined by macOS Gatekeeper. Clear it once after extracting:
>
> ```
> xattr -dr com.apple.quarantine ./gpilot
> ```
>
> This is not needed when you install via Homebrew or the shell script, which do not set the quarantine flag.

Verify a download

Each release from v1.0.1 on is signed. `SHA256SUMS.txt.sigstore.json` is a keyless Sigstore signature of `SHA256SUMS.txt`, made by GraphPilot's CLI release workflow at the release's tag and recorded in Sigstore's public transparency log. A valid signature shows that the checksums come from that workflow, not from whoever can upload files to the release, and the checksums in turn cover every archive.

Download the archive, `SHA256SUMS.txt` and `SHA256SUMS.txt.sigstore.json` from the same release, then check the signature with cosign v2.5.0 or later (older versions cannot read the signature format) and the archive against the checksums:

```
VERSION=1.0.2   # the release you downloaded, without the leading v
cosign verify-blob --bundle SHA256SUMS.txt.sigstore.json \
  --certificate-identity "https://github.com/GraphPilot/graphpilot-
proxy/.github/workflows/release-cli.yml@refs/tags/gpilot-cli-v${VERSION}" \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com \
```

```
  SHA256SUMS.txt
sha256sum --check --ignore-missing SHA256SUMS.txt   # macOS: shasum -a 256 --check --
ignore-missing SHA256SUMS.txt
```

cosign prints `Verified OK` for a valid signature. The identity names the release's own tag, so checksums signed for a different version do not pass.

The install scripts run the same check:

- **cosign v2.5.0 or later on the `PATH`:** the signature is checked before the checksum, and the script stops without installing anything if it does not match.
- **No cosign, or an older one:** the script prints a warning with the command above and installs on the checksum alone. The scripts never download cosign themselves.
- **A signed release without its signature file:** the script refuses to install, with or without cosign.
- **Releases before v1.0.1** (installed with `GPILOT_VERSION`) carry no signature. The script warns and relies on the checksum.

To make the signature mandatory, so that a machine without a suitable cosign fails instead of falling back to the checksum, pass `--require-signature` or set `GPILOT_REQUIRE_SIGNATURE=1`:

```
curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh -s -- --require-
signature
```

```
$env:GPILOT_REQUIRE_SIGNATURE = "1"; irm https://get.graphpilot.io/install.ps1 | iex
```

With the signature required, an unsigned release is refused too.

Shell completions

`gpilot` can generate completions for your shell:

```
gpilot completion bash    # or zsh, fish, powershell, elvish
```

The Homebrew formula installs bash, zsh, and fish completions automatically.

Updating

> **Homebrew**
>
> ```
> ```

```
brew upgrade gpilot
```

Scoop

```
scoop update gpilot
```

Script

Re-run the install command; it always fetches the latest release.

```
curl --proto '=https' --tlsv1.2 -fsSL https://get.graphpilot.io | sh
```

Staying up to date

Before any command that talks to the GraphPilot API, the CLI checks the minimum version GraphPilot supports. A new release may be published as the new minimum, and then an older `gpilot` stops working for API commands soon after a new version ships. Plan on upgrading whenever one comes out:

- **Behind the latest release, but supported:** the command runs and prints a notice on stderr that an update is available.
- **Below the minimum:** the command refuses to run and exits with code `6`. The error names the upgrade command for how you installed the CLI (`brew upgrade gpilot`, `scoop update gpilot`, or the install script).

Commands that never leave your machine, such as `gpilot profile` and `gpilot config convert`, are not checked.

The check reads a small version manifest from GitHub (`raw.githubusercontent.com`) and remembers the answer for five minutes. When the manifest cannot be fetched, the last answer it remembers applies, however old. A machine that has never reached it runs without the check.

> ⚠ **Pinned CLI versions in CI**
>
> A CI image or cache that pins an older `gpilot` stops working for API commands once a newer release ships. Upgrade the CLI as part of the job (re-run the install script, or `brew upgrade gpilot`). If you need a pipeline back before you can upgrade it, set `GPILOT_SKIP_VERSION_CHECK=1` for that run. Treat it as a temporary escape, not a setting to keep.

Related

- Signing in: log in after installing.
- CLI overview: global options, exit codes and environment variables.

- Command reference: every command and flag.

CLI

Signing in

The `gpilot` CLI authenticates against GraphPilot. Every authenticated session is **scoped to a single organization**. The credential the CLI uses (an access token or an API key) carries that organization, and all commands run against it.

Log in

```
gpilot auth login
```

With no flags, the CLI asks **how** you want to authenticate and walks you through it. In non-interactive mode it cannot ask, so it refuses and exits `1` unless you pass `--browser`, `--device` or `--api-key` (or set `GPILOT_TOKEN` instead of logging in). You can also pick the method up front:

Browser

`gpilot auth login --browser`

Opens your browser for an OAuth2 Authorization Code (PKCE) login. The CLI starts a temporary local callback server, you sign in, and it captures the result automatically. A request to that callback that does not carry this login's state gets an error page, and the login keeps waiting. Best for everyday use on your own machine.

Device code

`gpilot auth login --device`

Shows a short code and a URL to open on any device. Best for headless or remote machines where no browser is available locally.

API key

`gpilot auth login --api-key`

> Authenticates with an organization API key (`sk_…`). Omit the value to be prompted securely. Best for automation and shared machines.

A successful login makes the profile it logged into the active one. When that profile still holds an unexpired login, `gpilot auth login` says so and does nothing; pass `--force` to log in again. Naming the profile with the global `--profile` option always logs in again.

Choosing an organization

After a **browser** or **device** login, the profile is set to one organization, so the session is immediately usable, with no extra step:

- When you already chose an organization while signing in in the browser, and it is one you belong to, the CLI takes that choice and asks nothing more.
- Otherwise it shows the organizations you belong to and asks which one to work in. Your selection mints an **organization-scoped token** and is saved as the profile's active organization.
- When the account belongs to no organization yet, the login is saved without one, and the CLI says so and names `gpilot organization create` as the next step.

```
✓ Login successful. Profile 'default' is now using organization Acme Corp (acme)
```

Skip the picker with `--org` when you already know the slug:

```
gpilot auth login --browser --org acme
```

> **Nothing is saved until you confirm**
>
> When the CLI asks you to pick an organization, it only stores your credentials once you have picked one and an organization-scoped token has been minted. Canceling the picker leaves you logged out.

API key login

```
# Prompted securely (input hidden):
gpilot auth login --api-key

# Or pass the value directly (e.g. from a secret store):
gpilot auth login --api-key "$MY_KEY"
```

API keys are **pinned to one organization**, so there is no organization picker. The CLI validates the key against the GraphPilot API, reads the organization it belongs to, and uses it. With `--org`, the login is refused when the

key belongs to a different organization. See API keys for how to create an organization key and which permissions to give it.

> **API keys cannot create organizations**
>
> An API-key session (including a personal key) acts within its fixed organization and cannot perform user-level actions such as **creating an organization**. Use an interactive `--browser` or `--device` login for that.

For fully non-interactive use (CI/CD), set the key as an environment variable instead of logging in:

```
export GPILOT_TOKEN="sk_…"
gpilot --no-interactive deployment list
```

When `GPILOT_TOKEN` is set, it takes precedence over any stored profile.

Token lifetime and refresh

How long your session lasts depends on the method you used:

Method	Lifetime	Refresh
Browser (`--browser`)	Short-lived access token (minutes)	**Automatic**, refreshed on the next command
Device code (`--device`)	Short-lived access token (minutes)	**Automatic**, refreshed on the next command
API key (`--api-key`)	Does not expire	Validated by the server on each use
API key from the environment (`GPILOT_TOKEN`)	Does not expire	Validated by the server on each use

For browser/device sessions, the access token is intentionally short-lived. You do **not** need to log in again when it expires: on the next command, the CLI detects the expired token, uses the stored **refresh token** to obtain a new one, and saves it transparently.

> **Why `auth status` can show only a few minutes left**
>
> `gpilot auth status` reports the remaining lifetime of the **currently stored** access token without refreshing it, so a small number like “0h 4m” is normal. The next command that actually calls the API will refresh it automatically. You only need to log in again if the refresh token itself has expired or been revoked.

Check your status

```
gpilot auth status
# or the top-level shortcut:
gpilot whoami
```

```
ℹ Authentication Status
──────────────────────────────────────

  Profile: default
  Method: Browser / device (OAuth)
  Session: logged in
  Valid: yes
  User: ada@example.com (Ada Lovelace)
  Organization: acme (org_01J…)
  Token: eyJh****c2Fk
  Expires in: 0h 4m
```

The output adapts to how you authenticated. It shows the profile (except for a `GPILOT_TOKEN` session, which uses no profile), the auth method, both verdicts below and the masked token. The organization is shown once the API has named it, and the user and token expiry for OAuth logins. Add the global `--profile <name>` to inspect a specific profile: `gpilot --profile acme auth status`.

With `--format json` (or `yaml`) it prints one document instead. The token is masked, as in the pretty output, and fields that do not apply are left out:

```
{
  "authenticated": true,
  "session": "active",
  "valid": true,
  "validity": "valid",
  "method": "oauth",
  "profile": "default",
  "user": "ada@example.com (Ada Lovelace)",
  "organizations": [{ "slug": "acme", "id": "org_01J…" }],
  "token": "eyJh****c2Fk",
  "expires_in_secs": 240
}
```

Field	Values
`authenticated`	`false` only when the API rejected the credential, or nobody is signed in.
`session`	The stored state: `active`, `renewable` (the access token expired, a refresh token is stored) or `expired`.
`valid`	The API's verdict, `true` or `false`. Left out when it was not checked.

Field	Values
`validity`	`valid`, `invalid`, `unverified` (`--offline`), `unreachable`, or `none` when nobody is signed in.
`reason`	Why the credential is `invalid` or `unreachable`.
`method`	`oauth` (browser or device), `api_key` (stored in the profile), `api_key_env` (from `GPILOT_TOKEN`), or `none`.
`expires_in_secs`	Seconds left on the stored access token, for an OAuth login.

Two lines answer two different questions, and they can honestly disagree:

- `Session` is read from the stored credential alone, with no network involved: `logged in`, `logged in (access token expired, renewable)`, or `logged out (session expired…)`.
- `Valid` is the API's verdict on that credential. By default `auth status` **verifies the credential against the API**, so a key you revoked in the portal is reported as `Valid: no` and the command exits `3` (see exit codes). If the API cannot be reached, the line reads `Valid: unknown (…)` and the exit code stays zero: an unreachable API is not a logout.

With nobody signed in (no stored credential and no `GPILOT_TOKEN`), the command says so and exits `3`.

Pass `--offline` to skip the check and report the stored state only. That also avoids spending the single-use refresh token, so it is the safer way to ask "am I logged in?" repeatedly.

> **Revocation has a short delay**
>
> API-key validity is cached for up to 60 seconds, so a key you just revoked may still verify as valid for a brief window. See the API keys guide for details.

Switch organizations

Browser/device sessions can switch the active organization without logging in again:

```
gpilot organization select acme
```

This re-mints an organization-scoped token for the chosen organization. API-key sessions are pinned to their organization and cannot switch.

Log out

```
gpilot auth logout
```

This removes the stored credential for the active profile and leaves the profile itself in place; `gpilot auth logout --all` does the same for every profile. To remove a profile entirely, credential and configuration together, use `gpilot profile delete`. See Profiles for working with more than one identity.

Related

- Profiles: keep several logins side by side.
- API keys: create the key a CI job logs in with.
- CLI overview: `GPILOT_TOKEN` and the other environment variables.

CLI

Profiles

A **profile** is a named slot that holds one identity for the CLI: its credentials (either OAuth tokens or an API key) together with the organization that identity is currently working in. Profiles let you keep several logins side by side and switch between them without re-authenticating.

This is most useful when you work across **separate organizations or identities**. Imagine you are a consultant with a login at two client companies, Acme Corp and Globex. Keep one profile per company and switch with a single command, instead of logging out and back in each time:

```
gpilot profile use acme      # work as your Acme login…
gpilot profile use globex    # …then switch to Globex
```

Each profile remembers its own credentials and organization, so switching is instant and nothing carries over between them.

The layout is modeled on the AWS CLI: a `config.toml` that records the active profile and per-profile settings, plus one credentials file per profile. See where configuration is stored.

The default profile

If you never mention a profile, the CLI uses one called `default`. A plain `gpilot auth login` logs into `default`, and every other command reads from it.

Create or log into a named profile

`--profile` is a global option, accepted by every command. Log in while targeting a specific profile; it is created if it does not exist. Continuing the two-company example, set up one profile per client:

```
gpilot --profile acme auth login     # your login at Acme Corp
```

```
gpilot --profile globex auth login   # your login at Globex
```

Each profile carries its own credentials and its own selected organization, so `acme` and `globex` can point at different organizations and even different identities (one OAuth, one API key).

A successful login also makes that profile the active one, so after the two commands above `globex` is active. With `--profile`, `auth login` always logs in again, even when the profile still holds a valid login.

A newly created profile name may contain letters, digits, `.`, `_` and `-`, up to 64 characters. Profiles created by older CLI versions keep working under whatever name they were given.

List profiles

```
gpilot profile list
```

One row per configured profile, with a `*` marking the active one, plus its organization, authentication method (`oauth` or `api_key`) and the email behind an OAuth login. It reads the profile store only and never calls the GraphPilot API.

With `-o json` it prints one array of `{ profile, active, kind, organization, email }`, the field names `profile show` uses; `-o jsonl` prints one object per line and `-o yaml` one document. `kind` is `oauth`, `api_key`, or `error` when the stored credential cannot be read; `organization` and `email` are `null` when the profile holds none. An empty store is `[]`.

Inspect one profile

```
gpilot profile show           # the active profile
gpilot profile show globex    # a specific one
```

The offline counterpart to `auth status`: it reports what is on disk (organization, selected service, email, remaining token lifetime, and an `api_url` override when one is set) without asking the GraphPilot API whether the credential is still accepted. That is the answer you want when the network is the thing that is failing.

With `--format json` (or `yaml`) it prints one document:

```
{
  "profile": "globex",
  "active": false,
  "kind": "oauth",
  "organization": "org_01J…",
  "service": "checkout-api",
  "email": "ada@example.com",
  "expires_in_secs": 240
```

```
}
```

`kind` is `oauth` or `api_key`. `organization`, `service`, `email` and `expires_in_secs` are `null` when the profile holds none (an API key has no email or expiry). `api_url` appears only when the profile overrides the API endpoint. A profile with no stored credential is an error, not an empty document.

Switch the active profile

```
gpilot profile use globex
```

This makes `globex` the default for subsequent commands by updating `config.toml`. Only a profile that holds a stored credential can be made active; for any other name the command exits `1` and names the `gpilot --profile <name> auth login` that creates it.

Per-command and environment overrides

You can also select a profile without changing the on-disk default:

- **Per command**, with the global `--profile` flag:

  ```
  gpilot --profile globex auth status
  ```

- **For a shell session**, via the environment variable:

  ```
  export GPILOT_PROFILE=globex
  ```

The CLI resolves which profile to use in this order: a `--profile` flag, then `GPILOT_PROFILE`, then the `active_profile` from `config.toml`. The one exception to “without changing the on-disk default” is `auth login`: a successful login makes the profile it logged into the active one.

> **`GPILOT_TOKEN` bypasses profiles**
>
> If `GPILOT_TOKEN` is set, the CLI authenticates with that API key directly and does not consult any profile. This is the recommended path for CI/CD; see signing in.

Rename a profile

```
gpilot profile rename globex globex-eu
```

Renames the profile in place, credentials and configuration together, without logging in again. It is refused if the target name already exists.

Log out versus delete

The two are different, and the difference is what stays behind:

```
gpilot auth logout                # clear the credential, keep the profile
gpilot profile delete globex      # remove the profile entirely
```

`auth logout` drops the stored credential for the active profile and leaves the profile standing, including its selected service, so logging back in resumes where you were. Use `--profile` (or `GPILOT_PROFILE`) to target a different one, or `gpilot auth logout --all` to clear every profile's credential at once.

`profile delete` removes the credential **and** the `config.toml` entry. It takes several names at once, or `--all` for every configured profile, validates every name before deleting any of them, and asks for confirmation naming the organization and service about to be lost. In a non-interactive shell, where the prompt cannot be shown, it refuses unless you pass `--yes`. Deleting the active profile falls back to `default`.

Know which profile you are in

Commands that act on an organization or a service print a context line on stderr naming the profile, organization and service before their output, including in CI. See the context line.

Related

- Signing in: log in, check your status, and log out.
- Where configuration is stored: the files behind each profile.
- Command reference: every `gpilot profile` subcommand.

CLI

Command reference

This page lists every command the `gpilot` CLI provides. Append `--help` to any command for its complete, authoritative set of flags. All commands also accept the global options (`--profile`, `--format`, `--no-interactive`, `--no-color`, `--quiet`).

> **Tip**
>
> Most resource commands operate on a **currently selected** organization and service. Use `organization select` and `service select` to choose them once, or pass an override flag (such as `--service`) per command.

Shortcuts

A few common actions have top-level shortcuts. Each one is also shown on the subcommand it expands to, below.

Command	Expands to
`gpilot login`	`gpilot auth login`
`gpilot logout`	`gpilot auth logout`
`gpilot whoami`	`gpilot auth status`
`gpilot deploy`	`gpilot deployment create`
`gpilot status`	`gpilot deployment list`
`gpilot cancel`	`gpilot deployment cancel`
`gpilot requests`	`gpilot service requests`
`gpilot purge`	`gpilot service purge`

Command groups and subcommands have aliases, listed with each one below. `gpilot pq` is `gpilot persisted-query`, and `pq ls` is the same command as `pq list`. The rest of the documentation writes the subcommands by their names (`gpilot pq list`, `gpilot pq push`, `gpilot pq rm`, `gpilot apq ls`, `gpilot apq rm`).

Not in the CLI

Deleting an organization is done in the portal only. Deleting a service is different: `gpilot service delete` deletes it, with the emailed code. There is no `gpilot settings` command either (the CLI refuses it as an unknown command, with exit `2`): the CLI's own settings are its environment variables and profiles. A schema reaches GraphPilot with `gpilot deploy`; `gpilot schema pull` only reads one.

gpilot auth

Log in, log out, and inspect the current session. Managing the profiles those credentials live in is `gpilot profile`. See Signing in and Profiles for the full workflow.

login

shortcut

```
gpilot login
```

```
gpilot auth login
```

Log in via browser, device code, or API key.

Argument	Description
`--browser`	Log in via browser (OAuth2 authorization code flow).
`--device`	Log in via device code, for headless or CI environments.
`--api-key [KEY]`	Log in with an organization API key. Omit `KEY` to be prompted securely.
`--org <slug>`	Skip the organization picker and use this slug directly. With `--api-key`, the login is refused when the key belongs to a different organization.
`-f, --force`	Force re-authentication even if already logged in.

`--browser`, `--device` and `--api-key` exclude each other. Without any of them the command asks which method to use; in non-interactive mode it refuses and exits `1`.

The global `--profile <name>` option chooses which profile to log into; it is created if it does not exist: `gpilot --profile acme auth login`. A successful login makes the profile active.

logout

shortcut

```
gpilot logout
```

```
gpilot auth logout
```

Remove the stored credential for the active profile, leaving the profile itself (and its selected service) in place. To remove a profile entirely, use `profile delete`.

Argument	Description
`--all`	Log out of every profile. Cannot be combined with `--profile`.

status

shortcut

```
gpilot whoami
```

```
gpilot auth status
```

Show the current authentication status, verifying validity against the API.

Argument	Description
`--offline`	Skip the online validity check and report the stored state only.

With `-o json` or `-o yaml` it prints one document; its fields are listed under check your status.

Pass the global `--profile <name>` to inspect a profile other than the active one: `gpilot --profile acme auth status`.

gpilot profile

Aliases: `p`, `prof`. Manage the named slots the CLI stores credentials in. Every subcommand here reads and writes local state only and never calls the GraphPilot API. See Profiles.

list

```
gpilot profile list
```

List configured profiles: the active one (marked `*`), its organization, authentication method, and the email behind an OAuth login.

No arguments.

show

```
gpilot profile show [name]
```

Show a profile's stored context: organization, selected service, email, remaining token lifetime, and an `api_url` override when one is set. The offline counterpart to `auth status`. Its `-o json` document is shown under inspect one profile.

Argument	Description
`[name]`	Profile to show. Defaults to the active profile.

use

```
gpilot profile use <name>
```

Switch the active profile.

Argument	Description
`<name>`	Profile to make active.

rename

```
gpilot profile rename <old> <new>
```

Rename a profile, credential and configuration together, without logging in again. Refused if `<new>` already exists.

Argument	Description
`<old>`	Profile to rename.
`<new>`	New name.

delete

```
gpilot profile delete <name> ...
```

Remove profiles whole: the credential and the `config.toml` entry. Every name is validated before anything is deleted. Deleting the active profile falls back to `default`.

Argument	Description
`<name>...`	Profiles to delete. Required unless `--all` is given.
`--all`	Delete every configured profile.
`-y, --yes`	Skip the confirmation prompt. Required in non-interactive shells.

gpilot organization

Aliases: `o`, `org`. Manage organizations and choose the one the CLI works in.

Organizations are deleted in the portal, which asks for a confirmation code sent to you by email. Deleting an organization is not available from the CLI.

list

```
gpilot organization list
```

List all organizations you can access. **Alias:** `ls`.

No arguments.

show

```
gpilot organization show <slug>
```

Show a single organization.

Argument	Description
`<slug>`	Organization slug.

create

```
gpilot organization create --name "Your Company" --accept-terms
```

Create a new organization. **Alias:** `new`.

Argument	Description
`-n, --name <name>`	Organization name (required).
`-s, --slug <slug>`	Slug; auto-generated from the name if omitted.
`-w, --website-url <url>`	Website URL.
`-i, --industry <industry>`	Industry.
`--accept-terms`	Accept the GraphPilot terms in force. Required, see below.

Creating an organization accepts the GraphPilot terms in force. The command reads the current version from the API (`legalTerms`) and accepts exactly that version, and only when you pass `--accept-terms`. Without the flag it creates nothing, prints the version, the URL of the terms and the URL of the privacy policy (linked for information, it is not accepted), and exits `1`. In an interactive terminal it asks instead, and the answer defaults to no. In non-interactive mode, as in CI, the flag is the only way to accept; `--yes` does not accept the terms.

If the terms change between the moment the command reads them and the moment it creates the organization, the API refuses with `TERMS_NOT_ACCEPTED` and nothing is created. The command names the version now in force and exits `1`; run it again with `--accept-terms` to accept that version.

> **Note**
> Not available for API-key sessions: creating an organization is a user-level action that needs an interactive user login (`--browser` or `--device`).

edit

```
gpilot organization edit <slug>
```

Edit an organization.

Argument	Description
`<slug>`	Organization to edit.
`--name <name>`	New name.
`--new-slug <slug>`	New slug.
`--website-url <url>`	New website URL.
`--industry <industry>`	New industry.

select

```
gpilot organization select [slug]
```

Set the active organization. **Alias:** `use` .

Argument	Description
`[slug]`	Organization slug. Interactive selection if not provided.

> **Note**
> For API-key sessions the organization is fixed by the key, so `organization select` will not switch it. For browser/device sessions, selecting re-mints an organization-scoped token.

gpilot service

Aliases: `s` , `svc` . Manage the services in the active organization.

Services are deleted with `service delete` or in the portal; both ask for a confirmation code sent to you by email. To take a service off the edge but keep it, use `service destroy` .

list

```
gpilot service list
```

List services in the active organization. **Alias:** `ls`.

No arguments.

show

```
gpilot service show <subdomain>
```

Show a single service: its name, subdomain, URL, origin URL and environment.

Argument	Description
`<subdomain>`	Service subdomain.

URL is the service's primary address: its custom domain when one is primary, otherwise `https://<subdomain>.<label>.graphpilot.cloud` under your organization's hostname label (`https://<subdomain>.graphpilot.cloud` for a service created before labels). `gpilot service list` prints the same column. Other custom domains and the service ID are shown in the portal, not by this command.

create

```
gpilot service create
```

Create a new service. **Alias:** `new`.

Argument	Description
`-n, --name <name>`	Service name (required).
`-s, --subdomain <subdomain>`	Subdomain, unique within the organization; auto-generated from the name if omitted. The service answers on `<subdomain>.<label>.graphpilot.cloud`, which `service create` prints as **URL**.
`-u, --url <url>`	Origin URL (required). Scheme, host and optional port only; a URL with a path, query or fragment is refused. Without a port, `https://` connects to `443` and `http://` to `80`.
`-e, --environment <env>`	Environment, e.g. `production` or `staging`. Defaults to `production`.
`--color <hex>`	Display color as a hex code, e.g. `#FF5733`. Defaults to `#3B82F6`.

| `--deploy` | Deploy right away without asking: no schema, the default configuration, then GraphQL detection through the service's hostname. Conflicts with `--no-deploy`. | | `--no-deploy` | Only create the service. | | `--keep-plain-http` | Never apply a detected schema and configuration: finish the setup with the service as plain HTTP. | | `--detach` | Start the first deployment and return without following the setup. | | `--timeout <duration>` | Maximum time to follow the setup, such as `30m` or `1h`. Defaults to `30m`. Time spent answering a question does not count. Nothing is canceled when it runs out. | | `-y, --yes` | Answer every question with its default: deploy, apply the detected schema and configuration, and keep probing an origin the edge cannot reach, every 30 seconds, until `--timeout`. |

`--name` and `--url` are required; the command does not prompt for them.

The command prints the service's signing key once, as the **Signing key** row (`signing_key` with `-o json` or `-o yaml`), and says so on stderr. GraphPilot never shows it again, so store it right away; if you lose it, rotate it in the portal. See the signing key. In the default format the table with the key is printed as soon as the service exists, before the setup starts.

Deploying right away

`service create` runs the same setup as the portal: a first deployment without a schema and without a configuration file, then GraphQL detection, then, if GraphQL was found, a second deployment with the detected schema and configuration.

1. **Whether to deploy.** `--no-deploy` creates the service only. Without a terminal and without `--deploy` or `--yes`, it also creates only, as before. Otherwise the CLI first asks GraphPilot whether the edge can reach the origin URL; a private or local address is not deployed, and the CLI prints `Your origin is on a private address that the edge cannot reach. Deploy once it is reachable from the internet.` and creates only. With `--deploy` or `--yes` it deploys; otherwise it asks `Deploy right away? (no schema, default configuration; GraphPilot then looks for GraphQL on your origin)`, default yes.
2. **Following the setup.** With `--detach` it prints `Deploying. Follow the setup with: gpilot service setup --service <subdomain>` and exits. Otherwise it prints GraphPilot's sentence for each state of the setup, once, on stderr, and then:
   - **GraphQL found, with a schema:** a table of the probed paths (**Path**, **Status**, **Result**), `GraphQL answers on <path>; introspection read a schema with <n> types.`, and, when the suggestion sets a path, `The suggested configuration sets request.graphql_paths to ["<path>"].` It asks `Apply the suggested configuration and schema? This deploys the service again, with its GraphQL features on`, default yes. No, or `--keep-plain-http`, keeps the service as plain HTTP.
   - **GraphQL found without a schema, or no GraphQL:** the findings, with `No GraphQL endpoint was detected, so the service runs as plain HTTP. If your API answers GraphQL on another path, set request.graphql_paths in gpilot.toml and deploy with: gpilot deploy` when nothing answered. When introspection failed, it suggests `gpilot schema pull` with the credentials your API needs. It asks `Keep the service as plain HTTP?`, default yes; no leaves the setup open.
   - **The edge could not reach the origin:** the edge's IPv4 and IPv6 ranges to allow at your firewall. It asks `Retry detection?`, default yes; with `--yes` it probes again every 30 seconds.

- **Without a terminal and without** `--yes`, a question is left open rather than answered: the CLI prints `The setup of service '<subdomain>' is waiting for your decision.` with the `gpilot service setup` command that continues it, and exits `0`.

3. **At the end** it prints a summary: `Setup done.` with the outcome (`The service serves GraphQL with the detected schema`, `The service runs as plain HTTP` or `A deployment of your own replaced the automatic one`), or the state the setup is in, and the rows **Service**, **URL**, **Setup**, **GraphQL path**, **Schema types** and **Deployment**.

With `-o json`, `-o yaml` or `-o jsonl` it prints one document at the end, also when the setup failed, so the signing key is never lost: the service's `name`, `subdomain`, `url`, `origin_url`, `environment` and `signing_key`, and `setup` (`null` without one), with `service`, `url`, `state`, `outcome`, `message`, `errorCode`, `graphqlPath`, `typeCount`, `initialDeploymentId` and `appliedDeploymentId` in the API's words (`DONE`, `GRAPHQL`). Progress and questions go to stderr.

Exit code	When
`0`	Created, set up, skipped for a private origin, detached, or a question was left open.
`1`	The setup failed (`The setup of service '<subdomain>' failed: <message>`), the origin stayed unreachable and no retry was taken, or `--timeout` ran out (`... is still <STATE> after --timeout ran out. Nothing was cancelled`). Each names the command to continue with.
`130`, `143`	You stopped following the setup with Ctrl+C (`130`) or `SIGTERM` (`143`, Unix). The command prints its normal output first (with `-o`, the one document, signing key included) and says `Stopped following the setup of service '<subdomain>'; it goes on at GraphPilot` with the `gpilot service setup` command to follow it. The setup is not canceled.
`2`	A usage error, such as `--deploy` with `--no-deploy`.
`3`	Authentication failed.
`4`	A network or API failure.

setup

```
gpilot service setup
```

Follow or resume the setup of a service: the same steps, questions, output and exit codes as `service create` from the moment the service exists, except `130` and `143`, which only `service create` returns. At the end it prints the summary, or with `-o` a document of the setup alone.

Argument	Description
`--service <subdomain>`	Override the active service.

Argument	Description
`--retry`	Probe the origin again first. Also starts a setup for a service created without deploying, once one of its deployments has completed.
`--keep-plain-http`	Never apply a detected schema and configuration.
`-y, --yes`	Answer every question with its default.
`--timeout <duration>`	Maximum time to follow the setup. Defaults to `30m`. Time spent answering a question does not count.

A service without a setup is refused with `Service '<subdomain>' has no setup to follow: it was created without deploying` and exit `1`; start one with `--retry`.

edit

```
gpilot service edit <subdomain>
```

Edit a service.

Argument	Description
`<subdomain>`	Service subdomain to update.
`--name <name>`	New name.
`--new-subdomain <subdomain>`	New subdomain.
`--url <url>`	New origin URL.
`--environment <env>`	New environment.
`--color <hex>`	New color (hex code).

select

```
gpilot service select [subdomain]
```

Set the active service. **Alias:** `use`.

Argument	Description
`[subdomain]`	Service subdomain. Interactive selection if not provided.

delete

```
gpilot service delete <subdomain>
```

Delete a service and tear down its edge infrastructure. **Alias:** `rm`. Its deployments, schema and configuration go with it, and **this cannot be undone**. To take a service off the edge but keep it, use `service destroy` instead.

Argument	Description
`<subdomain>`	Service subdomain to delete.
`--code <code>`	The confirmation code GraphPilot emailed for this deletion. Skips requesting a new code and the confirmation prompt.
`-y, --yes`	Skip the confirmation prompt before the code is requested.

GraphPilot confirms a deletion with a code it emails to the person asking. Without `--code`, the command says what is about to be deleted, asks you to confirm, has GraphPilot email you a code and asks for it. Once the code is accepted, the service is deleted and the command exits `0`; its edge infrastructure is torn down in the background.

- **It needs a user login.** With an API key (an API-key profile or `GPILOT_TOKEN`) the command refuses before sending anything and exits `1`, because a key has no inbox. Sign in with `gpilot auth login`, or delete the service in the portal.
- **Without a terminal to ask on**, it refuses without `--yes` and exits `1`. With `--yes` it requests the code, has nowhere to type it in, and exits `1` without deleting; finish with `gpilot service delete <subdomain> --code <code>`.
- **A code that is not accepted** (mistyped, expired or already used) exits `1` and deletes nothing. Run again with `--code` for a typo, or without it for a new code.

destroy

```
gpilot service destroy <subdomain>
```

Tear down the deployed infrastructure of a service while keeping the service itself, its schema, its config and its history. Deploy again to bring it back.

Argument	Description
`<subdomain>`	Service whose deployed infrastructure should be torn down.
`-y, --yes`	Skip the confirmation prompt. Required in non-interactive runs, where the command otherwise refuses and exits `1`.

The teardown runs in the background, so the command reports that the request was accepted rather than waiting for it.

runtime

```
gpilot service runtime show
gpilot service runtime pin <version>
gpilot service runtime unpin
```

Show or change the proxy runtime version a service runs. See Runtime versions.

Subcommand	Description
`show`	Shows the version the service runs now, whether it follows stable or is pinned, and the version its next deployment gets. For a service pinned to a withdrawn version, whose deployments are refused, that last row reads `Unknown (no runtime version resolvable)`.
`pin <version>`	Keeps the service on one version. A withdrawn or unknown version is refused.
`unpin`	Lets the service follow the stable version again.

All three take `--service <subdomain>` to override the active service.

`pin` and `unpin` redeploy the service, so they need `service:edit` and `deployment:edit`.

requests

shortcut

```
gpilot requests
```

```
gpilot service requests [filters] [--follow | --wait[=N]]
```

Inspect and tail request telemetry. **Alias:** `req`. Without flags it prints the last 50 matching requests from the past 15 minutes.

It reads stored request telemetry, so it shows only requests that were stored; see what Insights needs. The filters need the same plan as their Insights counterparts; a filter your plan does not include is refused.

Argument	Description
`--path <glob>`	Filter by request path. Wildcards supported, repeatable, ORed.
`--operation <name>`	Filter by GraphQL operation name. Wildcards supported, repeatable, ORed.
`--method <method>`	Filter by HTTP method. Repeatable, ORed.

Argument	Description
`--status <code>`	Filter by HTTP response status. Repeatable, ORed.
`--correlation-id <id>`	Filter by correlation id, the value the client sent in `GP-Correlation-ID` (see `telemetry.correlation_header`). Exact match, repeatable, ORed. Needs Airliner (Business) or above; on a smaller plan the API refuses the request with `FORBIDDEN` and the command exits `4`.
`--request-id <id>`	Filter by request id, the value of the `gp-request-id` response header. Exact match, repeatable, ORed. Needs Airliner (Business) or above; on a smaller plan the API refuses the request with `FORBIDDEN` and the command exits `4`.
`--surrogate-key <key>`	Filter to responses that carried this surrogate key. Exact match, repeatable, ORed. Needs Airliner (Business) or above, like `--request-id`.
`--wide`	Add the request id, the cache reason (pass reason, else skip reason), the cache key and the surrogate keys to the table.
`--ip[=<ip>]`	Filter by client IP. Bare `--ip` uses your public IP as GraphPilot's API sees it (`Query.clientIp`), looked up once at start; no third-party service is asked. When the API knows no address for your connection, the command fails with `Pass it explicitly with --ip=<address>.` rather than listing unfiltered rows. The match is exact: on a service whose `telemetry.client_ip` is `"truncated"`, pass the truncated address; with `"none"`, nothing matches.
`--since <dur>`	How far back to look. Default `15m`.
`--limit <n>`	Maximum rows per query. Default `50`.
`-f, --follow`	Tail new matching requests as they arrive. How long it runs is below the table.
`--wait[=N]`	Wait until N matching requests exist, then print and exit. Default `N=1`.
`--timeout <dur>`	Maximum time to wait in `--wait` mode. Default `120s`.
`--poll-interval <dur>`	Poll interval for `--wait`. Default `3s`. Ignored by `--follow`, which the server paces.
`--service <subdomain>`	Override the active service.

`--wait` is what you want in a test or a CI step: it blocks until the request you are expecting shows up rather than racing it.

The server closes a live stream after five minutes without a new matching request. `--follow` reconnects on its own when that happens, picking up after the last row it printed without repeating one, so it runs until you stop it with Ctrl+C. It stops earlier, with an error, when the server answers with an error (a credential that stopped working, for example), the connection closes, or reconnecting fails.

The table shows time, status, method, latency, cache status, path and correlation ID, then the GraphQL operation name (OPERATION) as the last column, never cut short; `--wide` adds REQUEST ID, REASON, CACHE KEY and SURROGATE KEYS before it. On a narrow terminal the correlation ID, cache status, method and path columns are

dropped, in that order, until the line fits; `--wide` drops nothing. With `-o json`, `-o jsonl` or `-o yaml`, each row carries `id`, `timestamp`, `correlation_id`, `method`, `path`, `status`, `cache_status`, `operation`, `latency_ms`, `cache_key`, `pass_reason`, `skip_reason` and `surrogate_keys`. A field your plan does not list comes back empty (`null`): the cache key and the surrogate keys are listed from Airliner (Business); the cache status and the reasons are listed on every plan. `--follow` streams only as the table or as `-o jsonl`; `json` and `yaml` produce a single document and cannot tail.

To find one request, filter by its request id (`--request-id`) or its correlation ID. Purge events are not part of this output; the portal shows them.

purge

shortcut

```
gpilot purge
```

```
gpilot service purge [<key> ... ] [--all] [--soft]
```

Purge cached responses. See invalidation for what a key is and which one to reach for.

Argument	Description
`[<key> ... ]`	Surrogate keys to purge. Reads from stdin, one key per line, when none are given and stdin is not a terminal.
`--all`	Purge the service's entire cache.
`--soft`	Mark entries stale instead of evicting them, so a revalidation serves rather than a miss. Conflicts with `--all`.
`--service <subdomain>`	Override the active service.
`-y, --yes`	Skip the confirmation prompt. Required in non-interactive runs, where the command otherwise refuses and exits `1`.

Every purge asks for confirmation, naming the service and its environment. Reading keys from stdin lets you purge a list produced by another tool:

```
sort -u keys.txt | gpilot purge --yes
```

With `-o json` (or `yaml`) it prints the API's own purge payload, field names included:

```
{
  "success": true,
```

```
  "purgedKeyCount": 2,
  "purgedKeys": ["User:id:1", "User:id:2"]
}
```

`purgedKeyCount` counts keys, not cached responses. A purge-all reports `0` and an empty list.

gpilot runtime

Look up the proxy runtime versions. Neither command needs a login. To see or change what a service runs, use `service runtime`.

versions

```
gpilot runtime versions
```

List the runtime versions, newest first, with the stable one marked. **Alias:** `ls`.

Argument	Description
`--include-yanked`	Also list withdrawn versions, with the reason they were withdrawn.

changelog

```
gpilot runtime changelog --from <version> [--to <version>]
```

Print what changed in the runtime after a version.

Argument	Description
`--from <version>`	The version you run now; the changelog starts after it (required).
`--to <version>`	The last version to include. Defaults to the current stable version.

gpilot deployment

Aliases: `d`, `dpl`. Create and manage deployments for the active service.

create

shortcut

```
gpilot deploy
```

```
gpilot deployment create
```

Create a new deployment. **Alias:** `new`.

Argument	Description
`-s, --schema <file>`	Path to the GraphQL schema file. Optional; defaults to `schema.graphql` or `schema.gql` in the current directory. When neither exists, the service is deployed without a schema, as a plain HTTP service with GraphQL features off. A path given here that does not exist is an error.
`-c, --config <file>`	Path to the service config: TOML, JSON, JSON5 or YAML, picked by extension. Anything that is not TOML is converted before it is uploaded. Optional; defaults to the first of `gpilot.toml`, `gpilot.tml`, `gpilot.json`, `gpilot.json5`, `gpilot.yaml` and `gpilot.yml` in the current directory. A `[runtime]` `version` in it applies to this deployment only.
`--detach`	Create the deployment and return immediately, without waiting for it to finish. **Alias:** `--no-wait`.
`--timeout <duration>`	Maximum time to wait for the deployment when not using `--detach`, watching its progress and, if the stream ends early, checking it until it ends. Accepts human durations such as `20m`, `1h 30m`, or `900s`. Defaults to `20m`.
`--no-schema`	Deploy as a plain HTTP service even when `schema.graphql` or `schema.gql` is present. Conflicts with `--schema`. Prints the `No schema found` line below.
`--default-config`	Deploy on GraphPilot's default configuration: no config file is read or uploaded, and no `[runtime]` `version` applies. The summary shows `Config: GraphPilot defaults, no config file`. Conflicts with `--config`.
`--force-build`	Force a rebuild even if nothing changed.
`--service <subdomain>`	Override the active service.
`-y, --yes`	Skip the confirmation prompt.

By default `gpilot deploy` waits for the deployment to finish, streaming per-step progress (build, edge deploy, TLS verification, activation) as it runs, and exits non-zero if the deployment fails or the wait exceeds `--timeout`. If the progress stream ends before the deployment does, it keeps checking the deployment within the same `--timeout`, and exits `7` only if the deployment is still open when that runs out. Pass `--detach` to create the deployment and return immediately, leaving it to run in the background.

In a non-interactive context the confirmation prompt is skipped automatically, so deploys run unattended in CI without `--yes` (see interactive vs. non-interactive).

When `--schema` or `--config` are omitted, the CLI looks for the conventional file names in the current directory (the first match wins). This lets you run `gpilot deploy` with no arguments from a project that follows the standard layout. If no config file is found, the command exits with an error naming the candidates it looked for;

pass `--default-config` to deploy without one. A deployment without a config file is printed with `Config: default configuration` in place of its config version.

A schema is optional. Without `--schema` and without a `schema.graphql` or `schema.gql`, the command deploys the configuration alone, as a plain HTTP service, and says so before it asks you to continue:

```
No schema found: deploying as a plain HTTP service, GraphQL features are off.
```

Nothing is uploaded for the schema, and the schema the service had before is not reused. When the live deployment had a schema, the summary of what changed shows `Schema removed: GraphQL features off`. The next deployment with a schema turns GraphQL on again. When the build completes, its diagnostics include a notice that the service has no schema; notices are printed as notices, not as errors, and never fail the command.

With `-o json`, `-o jsonl` or `-o yaml`, `gpilot deploy` prints one document as soon as the deployment exists: `id`, `status` (lowercase, as in `gpilot status -o json`), `created_at`, `schema_version` and `config_version`. Progress goes to stderr, and the outcome is the exit code; read the result afterwards with `gpilot status -o json` (below).

list

shortcut

```
gpilot status
```

```
gpilot deployment list
```

List deployments for the active service. **Alias:** `ls`.

Argument	Description
`--service <subdomain>`	Override the active service.

With `-o json`, `-o jsonl` or `-o yaml`, each deployment carries `id`, `status`, `created_by` (who requested it: a person by email, an API key by its name and redacted value such as `CI (sk_…9W8H)`, or empty when nothing was recorded), `created_at` and `completed_at` (empty while it runs). `status` is one of `started`, `pending`, `completed`, `failed` or `cancelled`, and `unknown` for a status this CLI version does not know.

```
gpilot status -o json | jq -r '.[0].status'
```

cancel

shortcut

```
gpilot cancel
```

```
gpilot deployment cancel [id]
```

Cancel a deployment.

Argument	Description
`[id]`	Deployment ID. Interactive selection if not provided.
`--service <subdomain>`	Override the active service.
`-y, --yes`	Skip the confirmation prompt. Required in non-interactive runs, where the command otherwise refuses and exits `1`.

gpilot persisted-query

Alias: `pq`. Manage the operations you declared for a service: the ones pushed from a manifest. What clients registered on their own through APQ is managed with `gpilot apq`. See Persisted queries for how they work and how to configure them.

list

```
gpilot persisted-query list [--search <text>] [--sort <name|hash>] [--documents] [--limit <n> | --all]
```

List the operations declared for a service (pushed from a manifest). Client registrations are not included; list those with `apq ls`. Alias: `ls`.

Argument	Description
`--search <text>`	Keep only operations whose name contains this text, case-insensitively. Matched literally: `%` and `_` are ordinary characters. Anonymous operations have no name and never match.
`--sort <name|hash>`	Order the listing by operation name (A to Z, anonymous operations last) or by hash. Defaults to `name`.
`--documents`	Also print each entry's GraphQL document.
`--limit <n>`	Stop after this many entries. Defaults to 50.
`--all`	List the whole manifest. Conflicts with `--limit`.
`--service <subdomain>`	Override the active service.

show

```
gpilot persisted-query show <hash>
```

Print one registered document, whether it was pushed or registered by a client.

Argument	Description
`<hash>`	SHA-256 of the document, hex encoded.
`--service <subdomain>`	Override the active service.

push

```
gpilot persisted-query push <path> ... [--as <format>] [--rehash] [--dry-run]
```

Register operations from a manifest. Additive and idempotent: nothing is removed, and re-sending an entry is a no-op.

Argument	Description
`<path> ...`	Manifests, `.graphql` sources, directories or globs. Repeatable. An argument is read as a glob only when no file or directory of that exact name exists, so an existing path such as `src/app/[id]/q.graphql` is read as it is, `[`, `*` and `?` included.
`--as <format>`	Read the files as this format instead of detecting it from their contents: `apollo`, `map`, `relay-array`, `operation-output` or `graphql-source`. `formats` says which tool produces which.
`--rehash`	Register documents under GraphPilot's hash even when the manifest declares a different id. Only correct when you are regenerating your clients too; on its own it registers documents under ids no client sends.
`--dry-run`	Parse, verify and report, but write nothing.
`--service <subdomain>`	Override the active service.

rm

```
gpilot persisted-query rm <hash> ... | --from-file <file> | --all
```

Revoke operations: delete the registry entries and purge everything cached for them. Clients still sending these hashes will break, which is the intent. Alias: `remove`.

Argument	Description
`<hash>...`	Hashes to revoke, hex encoded.
`--from-file <file>`	Read hashes from a file, one per line.
`--all`	Revoke everything the registry holds.
`--service <subdomain>`	Override the active service.
`-y, --yes`	Skip the confirmation prompt. Required in non-interactive runs, where the command otherwise refuses and exits `1`.

Exactly one of the three targets is required. To revoke only what clients registered, use `apq rm --all`.

formats

```
gpilot persisted-query formats
```

List the manifest formats `push` accepts, with the tools that produce each one and a link to their documentation.

No arguments.

gpilot apq

Manage what clients registered at the edge themselves through Automatic Persisted Queries (APQ), as opposed to the operations you pushed. See Persisted queries.

ls

```
gpilot apq ls [--limit <n> | --all]
```

List the hashes clients registered that your manifest does not declare. The listing shows hashes only; adopt an operation with `pq push` to give it a name. Alias: `list`.

Argument	Description
`--limit <n>`	Stop after this many entries. Defaults to 50.
`--all`	List every registration. Conflicts with `--limit`.
`--service <subdomain>`	Override the active service.

show

```
gpilot apq show <hash>
```

Print one registered document. The same command as `persisted-query show`.

Argument	Description
`<hash>`	SHA-256 of the document, hex encoded.
`--service <subdomain>`	Override the active service.

rm

```
gpilot apq rm <hash>... | --from-file <file> | --all
```

Revoke client registrations: delete the registry entries and purge everything cached for them. Clients still sending these hashes will break. To keep an operation, adopt it with `pq push` first and clear the rest afterwards. Alias: `remove`.

Argument	Description
`<hash>...`	Hashes to revoke, hex encoded.
`--from-file <file>`	Read hashes from a file, one per line.
`--all`	Revoke every client registration your manifest does not declare.
`--service <subdomain>`	Override the active service.
`-y, --yes`	Skip the confirmation prompt. Required in non-interactive runs, where the command otherwise refuses and exits `1`.

Exactly one of the three targets is required.

gpilot schema

Read a GraphQL schema as SDL. It deploys nothing: a schema reaches GraphPilot with `gpilot deploy`.

pull

```
gpilot schema pull
gpilot schema pull --url https://api.example.com/graphql --header 'Authorization: Bearer ...' --output schema.graphql
```

Print the schema the active service is deployed from, or, with `--url`, the schema your API answers introspection with.

Argument	Description
`--url <url>`	Introspect this GraphQL endpoint instead. Conflicts with `--service`.
`-H, --header <'Name: value'>`	A header to send with the introspection query, such as a credential. Repeat for several. Requires `--url`.
`--output <file>`	Write the schema to this file instead of stdout, overwriting it without asking, and print `Wrote the schema of <url or subdomain> to <file>`. There is no default file name. `--output`, not `-o`, which is the global format option.
`--timeout <duration>`	How long to wait for the API to answer. Defaults to `30s`. Requires `--url`.
`--service <subdomain>`	Override the active service.

Without `--url` it prints the schema of the service's live deployment. A service without one is refused: `Service '<subdomain>' is not deployed with a schema: it has no deployment yet, or it runs as plain HTTP`.

With `--url` it asks your API directly from your machine, never through GraphPilot, with the standard introspection query and the headers you give, so an API that needs a credential for introspection still answers. The schema is printed in the usual SDL style, without built-in scalars, `__` types and the specified directives. Deprecated fields, enum values, arguments and input fields are included, marked `@deprecated`. A server that cannot list deprecated arguments and input fields refuses that query; the CLI then asks again without them, warns `<url> does not read deprecated arguments and input fields, so the schema was read without them: deprecated inputs may be missing`, and the pull still succeeds. Introspection carries no applied directives: `@cacheControl`, `@surrogateKey` and `@vary` are absent, so add them to the file before you deploy it if you use them. Rules it applies:

- `https://` only, except `http://` without headers or to a loopback host (`localhost`, `127.x`, `[::1]`): headers may carry a credential, so they are not sent over plain `http://` to anything else.
- Redirects are not followed; the message names where it redirects, to pass with `--url`.
- An answer that is not JSON, or that carries no data, is refused with a message naming the status and the server's own errors, and a hint to send a credential with `--header` or to allow introspection. Answers over 32 MiB are refused.
- An answer that carries any `errors`, even beside data, is refused as incomplete: `<url> answered the introspection query with errors: <the server's messages>`. So is one that leaves the members of a type empty (its fields, input fields, enum values or union members): `The introspection answer does not list the members of type <type>, so the schema would be incomplete. Nothing was written`.
- After any refusal nothing is written, and an existing `--output` file is kept as it was.
- Messages print the URL without credentials or query string.

Nothing but the schema goes to stdout. Exit `0` on success, `1` for every refusal above, `2` for a usage error, and `3` or `4` for an authentication or API failure (without `--url` only).

gpilot config

Aliases: `c`, `cfg`. Read and convert the **service** configuration, the `gpilot.toml` a deployment is built from. The CLI's own settings on your machine are covered in environment variables and profiles.

get

```
gpilot config get [--to <format>] [--output <file>]
```

Print the config the service is currently deployed from. It comes back exactly as it was deployed, a JWT provider's `secret` included, so this needs `service:edit`, for a member and an API key alike; with only `service:view` it is refused (`FORBIDDEN`). See permissions.

Argument	Description
`--to <format>`	Render as `toml`, `json` or `yaml`. Defaults to `toml`, which is the stored document and is printed byte for byte; the others are rendered from it.
`--output <file>`	Write to this file instead of stdout.
`--service <subdomain>`	Override the active service.

`--to` is not `--format`: the global `--format` decides how command output is rendered, which is a different question from what language this document is written in.

convert

```
gpilot config convert <path> [--to <format>] [--output <file>]
```

Convert a local config file between the accepted formats, without touching the deployed one.

Argument	Description
`<path>`	The file to convert. TOML, JSON, JSON5 or YAML, picked by extension.
`--to <format>`	Format to convert to. Defaults to `toml`, which is what a deploy uploads.
`--output <file>`	Write to this file instead of stdout.

gpilot completion

Generate shell completions. Supported shells: `bash`, `zsh`, `fish`, `powershell`, `elvish`.

```
```

```
gpilot completion <shell>
```

Argument	Description
`<shell>`	Shell to generate completions for.

Examples:

```
gpilot completion zsh > ~/.zfunc/_gpilot
```

```
gpilot completion bash > /usr/local/etc/bash_completion.d/gpilot
```

Reference

REFERENCE

Configuration reference

A service is configured by one file. This page lists everything it accepts.

The file

You write it as `gpilot.toml` in the directory you deploy from. `gpilot deploy` picks it up automatically, along with `schema.graphql` beside it.

TOML is canonical, but the CLI also accepts YAML, JSON and JSON5, chosen by extension (`gpilot.yaml`, `gpilot.yml`, `gpilot.json`, `gpilot.json5`, `gpilot.tml`) and converts them to TOML on the way up. `gpilot config convert` does the same conversion locally, and `gpilot config get` prints what the service is currently deployed from.

Every setting has a default, and an empty file is valid. Configure what you need to change. A deployment without a configuration file runs on the defaults alone; see what goes into a deployment.

> **Note**
> Configuration is applied **when the service is deployed**, not per request. A change takes effect on your next `gpilot deploy`, and entries already in the cache keep the behavior they were stored with.

`[request]`

What the edge accepts, and how much GraphQL it is willing to parse. Not every limit here runs on every request:

- **On every request, a cache hit included:** `graphql_paths`, which decides whether a request is GraphQL at all and refuses a GraphQL operation sent anywhere else, and `max_body_bytes`. Authentication and the client profile checks also run before the cache.
- **Only on requests that reach GraphQL processing**, which means a cache miss or a pass: `max_query_depth`, `max_document_bytes`, `max_selections`, `introspection` and `introspection_cache_max_age`. A hit is answered without parsing the document, so these limits cost it nothing. They protect your origin, which a hit does not reach.

Key	Type	Default	Does
`request.graphql_paths`	list of strings	`["/graphql"]`, or `[]` without a schema	The paths treated as GraphQL. Several paths are allowed (for example the old and the new path during a migration); all of them serve the service's one schema, so a second API with a schema of its own is a second service. Matched the way your origin's router would: case does not matter, repeated slashes, a trailing slash, `.` and `..` segments, percent-escapes (except `%2F`) and `;parameters` are ignored, and a path **below** an entry counts as the entry (`/graphql/x` is `/graphql`, `/graphql-admin` is not), because that is how Express mounts Apollo Server and Yoga. An entry of `/` matches the root only. Anything else is forwarded as ordinary HTTP, except a request that carries a GraphQL operation there: that is refused with `404` and `NOT_A_GRAPHQL_PATH` while any GraphQL check is in force (the introspection block, `persisted_query.only_registered`, a client profile with `persisted_query = "required"`, or a depth, document size or selection limit, on the service or a profile), which by default it is. See GraphQL on other paths. Serve GraphQL only on these paths all the same: an upload or a WebSocket subscription sent elsewhere is not recognized. An empty list is taken at its word: GraphQL nowhere, and nothing is refused for being GraphQL. A deployment without a schema always runs with `[]`: setting the key there (any value, `[]` included) does not fail the deployment, the key is ignored, and the build records a `request_graphql_paths` error saying so.
`request.max_body_bytes`	integer (bytes)	`8388608` (8 MiB)	Largest request body accepted, inclusive. A larger one is refused with `413` and `PAYLOAD_TOO_LARGE` before it is parsed and before it reaches your origin. Raise or lower it with any integer. The limit cannot be removed: TOML has no null, and `gpilot` refuses a null in a YAML or JSON file. `0` is not "no limit" here: it refuses every request that carries a body.

Key	Type	Default	Does
`request.forward_unchecked_bodies`	list of `"batch"`, `"multipart"`, `"other"`	`[]`	The kinds of body on a GraphQL path that are forwarded to your origin unchecked instead of refused with `415` and `UNSUPPORTED_GRAPHQL_BODY`. `batch` is a JSON array of operations, `multipart` a `multipart/form-data` upload, `other` anything else (text, no content type, JSON that names no operation, a body that does not parse). A listed kind reaches your origin as the bytes the client sent, past persisted query enforcement, the introspection block and the document limits. A request without a body is never refused here. A compressed body (any `Content-Encoding` other than `identity`) is refused with `400` and `INVALID_GRAPHQL_REQUEST`, not forwarded, whichever kinds are listed. See bodies on a GraphQL path.
`request.max_query_depth.client_document`	integer	`10`	Deepest nesting allowed in an operation a client sent. `0` disables the limit.
`request.max_query_depth.registered_document`	integer	`10`	The same for a document resolved from the persisted query registry. `0` disables it, which is only safe with `persisted_query.preset = "trusted"`: under `apq` any client can register a document and then send its hash.
`request.max_document_bytes`	integer	`65536` (64 KiB)	Longest GraphQL document parsed, in bytes of the `query` string, inclusive. A longer one is refused with `400` and `DOCUMENT_SIZE_LIMIT_EXCEEDED` before it is parsed. Governs a document the request carried; a **registered** one is measured against `persisted_query.max_document_bytes`, which inherits this value unless you set it. An APQ registration over the registry's ceiling is not stored. `0` disables the limit.

Key	Type	Default	Does
`request.max_selections`	integer	`4000`	Most selections an operation may make, inclusive, counted with every alias and every fragment spread expanded. More is refused with `400` and `DOCUMENT_SELECTION_LIMIT_EXCEEDED` before the query plan is built. Governs a document the request carried; a **registered** one is measured against `persisted_query.max_selections`, which inherits this value unless you set it. `0` disables the limit. See how selections are counted.
`request.introspection`	`"blocked"` | `"allowed"`	`"blocked"`	Whether the edge answers `__schema` and `__type` introspection. Blocked by default so a public service does not hand out its schema shape. `__typename` is never blocked.
`request.introspection_cache_max_age`	integer (seconds)	unset	Accepted and has no effect. It used to give the response to an introspection request **GraphPilot itself sends** a cache lifetime; such a platform probe is now never cached, so it always reads what your origin answers. A configuration that sets it still deploys.

```
[request]
graphql_paths = ["/graphql", "/api/graphql"]
introspection = "allowed"
forward_unchecked_bodies = ["multipart"]   # this API takes file uploads

[request.max_query_depth]
client_document = 8
```

See limits for what each limit protects against.

`[origin]`

How the request is sent to your API. The origin **URL** is not here: it belongs to the service and is changed with `gpilot service edit`.

Key	Type	Default	Does
`origin.field_prefix`	string	`"__gp_"`	The prefix on fields selection set expansion injects, so they can be recognized and stripped again. A client document whose alias (or unaliased field name) starts with it is refused with `400` `RESERVED_FIELD_PREFIX`. Change it only if it collides with your own field names: a schema field starting with the prefix can no longer be selected without an alias. An empty or blank value means the default `__gp_`.
`origin.method_rewrite`	`"verbatim"` | `"query_to_post"` | `"post_to_query"`	`"verbatim"`	Rewrites the HTTP method on the way to your origin. See below.
`origin.connect_timeout_ms`	integer or unset	`1000`	How long to wait for TCP and TLS. Capped at `5000` on every plan.
`origin.first_byte_timeout_ms`	integer or unset	`15000`, or your plan's cap if lower	How long to wait for the first response byte. Plan-dependent; see origin timeouts.
`origin.between_bytes_timeout_ms`	integer or unset	`10000`	How long to wait between two bytes of a response already in flight. Capped at `60000`.
`origin.request_id_header`	string or unset	unset	A header name your origin receives the GraphPilot request id under, so your own logs can be joined to GraphPilot's. You choose the name, typically one your origin already logs, such as `x-request-id`. A name in the `gp-` namespace **refuses the deployment**, because that namespace never reaches your origin; so does a name that is not a valid HTTP header name. Do not reuse the name of your `telemetry.correlation_header`: GraphPilot's id replaces the client's value under that name. See request ids.
`origin.forward_client_ip`	boolean	`false`	Whether your origin receives the client's IP address, in the `gp-client-ip` request header. Off by default: an end user's IP address is personal data, so passing it on is your decision. On, the value is the address the edge saw for the connection (on a shielded request, the one the first edge location saw, never a GraphPilot address), and no header is sent when the address is unknown. A client cannot set it: every `gp-` header a client sends is dropped on arrival. Independent of `telemetry.client_ip`, which decides what GraphPilot stores, not what your origin receives. See request headers your origin receives.

Key	Type	Default	Does
`origin.forwarded_headers`	`"strip"` | `"pass"`	`"strip"`	What your origin receives of the forwarding headers a client sent: every header whose name starts with `X-Forwarded-`, plus `Forwarded`, `X-Real-IP`, `X-Client-IP`, `Client-IP`, `True-Client-IP`, `CF-Connecting-IP`, `Fastly-Client-IP`, `X-Original-URL`, `X-Rewrite-URL` and `X-Original-Host`. `"strip"` removes them right before the request is sent to your origin; GraphPilot's own policies and vary rules still read them, and so does the capture of the client's request headers in telemetry. `"pass"` forwards them unchanged, and then trusting them and keying the cache on them is up to you. The client's address your origin can trust is `gp-client-ip` (`origin.forward_client_ip`). An `[origin.headers]` `only` list that keeps one of these headers, or the pattern `x-forwarded-*`, while they are stripped **refuses the deployment**. See forwarding headers.
`origin.shield`	string or unset	unset	The shield location that cache misses go through before they reach your origin, named `<region>-<city>` (see the location table). Off by default, and available on paid plans. On Glider the deployment still succeeds, with a `plan` warning in its findings: the shield is configured but inactive, and it turns on by itself after an upgrade, without a redeploy. A value that is not a location the network knows refuses the deployment and suggests the closest match.

A timeout above your plan's ceiling or below `100` ms, or a retry policy whose worst case does not fit 120 seconds, refuses the deployment. See a value out of bounds is refused and timeouts and retries.

The `QUERY` method (`draft-ietf-httpbis-safe-method-w-body`) is a safe, idempotent method that carries a body: a read with a body, unlike `GET`. GraphPilot accepts it from clients and caches it exactly like the same operation over `POST`. `method_rewrite` lets the two sides adopt it independently:

Value	Effect
`verbatim`	Forward the method unchanged.
`query_to_post`	Downgrade `QUERY` to `POST`, so clients can adopt `QUERY` while your origin stays on `POST`.
`post_to_query`	Upgrade `POST` to `QUERY`, so your origin can adopt `QUERY` while clients keep sending `POST`. Only `query` operations are upgraded; mutations and subscriptions stay `POST`, because `QUERY` is a safe method and must not carry them.

`[origin.retry]`

Off unless you opt in. Full detail on origin retries.

Key	Type	Default	Does
`origin.retry.enabled`	boolean	`false`	Whether failed origin requests are retried at all.
`origin.retry.max_retries`	integer	`1`	Additional attempts. `1` means the origin sees the request at most twice.
`origin.retry.status_codes`	list of integers	`[502, 503, 504]`	Origin statuses that trigger a retry.
`origin.retry.on_network_error`	boolean	`true`	Whether a transport failure is retried.
`origin.retry.retry_mutations`	boolean	`false`	Whether GraphQL mutations are retried. Turn on only if your origin never returns a gateway status after a write already executed, or your mutations are idempotent. Does not apply to plain HTTP requests.
`origin.retry.retry_unsafe_methods`	boolean	`false`	Whether plain HTTP requests with a method other than `GET`, `HEAD` or `OPTIONS` (`POST`, `PUT`, `PATCH`, `DELETE` and any other) are retried. Turn on only for an origin that makes its writes idempotent. Does not apply to GraphQL requests.
`origin.retry.backoff`	`"none"` | `"fixed"` | `"exponential"`	`"fixed"`	How the delay is computed.
`origin.retry.delay_ms`	integer	`1000`	Base delay.
`origin.retry.max_delay_ms`	integer	`2000`	Ceiling for any single delay, for both `fixed` and `exponential`.

A plain HTTP request is retried only when its method is `GET`, `HEAD` or `OPTIONS`; any other method is sent once unless `retry_unsafe_methods` is `true`. See which requests are repeated.

`[origin.headers]` and `[origin.cookies]`

Names dropped from the request on its way to your origin, after the cache decision has already been made from the request as the client sent it. Full detail on header and cookie filters.

Key	Type	Default	Does
`origin.headers.hide`	list of strings	unset	Header names removed before the request reaches your origin.

Key	Type	Default	Does
`origin.headers.only`	list of strings	unset	Header names that survive; everything else is removed. Mutually exclusive with `hide`.
`origin.cookies.hide`	list of strings	unset	Cookie names removed from the `Cookie` header.
`origin.cookies.only`	list of strings	unset	Cookie names that survive in the `Cookie` header; everything else is removed. Mutually exclusive with `hide`.

```
[origin.headers]
hide = ["x-internal-debug"]

[origin.cookies]
hide = ["internal_ab_test"]
```

Setting both `hide` and `only` on the same section, or naming a protected name (`host`, `content-type`, `content-encoding`, `content-length`, `transfer-encoding`, or anything starting with `gp-`), **refuses the deployment**. See protected names. So does a conditional `rules` list or `when`, which only `[response.headers]` and `[response.cookies]` accept. A `rules` list is refused with the reason, and the message closes with `[origin.headers] supports only the plain `hide` and `only` lists` (or `[origin.cookies]`); a bare `when` is refused as an unknown key.

> ⚠ **Caution**
>
> `origin.headers` matches case-insensitively, HTTP's own rule for header names. `origin.cookies` matches case-**sensitively**, RFC 6265's rule: `gaesa` and `GAESA` are different cookies, and a name in the wrong case simply never matches, deploy included. See matching.

[cache]

Key	Type	Default	Does
`cache.ignore_origin_header_without_freshness`	boolean	`true`	Lets an origin `Cache-Control` that constrains neither storage nor freshness (a bare `private`, `public` or `no-transform`) yield to the schema's `@cacheControl` policy instead of deciding the response itself. A `private` it stated is still carried over, so the entry stays partitioned. Set `false` to restore the older precedence, where the header's mere presence decided. See precedence.

[cache.request_directives]

What the `Cache-Control` directives a client sends on its own request do. Each value is `"honor"` or `"ignore"`; any other value, or an unknown key in this section, is refused when the file is read. Full detail on the client's own

`Cache-Control`.

Key	Type	Default	Does
`cache.request_directives.no_store`	`"honor"` | `"ignore"`	`"honor"`	Honored: a request carrying `no-store` is still answered from the cache when it can be, but its response is not written, reported as `CACHE_SKIPPED_REQUEST_NO_STORE`.
`cache.request_directives.no_cache`	`"honor"` | `"ignore"`	`"ignore"`	Honored: a request carrying `no-cache` or `max-age=0` skips the lookup, goes to your origin and is not written back, reported as `PASS` with `CACHE_PASS_REQUEST_NO_CACHE`. Ignored by default because any caller could then send every request past the cache to your origin.

```
[cache.request_directives]
no_cache = "honor"   # only for callers you trust
```

[cache.http_key]

What the cache key of a plain HTTP request is made of: any request that is not a GraphQL operation. By default it is the method, the path and the query string, plus a SHA-256 of the body when there is one, so `GET /products/42?color=blue` is stored under `GET:/products/42?color=blue` whichever hostname it arrived on. The same key is the `ck:` key you purge one response by, and the `cache.key` in request telemetry. GraphQL keys are not affected. An unknown value, or an unknown key in this section, is refused when the file is read. See the cache key.

Key	Type	Default	Does
`cache.http_key.host`	`"exclude"` | `"include"`	`"exclude"`	Whether the scheme and hostname are part of the key. Excluded, one path is one entry across the service hostname and every custom domain, and you purge it as `ck:GET:/products/42`. Included, the key is the full URL, `GET:https://catalog-api.k7m2qp.graphpilot.cloud/products/42`, and each hostname keeps its own entry. Include it only when your origin answers the same path differently per hostname.

```
[cache.http_key]
host = "include"   # only when your origin serves different content per hostname
```

Changing the setting changes every plain HTTP key, so the entries stored under the old ones are no longer found: each misses once, is filled again under the new key, and the old entry expires on its own lifetime.

[cache.policy]

How the cache is partitioned. Every value except `downstream` is a policy expression; the concepts are on identifying callers.

Key	Type	Default	Does
`cache.policy.public`	expression	unset	The value a shared entry is partitioned by, for example one of your customers.
`cache.policy.private`	expression	unset	The value a private entry is partitioned by. Without it, a private response is **not cached**. Write it so it identifies one caller, not their group.
`cache.policy.bypass`	expression	unset	When it matches, the request skips the cache entirely and its response is not stored. Evaluated as a truth, not as a value.
`cache.policy.downstream`	`"off"` | `"if_cacheable"` | `"always"`	`"off"`	Whether to emit a `Cache-Control` header on the response sent to the client. See downstream header.
`cache.policy.store_error_statuses`	list of integers	`[]`	The `4xx` and `5xx` statuses that may be stored, under the ordinary rules, when your origin marks them cacheable. Any other error status is not stored, whatever its `Cache-Control` says, and reports `PASS` with `CACHE_SKIPPED_ERROR_STATUS`. A value outside `400` to `599` refuses the deployment. See an error status is not stored unless you opt in.
`cache.policy.uncacheable_methods`	list of strings	`["OPTIONS"]`	Request methods that skip the cache entirely: no lookup, no key, no write back. Reported as `CACHE_PASS_UNCACHEABLE_METHOD`. Empty the list to let CORS preflights be answered at the edge; read the hazard first, since a preflight is only safe to cache because GraphPilot keys it on the caller's `Origin` and on what it asked for. Names are capitals, and one that is not a valid HTTP method is refused when the file is read.

`[[cache.policy.vary]]`

Repeatable. Each entry derives a value and writes it onto the request as a header. **Every response** is partitioned on it, plain HTTP included, whether or not your origin's `Vary` names it. An entry marked `scope_only` partitions a response your schema decided only when its scope names it in `@vary`; a response whose lifetime your origin's `Cache-Control` decided is still partitioned on it. A request without a credential that the expression does not resolve for is stored as the variant without the header. A request with a credential that it does not resolve for, or resolves empty for, is not stored (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`). Each entry costs hit rate, so configure only those whose value shapes a response. See vary rules.

Key	Type	Default	Does
`name`	string	required	The header the derived value is written to. Configuring an entry claims that name: a client sending it has its value removed, logged as `INGRESS_CLAIMED_HEADER_DROPPED`. Should not start with `gp-`: such a name is not refused and still partitions the cache, but the namespace is stripped before your origin, so `pass_to_origin` has no effect. A name that is not a legal header name is skipped and reported in the deployment's diagnostics.
`value`	expression	required	Computes the value.
`pass_to_origin`	`"never"` | `"anonymized"` | `"plain"`	`"never"`	What your origin receives. `anonymized` sends a keyed digest, so the origin can tell two contexts apart without learning what distinguishes them. Under `anonymized`, request telemetry records that same digest as the segment value. Under `never` and `plain`, `telemetry.vary_values` decides: a keyed pseudonym by default, the value in clear only if you choose it. See what Insights records.
`scope_only`	boolean	`false`	Leaves the entry to your schema's named scopes: a response your schema decided is partitioned on it only when its scope names it. Only accepted with `pass_to_origin = "never"`; any other combination **refuses the deployment**. See vary rules.

```
[cache.policy]
private    = "auth:idp.sub"
downstream = "if_cacheable"

[[cache.policy.vary]]
name           = "x-segment"
value          = "auth:idp.plan"
pass_to_origin = "anonymized"
```

`[cache.headers]` and `[cache.cookies]`

What the cache does with response headers and cookies when it writes an entry. Full detail on header and cookie filters.

Key	Type	Default	Does
`cache.headers.exclude`	list of strings	unset	Response headers not written to the entry. The client that caused the miss still receives them; a later hit does not. Refused at deploy time if it names a protected header (`content-type`, `content-encoding`, `cache-control`, `vary`, `surrogate-key`).

Key	Type	Default	Does
`cache.headers.allow`	list of strings	unset	Headers that no longer refuse storing the response, out of the fixed list GraphPilot otherwise refuses on (`WWW-Authenticate` , `Proxy-Authenticate` , `Authentication-Info` , `Proxy-Authentication-Info` , `Clear-Site-Data` , `Set-Login`). An allowed header is also never written to the entry. Refused at deploy time if it names anything other than one of those six, and refused too for `Set-Cookie` or `Set-Cookie2` , which `[cache.cookies].allow` governs instead; use `cache.headers.exclude` to drop an ordinary header from the cache entry.
`cache.cookies.allow`	list of strings	see the default list	Cookie names the service says carry no session: they do not make a request credentialed, do not refuse a response for carrying `Set-Cookie` , and are never written to the entry. **Setting this key replaces the default list rather than adding to it.**

```
[cache.headers]
exclude = ["x-request-id"]

[cache.cookies]
allow = ["GAESA", "GCLB", "my_custom_lb_cookie"]
```

Neither section accepts a conditional `rules` list or `when` . Writing one **refuses the deployment**, because what is stored under a cache key cannot depend on a condition the key does not include. The message says so and names the plain lists the section does read, closing with `[cache.headers] supports only the plain `exclude` and `allow` lists` , or `[cache.cookies] supports only the plain `allow` list` .

> **Caution**
>
> `cache.cookies.allow` defaults to a non-empty list of infrastructure cookies (App Engine's `GAESA` among them). Writing `[cache.cookies]` in your config discards that default entirely, so if you still need one of the shipped names, restate it alongside the one you are adding.

> **Danger**
>
> `cache.headers.exclude` and `cache.headers.allow` match case-insensitively, HTTP's own rule. `cache.cookies.allow` matches case-**sensitively**, RFC 6265's rule: `gaesa` and `GAESA` are different cookies. Writing the wrong case is not refused at deploy time, it just never matches, so the cookie stays treated as carrying a session. See matching.

`[response]` reporting headers

Whether a client sees the headers GraphPilot sets to report what the cache did. Changing one changes only what leaves: caching, purging and telemetry are unaffected. Every header is listed on headers; full detail on switching the headers on and off.

Key	Type	Default	Does
`response.cache_headers`	boolean	`true`	Sends the cache status family: `gp-cache`, `gp-cache-reason`, `gp-cache-hits`, `gp-cache-age`, `gp-cache-max-age` and `gp-cache-remaining`. Set `false` to withhold all six.
`response.surrogate_key_header`	boolean	`false`	Sends `gp-surrogate-key`, the surrogate keys the response is tagged with, plus `gp-surrogate-key-truncated` when keys were left out. The keys are stored and purging by key works either way.
`response.surrogate_key_header_max_bytes`	integer (bytes)	`8192`	The most the value of `gp-surrogate-key` may carry, whole keys only, cache and root field keys first. Applies to that one header, not to all headers together. Keys that do not fit are counted in `gp-surrogate-key-truncated`; if none fits, `gp-surrogate-key` is left off. From `1` to `128000` (Fastly's per-header limit); anything else refuses the deployment. Only read when `surrogate_key_header` is `true`. See size limit.
`response.request_id_header`	boolean	`false`	Sends `gp-request-id`, the id GraphPilot recorded this request under, shown as the **Request ID** column in Insights and as **Entry ID** in the request's **Overview** tab. The id is minted and the telemetry row carries it either way; this decides only whether the caller sees it. See request ids.

```
[response]
cache_headers = false
surrogate_key_header = true
surrogate_key_header_max_bytes = 32000
request_id_header = true
```

`cache_headers` and `surrogate_key_header` are the only way to withhold a `gp-` header: `[response.headers]` refuses the namespace at deploy time.

`[response.cors]`

CORS headers on the responses GraphPilot produces itself: its rejections and its persisted query answers, which never reach your origin and so carry none of the CORS headers your origin would send. Without this section, nothing changes. Full detail on edge rejections.

Key	Type	Default	Does
`response.cors.allow_origins`	list of strings	`[]`	The origins whose requests get CORS headers on a response GraphPilot produced, written the way a browser sends `Origin`: `scheme://host`, with a port where it is not the scheme's default, over `http` or `https`, with no path and no trailing slash. Matched case-insensitively. `"*"` on its own allows every origin.
`response.cors.allow_credentials`	boolean	`false`	Adds `Access-Control-Allow-Credentials: true`, for a browser app that sends cookies or uses `credentials: "include"`.

`"*"` combined with another origin or with `allow_credentials = true`, a partial wildcard such as `https://*.example.com`, an origin with a path, and `null` **refuse the deployment**. Responses from your origin, cache hits included, are never changed.

`[response.headers]` and `[response.cookies]`

Names dropped from the response on its way to your client, applied on delivery rather than when an entry is written, so it also governs a cache **hit** served from an entry stored before this configuration existed. Full detail on header and cookie filters.

Key	Type	Default	Does
`response.headers.hide`	list of strings	unset	Header names removed from the response before it reaches your client.
`response.headers.only`	list of strings	unset	Header names that survive; everything else is removed. Mutually exclusive with `hide`.
`response.cookies.hide`	list of strings	unset	Cookie names removed from `Set-Cookie` and `Set-Cookie2`.
`response.cookies.only`	list of strings	unset	Cookie names that survive in `Set-Cookie` and `Set-Cookie2`; everything else is removed. Mutually exclusive with `hide`.
`response.headers.rules`	array of tables	unset	Lists that apply only to the requests their `when` matches. See below.
`response.cookies.rules`	array of tables	unset	The same for cookies.

```
[response.headers]
hide = ["x-origin-latency-ms"]

[response.cookies]
hide = ["__internal_session"]
```

`[[response.headers.rules]]` and `[[response.cookies.rules]]`

Repeatable. Applied after the plain list and after one another, so a name survives only if every list that applies keeps it. Full detail on conditional rules.

Key	Type	Default	Does
`hide`	list of strings	unset	Names this rule drops.
`only`	list of strings	unset	Names this rule keeps; everything else is dropped. A rule must set exactly one of `hide` and `only`.
`when`	expression	unset	Whether the rule applies to this request. Evaluated as a **truth**, with `auth:` claims available. Absent means every request.

```
[[response.headers.rules]]
hide = ["x-debug-trace"]
when = "!auth:idp.role || auth:idp.role ≠ 'admin'"
```

`rules` and `when` are accepted **only** here. `[origin.*]` and `[cache.*]` refuse them at deploy time, because their filtering reaches the cache under a key that does not include the condition.

Setting both `hide` and `only` on the same section, or naming a protected name (`content-type`, `content-encoding`, `cache-control`, `vary`, `surrogate-key`, or anything starting with `gp-`), **refuses the deployment**. See protected names.

> ⚠ **Caution**
>
> `response.headers` matches case-insensitively, HTTP's own rule. `response.cookies` matches case-**sensitively**, RFC 6265's rule: a name in the wrong case never matches, and does not fail the deployment. See matching.

`[auth.providers.<name>]`

Repeatable, one block per provider. The block name is what a policy expression refers to (`auth:<name>.<claim>`). Full detail on authentication.

JWT providers (`type = "jwt"`):

Key	Type	Default	Does
`type`	`"jwt"`	required	

Key	Type	Default	Does
`allowed_algorithms`	list of strings	**required**	The algorithms this provider accepts. A security control, not a formality: it is what stops an algorithm downgrade. Accepted spellings: `HS256`, `HS384`, `HS512`, `RS256`, `RS384`, `RS512`, `PS256`, `PS384`, `PS512`, `ES256`, `ES384`, `ES256K`, `EdDSA` and `BLAKE2B`.
`jwks_url`	string	unset	Where the issuer publishes its keys. The key set is read at deploy time and delivered to the edge. When a token names a key id the delivered set lacks, the edge location fetches `jwks_url` (at most once a minute, or 10 seconds after a failed fetch) and shares the result, so a rotation is usually picked up without a redeploy. Only if that fetch fails is the token rejected with `AUTH_UNKNOWN_KEY_ID`. See authentication.
`public_key`	string (PEM)	unset	Public key for an asymmetric algorithm without a JWKS.
`default_key_id`	string	unset	Key to use when a token carries no `kid`.
`secret`	string	unset	Shared secret, for symmetric algorithms. Stored and deployed to the edge as written, as part of the configuration document: anyone with `service:edit`, a member or an API key, can read it back with `gpilot config get`. Keep a `gpilot.toml` that holds a secret out of version control, or use an asymmetric algorithm with `jwks_url` or `public_key`, which hold only public material.
`header`	string	`"Authorization"`	Header carrying the token. A `Bearer` prefix is stripped when present.
`allowed_issuers`	list of strings	**required** with `jwks_url`, `public_key` or an asymmetric algorithm, otherwise unset	Accepted `iss` values. A provider that verifies with a public key (it sets `jwks_url` or `public_key`, or lists an `RS*`, `PS*`, `ES*`, `ES256K` or `EdDSA` algorithm) is refused without it, or with an empty list, and the deployment fails. On a shared-secret provider, unset accepts any issuer and the deployment warns (`auth_provider`). Set it.
`allowed_audiences`	list of strings	unset	Accepted `aud` values. Unset accepts any audience, and the deployment warns (`auth_provider`). Set it.
`required_nonce`	string	unset	Required `nonce`.
`required_key_id`	string	unset	Required `kid`.

Key	Type	Default	Does
`time_tolerance`	duration	unset	Clock skew allowance. See the note below on how to spell a duration.
`max_validity`	duration	unset	Reject tokens issued longer ago than this, whatever their expiry.
`max_token_length`	integer	unset	
`max_header_length`	integer	unset	
`accept_future`	boolean	`false`	Whether a token whose `nbf` is in the future is accepted.
`allow_missing_exp`	boolean	`false`	Whether a token without an `exp` claim is accepted. By default it is rejected with `AUTH_INVALID_TOKEN`. Turn it on only for an issuer that mints tokens without `exp`, and set `max_validity` with it. A token that states an expiry is held to it either way.

API key providers (`type = "api_key"`):

Key	Type	Default	Does
`type`	`"api_key"`	required	
`prefixes`	list of strings	**required, non-empty**	The prefixes marking a value as this provider's credential. An empty list, or one containing an empty string, is refused.
`header`	string	`"Authorization"`	Header carrying the key.

`[auth.rejection]`

What happens to a credential that cannot be used, per reason. Letting a request continue is not the same as trusting it: the provider contributes no claims, so a `private` policy naming it yields no discriminator and the response is not cached.

Key	Type	Default	Does
`auth.rejection.invalid_token`	`"reject"` | `"continue"`	`"reject"`	The signature did not verify, the token expired, or a required claim did not match.
`auth.rejection.unknown_key_id`	`"reject"` | `"continue"`	`"reject"`	The token names a key id GraphPilot does not have.
`auth.rejection.key_material_unavailable`	`"reject"` | `"continue"`	`"reject"`	No key material at all for this provider.

```
[auth.providers.idp]
```

```
type                = "jwt"
allowed_algorithms = ["RS256"]
jwks_url            = "https://idp.example.com/.well-known/jwks.json"
allowed_issuers     = ["https://idp.example.com/"]

[auth.rejection]
invalid_token            = "reject"
unknown_key_id           = "continue"
key_material_unavailable = "continue"
```

[persisted_query]

Four independent switches, plus a `preset` that names the common combinations. See choosing a setup for how to choose.

Key	Type	Default	Does
`persisted_query.preset`	`"off"` | `"apq"` | `"trusted_migrating"` | `"trusted"`	`"off"`	Sets the four switches below to a named combination (see the table after this one). A switch you write beside it overrides the preset's value for that switch. `mode` is accepted as an alias of `preset`.
`persisted_query.enabled`	boolean	`false`	Whether a hash is resolved against the registry at all. Off, a request carrying a hash is answered with `PERSISTED_QUERY_NOT_SUPPORTED`.
`persisted_query.apq`	boolean	`false`	Whether a client may register an operation by sending it with its hash (Automatic Persisted Queries). Anyone who can call the service can then write to its registry. Off, a push is the only way in.
`persisted_query.only_registered`	boolean	`false`	Makes the registry an allowlist: an operation it does not hold is refused with `403` and `FORBIDDEN`, including a full query sent as text, and logged as `PERSISTED_QUERY_REFUSED`. Fails closed when the registry cannot be read, again as `FORBIDDEN`, logged as `PERSISTED_QUERY_REGISTRY_UNAVAILABLE`. Checked below the cache, so on misses and passes: a response cached before you switched it on is served until it expires or is purged.
`persisted_query.audit_unregistered`	boolean	`false`	Reports each operation the registry does not hold as `PERSISTED_QUERY_UNTRUSTED_OPERATION`, and lets it run. The list of what your manifest is still missing.

Key	Type	Default	Does
`persisted_query.document_ttl_secs`	integer	`604800` (7 days)	How long a resolved document stays in the edge-location cache. Entries are content-addressed, so a hash points at one document forever or at nothing.
`persisted_query.not_found_ttl_secs`	integer	`60`	How long a “this hash resolves to nothing” answer is remembered, so random hashes cannot force a lookup per request.
`persisted_query.max_document_bytes`	integer (bytes)	`request.max_document_bytes`	The document ceiling for a **registered** document: one resolved from the registry for a hash, and one a client registers over APQ. Silence inherits the `[request]` value, so setting this is how a registered document gets a different ceiling from an ad-hoc one. `0` means no limit. Refused with `400` and `DOCUMENT_SIZE_LIMIT_EXCEEDED`.
`persisted_query.max_selections`	integer	`request.max_selections`	The selection ceiling for a registered document, on the same terms. `0` means no limit. Refused with `400` and `DOCUMENT_SELECTION_LIMIT_EXCEEDED`.

What each preset sets:

Preset	`enabled`	`apq`	`only_registered`	`audit_unregistered`
`off`	`false`	`false`	`false`	`false`
`apq`	`true`	`true`	`false`	`false`
`trusted_migrating`	`true`	`true`	`false`	`true`
`trusted`	`true`	`false`	`true`	`false`

Two combinations are refused **when the file is read**, naming both keys:

Refused	Why
`only_registered = true` with `apq = true`	An allowlist any client can add to is not an allowlist. Set `apq = false` to enforce it.
`apq`, `only_registered` or `audit_unregistered` set to `true` while `enabled` is `false`	A service that does not resolve hashes has nothing to register, refuse or audit, so the switches would read as configured and do nothing.

```
[persisted_query]
preset = "trusted_migrating"
apq    = false               # collect the audit list without letting clients register
```

[client]

Which clients this service serves, and what it does with a request from one it does not recognize. The rules, including why a profile matched on a header is not a security boundary, are in telling your clients apart.

Key	Type	Default	Does
`client.unknown`	`"allow"` | `"reject"`	`"allow"`	What happens to a request matching no profile. `allow` serves it with no client recorded against it. `reject` refuses it with `403` and `UNKNOWN_CLIENT`, above the cache, before your origin is reached. A CORS preflight (an `OPTIONS` with `Access-Control-Request-Method` and no body) is never refused by `reject`, `min_version`, a profile's limits or `persisted_query = "required"`; an `OPTIONS` with a body is checked as usual.

[[client.profile]]

An **ordered** array. The first profile whose `when` is true wins. Every limit a profile sets can only tighten the service-wide value of the same name; `cache = "off"` and `max_age` relax.

Key	Type	Default	Does
`name`	string	required	The canonical name, recorded on every request this profile matches. May not contain `@` or be empty.
`when`	expression	required	Whether this request is that client. Evaluated as a **truth**, so a bare `header:x-app` means "present and not empty".
`version`	expression	unset	Where to read the client's version. Evaluated as a **value**. Absent means the client is recognized with no version recorded.
`min_version`	string	unset	The oldest release of this client the service serves. Anything below it, or a client reporting no readable version, is refused with `403` and `CLIENT_VERSION_UNSUPPORTED`. A value that is not a valid version refuses nothing and logs `CLIENT_MIN_VERSION_UNREADABLE`. Without `version`, refuses the deployment.
`max_body`	integer (bytes)	unset	A second ceiling under `request.max_body_bytes`. A larger body is refused with `413` and `PAYLOAD_TOO_LARGE`, the same code the service-wide ceiling returns; your logs record `CLIENT_BODY_TOO_LARGE` and name the client.
`max_query_depth`	integer	unset	How deep an operation from this client may nest. Refused with `400` and `QUERY_DEPTH_LIMIT_EXCEEDED`. `0` refuses the deployment; leave the key out for no ceiling.
`max_document_bytes`	integer (bytes)	unset	A second ceiling under `request.max_document_bytes`. Refused with `400` and `DOCUMENT_SIZE_LIMIT_EXCEEDED`. `0` refuses the deployment.

Key	Type	Default	Does
`max_selections`	integer	unset	A second ceiling under `request.max_selections`. Refused with `400` and `DOCUMENT_SELECTION_LIMIT_EXCEEDED`. `0` refuses the deployment.
`persisted_query`	`"any"` | `"required"`	`"any"`	`required` refuses any operation from this client that does not arrive as a **hash and nothing else**, with `403` and `PERSISTED_QUERY_REQUIRED`. A payload carrying a document is refused whatever hash travels beside it, so the APQ registration handshake is refused too. Does not apply to operations sent over a subscription socket. Ignored, logged as `CLIENT_PERSISTED_QUERY_UNENFORCEABLE`, while `persisted_query.enabled` is `false`.
`cache`	`"on"` | `"off"`	`"on"`	`off` takes this client off the shared cache: nothing is looked up, nothing is stored, every request reaches your origin.
`max_age`	integer (seconds)	unset	How old a cached entry may be before **this client** stops accepting it. Over the cap, its request goes to the origin and reports `PASS` with `CACHE_SKIPPED_ENTRY_NOT_REPLACED`; the entry is unchanged for other clients. Together with `cache = "off"`, refuses the deployment.

```toml
[client]
unknown = "allow"

[[client.profile]]
name    = "web-app"
when    = "header:gp-client-name == 'web-app'"
version = "header:gp-client-version"
min_version = "2.1.0"

[[client.profile]]
name    = "internal-bi"
when    = "auth:idp.role == 'analyst'"
version = "auth:idp.client_version"
cache   = "off"                    # fresh data every time, straight from the origin
max_query_depth = 6

[[client.profile]]
name    = "gp-cli"
when    = "starts_with(header:user-agent, 'gp-cli/')"
version = "split(header:user-agent, '/', 1)"
```

`[telemetry]`

What is recorded about a request.

Key	Type	Default	Does
`telemetry.max_query_bytes`	integer	`32768` (32 KiB)	How much of the client's query document is retained, on a request the edge answered without calling the origin itself (a cache hit, a request served through the shield). The document sent to the origin is not cut by this setting, only as far as the log line size limit requires. `0` removes the bound. Query **hashes** are always computed over the full text, so truncation never affects a cache key or a purge.
`telemetry.correlation_header`	string	`"GP-Correlation-ID"`	The request header a client sends its own correlation ID in. Matched case-insensitively. A name that is not a valid HTTP header name is refused at deploy. The header is forwarded to your origin unchanged, the default name included, unless `[origin.headers]` removes it. See correlating logs.
`telemetry.record`	expression or boolean	unset (record everything)	Which requests are recorded in Insights: a policy expression read as a truth. `"false"` records nothing. See recording only some requests.
`telemetry.client_ip`	`"full"`, `"truncated"` or `"none"`	`"full"`	How much of the end user's IP address, and of the geo and network data derived from it, the request's telemetry keeps. Unless `"full"`, no `[telemetry.headers]` rule records `gp-client-ip` or a header that carries the client's address either. Any other value refuses the deployment. See client IP address.
`telemetry.vary_values`	`"pseudonymized"` or `"clear"`	`"pseudonymized"`	The form request telemetry stores the value of a `[[cache.policy.vary]]` entry in, for entries whose `pass_to_origin` is `never` or `plain`: a keyed pseudonym, or the value in clear. Any other value refuses the deployment. See vary values.

Recording only some requests

Every request a service serves is recorded in Insights, on every plan, unless `telemetry.record` says otherwise. Set it to a predicate and only the requests it matches are recorded:

```
[telemetry]
# Nothing at all.
record = "false"
```

```
[telemetry]
# Everything but the cluster's health probes.
record = "!starts_with(header:user-agent, 'kube-probe/')"
```

- **Unset records everything.** `"false"` records nothing, and a bare TOML `false` means the same (`record = true` is the same as leaving the key out). Any other value that is not a string, and an expression that does not parse, fails the deployment.
- **A request the expression cannot decide is recorded.** The second example has no answer for a caller that sent no User-Agent, and that caller is kept. To leave them out as well, write `header:user-agent && !starts_with(header:user-agent, 'kube-probe/')`. See a request the expression cannot decide.
- **A request that is not recorded is still served and still counted.** It is billed and counts towards your allowance exactly like any other. Only what Insights stores changes: the request, its log lines and any purge it triggers stay out of the request log, the charts, the purge history and `gpilot requests`.
- **It can only leave things out.** Recording a request does not keep it while the account runs in lean mode.
- **An expression over `auth:` is decided on the verified claims.** A request refused before authentication completes is decided without them.
- **The portal says so.** With `"false"`, every Insights tab says the service records no requests and why, in place of empty lists; with a predicate, a note names the expression. The deployment also reports `record = "false"` as an info finding.

There is no `path` or `method` source, so a health check is told apart by what it sends, typically its User-Agent. The setting needs a runtime that ships it: an older runtime does not read the key, records everything, and the deployment's configuration report lists the key as unknown.

Client IP address

By default every recorded request keeps the client's IP address as the edge saw it, plus the geo and network data looked up from it. `telemetry.client_ip` reduces that per service, for data-minimisation requirements, without turning telemetry off:

```
[telemetry]
client_ip = "truncated"
```

- `"full"` (the default) keeps the address and everything derived from it. Leaving the key out changes nothing.
- `"truncated"` keeps the network part of the address: an IPv4 address with its last octet zeroed (`203.0.113.77` becomes `203.0.113.0`), an IPv6 address cut to its first 48 bits (`2001:db8:85a3:8d3::7` becomes `2001:db8:85a3::`). An IPv4 address in IPv6 notation (`::ffff:203.0.113.77`) is truncated as the IPv4 address it is. An address that cannot be read is not stored at all.
- `"none"` stores no address.

Field	`full`	`truncated`	`none`
Client IP	as seen	network part (/24 or /48)	not stored
Country, continent	kept	kept	kept
Region, city	kept	kept	not stored
Latitude, longitude, postal code	kept	not stored	not stored
UTC offset	kept	kept	not stored
ISP, ASN, connection speed and type, proxy type and description	kept	kept	not stored
Edge location and node, TLS and HTTP version	kept	kept	kept

The reduction applies to the telemetry row. Log lines carry no client IP, country or city at all, whatever the setting. The geo lookup itself still runs on the full address at the edge; only what is stored is reduced, so the country of a `"none"` service is as accurate as before.

- **Counting is unaffected.** Every request is still recorded, billed and counted towards your allowance exactly as before.
- **Your origin is unaffected.** `origin.forward client ip` decides whether your origin receives the full address in `gp-client-ip`, independently of this key.
- **Filtering by client IP is exact.** On a `"truncated"` service, filter by the truncated address (in Insights or with `gpilot requests --ip=<address>`). **Only my requests** and a bare `gpilot requests --ip` use your full address, so they match nothing on a `"truncated"` or `"none"` service.
- **In Insights**, a truncated address is shown as stored (an IPv4 one in IPv6 notation, for example `::ffff:203.0.113.0`), and a request without one shows **Not stored**. With `"none"` the UTC offset reads `0`.
- **It applies from the next deployment** on a runtime that ships the key. Requests recorded before keep what was stored then.

Vary values

Every request whose cache entry a `[[cache.policy.vary]]` entry partitioned records the value that entry derived, shown in Insights as the request's segment. `telemetry.vary_values` decides the form it is stored in, per service, for entries whose `pass_to_origin` is `never` (the default) or `plain`:

```
[telemetry]
vary_values = "pseudonymized"
```

- `"pseudonymized"` (the default) stores a keyed pseudonym, `hmac-sha256:` followed by 16 hexadecimal digits. It is made with a secret key of the service that neither you nor your origin holds, so the stored value cannot be turned back into the one it stands for. Equal values give equal pseudonyms within the service, so requests of one segment still group together and **Requests in this bucket** still finds them. Two

services give the same value two different pseudonyms. A pseudonym is never replaced by the value itself: if it cannot be derived, nothing is recorded.

- `"clear"` stores the value as derived. This is an explicit choice, and responsibility for it lies with you: a rule whose value is a session cookie, an `Authorization` header, an email address or any other personal data then stores that data in your request telemetry, where Insights shows it. The `[telemetry.headers]` sanitize rules do not apply to it.

Not affected by this key:

- **Entries with** `pass_to_origin = "anonymized"` store the `v1=` digest your origin receives, and `cache.policy.private` and `public` always store a `v1=` digest.
- **The cache and your origin.** What the cache partitions on and what `pass_to_origin` sends your origin stay the same; only the stored copy changes.
- **Counting.** Every request is still recorded, billed and counted as before.

A value in clear never equals its pseudonym, so after a change of this key requests of one segment form two groups, before and after the deployment. Requests recorded before keep what was stored then, until your plan's retention deletes them.

`[telemetry.headers]`

Four independent rule lists. Each is an ordered list of rules; the first match wins, and a header no rule matches is **not** recorded.

List	Covers
`telemetry.headers.request`	The request as the client sent it.
`telemetry.headers.origin_request`	The request as it leaves for your origin, after vary scrubbing, your `[origin.headers]` filter, the forwarding header strip and signing: what your origin actually received.
`telemetry.headers.response`	The response as your origin sent it. The name is older than the fourth list; read it as "origin response".
`telemetry.headers.client_response`	The response as the client receives it, after `[response.headers]` and `cache.policy.downstream`. The only list that shows GraphPilot's own `gp-` headers, so it answers "what did my client actually see".

Some headers are never recorded, whatever a rule says, a catch-all `.*` included. Names are matched case-insensitively.

- **Always:** every header whose name starts with `gp-vary-` (the cache's bucket discriminators), every header named in a `[[cache.policy.vary]]` entry, and `gp-shield-hop`.
- **Unless** `telemetry.client_ip` **is** `"full"` (so under `"truncated"` and `"none"`), the headers that carry the client's address: `gp-client-ip`, every header whose name starts with `x-forwarded-`, `forwarded`, `x-real-ip`, `x-client-ip`, `client-ip`, `true-client-ip`, `cf-connecting-ip`, `fastly-client-ip`, `x-original-url`, `x-rewrite-url`, `x-original-host`, `x-cluster-client-ip`, `cf-connecting-ipv6`, `x-envoy-external-address` and

`x-appengine-user-ip`. Under `"full"` a rule that includes them records them like any other header; no default rule does. The last four are not among the forwarding headers that `origin.forwarded_headers` strips.

A rule names either an exact header or a pattern, plus an action:

Action	Effect
`include`	Record the header and its value. The default when a rule names no action.
`sanitize`	Record a keyed digest of the secret part instead of the value: `hmac-sha256:` followed by 16 hex digits, an HMAC-SHA-256 with a secret key held per service. The parts that carry no user data stay readable: a cookie's name, an authorization scheme such as `Bearer`, and `Set-Cookie` attributes such as `Secure` or `SameSite=Lax`. The header is still visible as present, and two requests of one service carrying the same value get the same digest, so they are still comparable. The same value in two services gives two different digests, and a digest cannot be recomputed from candidate values without the key. If the service's key cannot be read, each secret part is recorded as `[redacted]`, never as a plain hash.
`exclude`	Do not record it at all.

```
[[telemetry.headers.request]]
name = "authorization"
action = "sanitize"

[[telemetry.headers.request]]
pattern = "^x-internal-"
action = "exclude"

[[telemetry.headers.request]]
pattern = ".*"
action = "include"
```

The defaults are deliberately narrow. Unless you replace a list, these are the only headers recorded:

List	Sanitized	Recorded
`request`	`authorization`, `cookie`	`content-type`, `accept`, `referer`
`origin_request`	`authorization`, `cookie`, `gp-signature`	`gp-timestamp`, `content-type`, `accept`
`response`	`set-cookie`	`content-type`, `cache-control`, `server`, `content-encoding`
`client_response`	`set-cookie`	every `gp-` header except `gp-surrogate-key` (excluded, the telemetry row carries the keys already), plus `cache-control`, `age`, `content-type`, `content-encoding` and `vary`

> ⚠ **Caution**
>
> Replacing a list replaces it entirely, including the sanitize rules the default provides. A list ending in `pattern = ".*"` with `action = "include"` records **every** header, credentials included. Put your sanitize rules before it.

`[surrogate_key]` and `[cache_control]`

How GraphPilot reads your schema's directives, and which fields count as keys without any directive at all.

`[[surrogate_key.directives]]` and `[[cache_control.directives]]`

Repeatable. Each entry names a directive GraphPilot reads, and the arguments it reads from it. See renaming directives.

The built-in entry uses the values in the second column. In an entry you write, a key marked **required** must be set; an optional `surrogate_key` argument you leave out is unset, and an optional `cache_control` key you leave out takes the built-in value.

Key	Built-in value	In an entry you write
`surrogate_key.directives[].directive_name`	`"surrogateKey"`	**Required.** The directive, without the `@`.
`surrogate_key.directives[].of_arg`	`"of"`	Optional. Unset when left out: the directive reads no key fields, so it works only as a marker.
`surrogate_key.directives[].static_arg`	`"static"`	Optional. Unset when left out: the directive reads no fixed keys.
`surrogate_key.directives[].ignore_arg`	`"ignore"`	Optional. Unset when left out: `ignore: true` on that directive does nothing, and no diagnostic says so.
`cache_control.directives[].directive_name`	`"cacheControl"`	**Required.** The directive, without the `@`.
`cache_control.directives[].max_age_arg`	`"maxAge"`	**Required.** The lifetime argument.
`cache_control.directives[].scope_arg`	`"scope"`	**Required.** The scope argument.
`cache_control.directives[].inherit_max_age_arg`	`"inheritMaxAge"`	**Required.** The inheritance argument.
`cache_control.directives[].swr_arg`	`"swr"`	Optional; `"swr"` when left out. The stale-while-revalidate argument.
`cache_control.directives[].vary_directive_name`	`"vary"`	Optional; `"vary"` when left out. The directive that declares a named scope.

Key	Built-in value	In an entry you write
`cache_control.directives[].vary_names_arg`	`"names"`	Optional; `"names"` when left out. That directive's list argument.

```
[[surrogate_key.directives]]
directive_name = "key"      # read Apollo Federation's @key(fields:)
of_arg         = "fields"
```

An entry replaces the built-in directive (see the caution below). `directives = []` switches the family off, so no directive is recognized.

Automatic key fields

Key	Type	Default	Does
`surrogate_key.scalars`	list of strings	`["ID"]`	Scalar type names whose fields are treated as key fields.
`surrogate_key.names`	list of strings	`["id", "_id", "key"]`	Field names treated as key fields whatever their type.
`surrogate_key.ignored_types`	list of strings	`[]`	Object types and interfaces the two rules above do not key. An interface listed here does not take the types implementing it along.
`surrogate_key.ignored_fields`	list of strings, each `"Type.field"`	`[]`	Single fields of an object type or interface the two rules above do not key.
`surrogate_key.mutation_payload_depth`	integer	`1`	How many levels below a mutation's return type the deployment build looks for types that carry surrogate keys, so a mutation purges an entity the client did not select. `1` reaches the entity inside a Relay-style payload (`updateX: UpdateXPayload { x: X }`); `0` switches this off. `2` also reaches the entity's owner and purges every cached response mentioning it, which is broad: prefer an owner key on the entity's type, such as `@surrogateKey(of: "service.id")` . See automatic invalidation.

```
[surrogate_key]
ignored_types  = ["RenderedSnippet"]
ignored_fields = ["Product.key"]
```

The exclusions switch off only those two automatic rules, never a key your schema declares; the build reports the conflict as a **warning**. The schema makes the same exclusions with `@surrogateKey(ignore: true)` . An entry that

matches no object type, interface or field is a **warning**, and an `ignored_fields` entry not of the form `Type.field` is an **error**; both exclude nothing and neither fails the deployment.

> ⚠ **A directives entry replaces the built-in one**
>
> Each key you leave out of `[surrogate_key]` or `[cache_control]` keeps its default, so opening the section to set `ignored_types` alone keeps `@surrogateKey`, the key scalars and the key names. `directives` is a list, though: one `[[surrogate_key.directives]]` entry replaces the built-in entry. Restate `surrogateKey` beside your own if you still use it, including `ignore_arg = "ignore"`. The same applies to `[[cache_control.directives]]` and `@cacheControl`.

`[runtime]`

Key	Type	Default	Does
`runtime.version`	string	none	The proxy runtime version for the one deployment that `gpilot deployment create` makes from this file. See Runtime versions.

This key pins only that deployment. The CLI checks the version first and refuses an unknown or withdrawn one before uploading anything, removes `[runtime]` `version` from the uploaded config and sends it as the deployment's runtime version. It is not the service's pin: a redeploy from the portal, or a change of the pin, does not carry it. Set the service-level pin in the portal or with `gpilot service runtime pin <version>`. Changing it redeploys the service unless the service already runs the version the pin resolves to, or you save it for the next deployment in the portal; either way it needs `service:edit` and `deployment:edit`.

Settings that do nothing

Stated so nobody configures them expecting an effect:

- The `[deployment]` section and `runtime.serving_mode` are written by GraphPilot. A value you supply is ignored.

Validation

Unknown keys

Whether a key GraphPilot does not know is refused depends on the section it is written in.

Refused. These sections deny unknown keys, so a misspelling in them fails when the file is read and the deployment does not start:

- `[client]` and every `[[client.profile]]`
- `[persisted_query]`
- `[response]`, `[response.headers]`, `[response.cookies]`, and every entry in their `rules`
- `[response.cors]`
- `[origin.headers]`, `[origin.cookies]` and `[origin.retry]`

- `[cache.request_directives]`, `[cache.http_key]`, `[cache.headers]` and `[cache.cookies]`
- `[auth.rejection]`

Ignored. Everywhere else an unknown key deploys cleanly and does nothing: the top level of the file, `[request]`, `[origin]`, `[cache]`, `[cache.policy]`, every `[[cache.policy.vary]]` entry, `[auth]`, every `[auth.providers.<name>]` block, `[telemetry]`, `[telemetry.headers]`, `[surrogate_key]` and `[cache_control]`, and `[runtime]` apart from its `version` key. So a typo like `[cache.polcy]` or `[reqest]`, or a misspelled key inside a provider block, is silent at deploy time.

> ⚠ **A misspelled key is often silent**
>
> If a setting appears to have no effect, check its spelling against this page before anything else. The deployment build also records every key it could not place, and every key that is accepted but read by nothing. The portal lists both under **Config → Findings** for the live deployment and flags the lines in the **gpilot.toml** view. See findings. On the GraphPilot API they are each deployment's `configAnalysis.unknown` and `configAnalysis.inert`, with `path` and `line`. The `gpilot` CLI does not show these two lists.

Other refusals

- An **invalid enum value** fails when the configuration is read. Values are matched exactly as spelled here; there is no case folding.
- An **unparseable policy expression** fails, and the whole expression must parse: a typo does not become a shorter, different expression.
- A `[[cache.policy.vary]]` **entry whose** `name` **is not a legal header name** is the exception to that: it does not fail, it is skipped, so the rule never fires. The deployment records it as an error in its diagnostics, which the portal shows under findings and `gpilot deploy` prints when the deployment finishes.
- An `api_key` **provider with an empty** `prefixes` **list**, or one containing an empty string, is refused with a message saying why.
- **A duplicate provider name** is rejected by TOML itself, before any of this runs.
- **A provider name longer than 977 bytes** (UTF-8) refuses the deployment with `AUTH_PROVIDER_NAME_TOO_LONG`, before a deployment is created. Every `[auth.providers.<name>]` block counts, with or without a `jwks_url`.
- An **origin timeout above your plan's ceiling or below** `100` **ms**, or a retry policy whose worst case does not fit 120 seconds, refuses the deployment. See `[origin]`.
- A **persisted query combination** that contradicts itself is refused. See `[persisted_query]`.

How to write a duration

`time_tolerance` and `max_validity` are the only duration-valued settings. Each is a table of two non-negative integers, `secs` and `nanos`, and both keys are required:

```
[auth.providers.idp]
time_tolerance = { secs = 30, nanos = 0 }
max_validity = { secs = 3600, nanos = 0 }
```

A plain number (`30`), a string (`"30s"`), a table missing either key, or any other key name refuses the deployment. A `nanos` value of one second or more carries into `secs` .

REFERENCE

GraphQL directives

GraphQL only A plain HTTP API has no schema; it uses Cache-Control and the Surrogate-Key header instead.

GraphPilot reads these directives from the schema you deploy, when it builds the deployment. None of them is forwarded to or needed by your origin. What each one is for is on the pages linked from each section; this page is the exact surface.

Directive	Decides	Concept
`@cacheControl`	How long a response lives and who shares it	Lifetime from the schema, scopes
`@vary`	What a named scope partitions on	Scopes in the schema
`@surrogateKey`	Which surrogate keys a response carries	Keys from the schema
`@auth` , `@graphpilotPolicy`	Nothing: parsed and reported, not enforced	

No finding on this page fails a deployment. Each is recorded on the deployment as a diagnostic, shown in the portal under **Schema → Findings**, printed by `gpilot deploy` , and returned by the GraphPilot API.

`@cacheControl`

schema.graphql

```
directive @cacheControl(
    maxAge: Int
    scope: CacheControlScope
    inheritMaxAge: Boolean
    swr: Int
) on OBJECT | INTERFACE | UNION | FIELD_DEFINITION
```

```
enum CacheControlScope {
    PUBLIC
    PRIVATE
}
```

Argument	Type	Default	Meaning
`maxAge`	`Int`	The default of the `maxAge` argument in your definition, else `0`	The freshness lifetime in seconds. An explicit `maxAge` always wins over inheritance, on the type or the field.
`scope`	`CacheControlScope`	`PUBLIC`	Who may be served the entry: `PUBLIC`, `PRIVATE`, or a named scope you declare on the enum. Omitted means `PUBLIC`, as in Apollo Server.
`inheritMaxAge`	`Boolean`	`false`	On a non-root field with no `maxAge`, take the parent field's effective `maxAge` instead of the default. No effect on a root field.
`swr`	`Int`	none	Seconds past `maxAge` an entry may still be served as `STALE` while a fresh copy is fetched. A field that states no `swr` does not switch the window off for the query. See stale windows.

Location	Effect
`OBJECT`, `INTERFACE`, `UNION`	Applies to every field that returns this type, unless that field has its own hint. On a member of a union or an implementor of an interface, the `scope` also joins every field returning that union or interface; its `maxAge` does not. See a field returning a union or interface.
`FIELD_DEFINITION`	Applies to that field only. Each argument it sets overrides the return type's hint; each one it leaves out, `scope` included, comes from the type. See a field hint overrides only the arguments it sets.

The default `maxAge`

The fallback for an unannotated root or composite field (Apollo's `defaultMaxAge`) is the **default value you declare for the `maxAge` argument** on the directive itself:

schema.graphql

```
directive @cacheControl(
    maxAge: Int = 30              # unannotated composite fields fall back to 30, not 0
    scope: CacheControlScope
    inheritMaxAge: Boolean
    swr: Int
) on OBJECT | INTERFACE | UNION | FIELD_DEFINITION
```

`maxAge: Int` with no default leaves the fallback at `0`, which is uncacheable, as in Apollo Server. A non-integer default is ignored and also leaves it at `0`.

What the build reports

Written	Reported as	Effect
`@cacheControl` used without a matching directive definition	Error	The hints are still resolved.
On any other location (a scalar, an enum, an input type or field)	Error	The hint is dropped.
On a mutation field	Warning	The hint is dropped; a mutation is never cached.
An argument name other than `maxAge`, `scope`, `inheritMaxAge` and `swr` (or your configured names)	Error	The whole hint is dropped, and the build names the argument, so a typo such as `maxage` is reported rather than passing in silence.
An argument of the wrong type: a `maxAge` or `swr` that is not a whole number from `0` to `4294967295`, an `inheritMaxAge` that is not a Boolean, a `scope` that is not an enum value	Error	The whole hint is dropped.
A `scope` value other than `PUBLIC` and `PRIVATE` that carries no `@vary`, or whose `@vary` is malformed	Error	The hint is kept with the scope read as `PRIVATE`, the narrowest audience it could have meant.
A field returning a union or an interface whose members state a lower `maxAge` than the field resolves to	Warning	Nothing changes: the response is cached for the field's `maxAge`. The warning names the field and each member below it. A field that inherits its parent's `maxAge` is not compared.

`@vary`

Declares a named scope: a value of your own `CacheControlScope` enum that partitions on the vary rules it lists.

schema.graphql

```graphql
directive @vary(names: [String!]!) on ENUM_VALUE

enum CacheControlScope {
    PUBLIC
    PRIVATE
    ORGANIZATION      @vary(names: ["x-org"])
    ORGANIZATION_ROLE @vary(names: ["x-org", "x-role"])
}
```

Argument	Type	Meaning
`names`	`[String!]!`	Header names, matched case-insensitively against the `name` of your `[[cache.policy.vary]]` entries.

Rule	Detail
Location	`ENUM_VALUE`, on the enum your `scope` argument declares. `PUBLIC` and `PRIVATE` take no `@vary`: their meaning is built in, and one written on either is ignored.
A malformed `names`	Missing, not a list of strings, or empty: an error, and every `scope` naming that value is read as `PRIVATE`.
A name with no `[[cache.policy.vary]]` entry	Not a deployment failure. Every response in that scope is refused (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`), and the build reports it at the annotation. See what the deployment build reports.
A name whose `[[cache.policy.vary]]` entry is not a legal header name	The same: the entry is skipped, every response in that scope is refused, and the build reports it at the annotation.
A name in the `gp-` namespace	The build reports it at the annotation, and the deployment does not fail. With a `[[cache.policy.vary]]` entry of that name, the scope still partitions on it: GraphPilot strips the namespace from what the client sends, then derives the entry's value. Without one, it is refused like any name with no entry. Rename it outside the namespace anyway: the namespace is also stripped before your origin, so the entry's `pass_to_origin` has no effect.
Composition	The names of every scope a query selects are unioned. See how scopes compose.

`@surrogateKey`

schema.graphql

```
directive @surrogateKey(
    static: String
    of: String
    ignore: Boolean
) repeatable on SCALAR | OBJECT | FIELD_DEFINITION | INTERFACE | UNION | ENUM
```

The definition may be made more restrictive (`String` to `String!`, for example), not extended to new types or locations. A bare `@surrogateKey`, with no argument, marks a field, scalar or enum as a key field. Examples for every location are on keys from the schema.

Argument	Type	Meaning
`static`	`String`	One or more fixed keys, separated by whitespace. See a group you purge by hand.

Argument	Type	Meaning
`of`	`String`	The fields that form the key, as a GraphQL selection set (`"user { id }"`) or in path syntax (`"user.id"`). Several fields form a composite key; the two notations can be mixed.
`ignore`	`Boolean`	`true` switches the automatic key rules off where the directive sits. See below.

`of`

Rule	Detail
Leaf selections only	Each path ends on a scalar, an enum, or a list of these (`[String!]` , `[Role!]`), never on an object, interface or union.
Fields without required arguments	A field whose arguments all have defaults, or that has none.
No globs or recursion	`*` , `**` and recursive selections are not supported; each path names a field of the enclosing type.
Only where a traversal is possible	On an object, interface or union. Elsewhere the argument is ignored.
A nullable path	Allowed, and reported as a **warning**: an object whose value on that path is `null` is still tagged, with an empty value (`Book:isbn:` for `@surrogateKey(of: "isbn")` on `Book`), so it cannot be purged by identity. A path is nullable when any step of it is: `author.id` is null whenever `author` is.

`ignore`

Written on	Effect
An `OBJECT` or `INTERFACE`	No field of that type is keyed by the automatic rules.
A field of an object type or interface	That one field is not keyed by the automatic rules.
A `SCALAR` , `ENUM` or `UNION`	Nothing. The build reports a **warning**.

Rule	Detail
Only the automatic rules	An exclusion never removes a key you wrote. `@surrogateKey(ignore: true)` alone is the opt-out and nothing else; it is not read as a bare `@surrogateKey` .
Beside an explicit key	`@surrogateKey(ignore: true, of: "slug")` swaps the automatic keys for the one written, with no warning.
Interfaces	An interface's exclusion does not reach the types implementing it.

Rule	Detail
Values	Only `true` switches anything off. `false` is the same as leaving the argument out; any other value is a schema **error** and the automatic keys stay.
Against the configuration	When `surrogate_key.ignored_types` or `ignored_fields` excludes something a directive keys explicitly, the directive wins and the build reports a **warning**.

When a directive produces no key

The build refuses the directive, with the type, the directive and its line and column, and the deployment still succeeds:

- an `of` naming a field that does not exist, or a path that does not end in a scalar, enum or list thereof;
- an argument of the wrong type;
- a bare `@surrogateKey` on a type that needs a selection;
- a directive written where no key can come from, such as inside `extend type` or on a field argument.

Renaming directives

Each directive can be read under a name of your own, configured in your service configuration. The keys and their rules are on `[surrogate_key]` and `[cache_control]`.

Directive and argument	Renamed by
`@cacheControl`	`[[cache_control.directives]]` `directive_name`
`maxAge`, `scope`, `inheritMaxAge`, `swr`	`max_age_arg`, `scope_arg`, `inherit_max_age_arg`, `swr_arg`
`@vary` and its `names`	`vary_directive_name`, `vary_names_arg`
`@surrogateKey`	`[[surrogate_key.directives]]` `directive_name`
`static`, `of`, `ignore`	`static_arg`, `of_arg`, `ignore_arg`

> ⚠ **A directives entry replaces the built-in one**
>
> Writing one `[[surrogate_key.directives]]` or `[[cache_control.directives]]` entry replaces the built-in directive rather than adding to it. Restate `@surrogateKey` or `@cacheControl` beside your own if you still use it. An empty `directives = []` switches the family off.

To read Apollo Federation's `@key(fields:)` as a surrogate key directive, for example:

gpilot.toml

```
[[surrogate_key.directives]]
directive_name = "key"
of_arg         = "fields"
```

`@auth` and `@graphpilotPolicy`

schema.graphql

```
directive @graphpilotPolicy(dsl: String) on ARGUMENT_DEFINITION | ENUM_VALUE
```

`@graphpilotPolicy(dsl:)` attaches a policy expression to a directive argument or to an enum value, so that a directive such as `@auth(requires: [Role!])` can state access rules in the schema.

> ⚠ **Parsed and reported, not enforced**
>
> The deployment build parses these directives and reports mistakes in them as diagnostics with the source `auth`. Nothing at the edge acts on them: a field annotated with `@auth(requires:)` is served to every caller exactly as it would be without the annotation. Enforce access in your origin.

REFERENCE

Policy expressions

Several settings in your configuration file (`gpilot.toml`) take an expression rather than a value:

Setting	What the expression produces
`cache.policy.public`	The value the shared entry is partitioned by.
`cache.policy.private`	The value a private entry is partitioned by.
`cache.policy.vary[].value`	The value written onto the request as a header.
`cache.policy.bypass`	A truth: whether to skip the cache for this request.
`client.profile[].when`	A truth: whether this request came from that declared client.
`client.profile[].version`	The version that client reported.
`response.headers.rules[].when`, `response.cookies.rules[].when`	A truth: whether that conditional response filter applies to this request.
`telemetry.record`	A truth: whether this request is recorded in Insights.

They all use the same small language, described completely below. It has no variables, no loops and no access to the response: an expression reads the request and returns a string, or returns nothing.

Sources

An expression reads from exactly four places.

Source	Syntax	Reads
Claims	`auth:<provider>.<path>`	A claim from a credential a provider already verified. The path may be nested: `auth:idp.org.id`. A claim whose name is a URL is quoted: `auth:idp.'https://acme.example/roles'`. A string reads as itself, a number or a Boolean as its JSON text (`42`, `1.5`, `true`), so `auth:idp.is_admin == 'true'` matches a Boolean claim. `null`, an object and an array resolve to nothing.
Headers	`header:<name>`	A request header, case-insensitively. A repeated header gives its first value, and a value with bytes outside printable ASCII resolves to nothing.
Cookies	`cookie:<name>`	A cookie from the request's `Cookie` header, the first one of that name, matched case-sensitively: the whole value after the first `=`, so `token=abc==` resolves to `abc==`, with surrounding whitespace and quotes removed. `name=` resolves to the empty string.
Query parameters	`query:<name>`	One query parameter, percent-decoded, with `+` read as a space. The **first** occurrence of a repeated name.

`header:`, `cookie:` and `query:` names may contain letters, digits, `_`, `-` and `.`. An `auth:` path is steps separated by `.`; a step is letters, digits, `_` and `-`, or any name in single or double quotes (`auth:idp.'https://acme.example/roles'`), which is one step whatever it contains and escapes like a string literal. An empty step does not parse.

> **Note**
>
> `auth:` reads a **verified** claim. The provider checked the signature, the expiry and the algorithm before any expression ran, so a claim that resolves is one GraphPilot vouches for. `header:`, `cookie:` and `query:` are whatever the caller sent, and are only as trustworthy as your origin makes them. A discriminator that carries a security boundary must therefore read from `auth:`; see a discriminator is only as trustworthy as its source.

Operators

Loosest binding first:

Operator	Spelling	Meaning
Or	`||`, `OR`	Either side.
And	`&&`, `AND`	Both sides.
Not	`!`, `NOT`	Inverts a truth. An expression that could not be evaluated has no truth to invert, so negating it does not produce one. See `!` and a missing value.
Comparison	`==`, `!=`, `<`, `<=`, `>`, `>=`	`==` and `!=` compare two operands as **strings**; the other four compare them as **numbers**. See comparisons are not all alike.
Concatenation	`+`	Joins operands into one string, **skipping the ones that do not resolve**, except in a cache key, where every operand is required.

Parentheses group. All six comparison operators share one precedence level and chain left to right. Comparison binds tighter than `&&` and `||` and looser than `+`, so `auth:idp.role == 'admin' || auth:idp.sub == 'u1'` reads as two comparisons joined by an or.

> **`&&` and `||` mean two things, depending on the setting**
>
> Where the setting wants a **value**, `&&` joins both sides into one string and `||` yields the first side that resolved: they assemble a cache key. Where it wants a **truth**, they are the ordinary boolean and and or. Which applies is decided by the setting, never by the expression. See values versus truths.

`+` and `&&` both join strings, and differ on what is missing

As a value, `&&` concatenates too. Outside the cache key the two differ on a missing part:

- `+` **keeps going.** `auth:idp.org + ':' + auth:idp.role` on a token carrying no role resolves to `org_7:` rather than to nothing.
- `&&` **requires every operand.** The same expression written `auth:idp.org && ':' && auth:idp.role` resolves to nothing when the role is absent.

Combining sources in a cache key

In the three values that partition the cache, `cache.policy.private`, `cache.policy.public` and every `[[cache.policy.vary]]` `value`, a chain of `+` and `&&` is derived more strictly, so that two different callers can never end up with the same key:

- **Every operand is required.** The chain resolves to nothing as a whole when any operand does, for `+` as well as `&&`. A caller missing a claim is then not cached, or shares the variant without the header, rather than sharing a partition keyed on what was left.
- **Each source keeps its boundary.** When two or more operands read the request, each of them is written as its length in UTF-8 bytes, a colon, and the value. Literals stay as you wrote them. `auth:idp.org + ':' + auth:idp.role` for org `org_7` and role `admin` derives `5:org_7:5:admin`, so org `a:b` with role `c` (`3:a:b:1:c`) and org `a` with role `b:c` (`1:a:3:b:c`) can no longer meet on `a:b:c`.

A chain with a single source, such as `'tenant-' + auth:idp.org`, derives the same text as before (`tenant-o7`), and so does an expression with no `+` or `&&`. A literal prefix such as `'v2/' + …` still works as a cache buster. `||` still yields the first side that resolved, each side derived by these rules. A chain inside a function argument or a comparison is that function's input and keeps the plain meaning above. Conditions (`when`, `bypass`, `telemetry.record`) and `[[client.profile]].version` are not cache keys and keep it too. See what every expression resolves to for measured examples.

Functions

A function whose result is a yes-or-no answer is called a **predicate** below. It resolves to the string `true` or `false`, which is what a truth context reads back.

Text

Function	Resolves to	Nothing when
`lower(s)`, `upper(s)`	`s`, case-folded	`s` is absent
`starts_with(s, prefix)`	predicate	either argument is absent
`ends_with(s, suffix)`	predicate	either argument is absent
`contains(s, needle)`	predicate	either argument is absent
`len(s)`	the number of characters	`s` is absent
`split(s, separator, index)`	the field at `index`, counting from `0`	`s` or the separator is absent, the separator is empty, the index is not a non-negative number, or there is no such field

Case folding keeps a partition from splitting on casing your callers do not control: `lower(header:x-customer-id)`.

`split` is how a composite header becomes its parts:

```
split(header:user-agent, '/', 1)      # 'gp-cli/0.8.0' gives '0.8.0'
```

An index past the end resolves to nothing rather than to an empty string, so a vary key built from it fails to build instead of collapsing every caller into one shared bucket.

Membership

Function	Resolves to	Nothing when
`in(x, [a, b, c])`	predicate: does `x` equal any candidate	`x` is absent

Every argument after the first is a candidate, and a list contributes its elements, so `in(x, ['a', 'b'])` and `in(x, 'a', 'b')` are the same expression. A candidate that resolves to nothing is skipped: one unreadable alternative does not make the rest undecidable.

```
in(header:x-region, ['eu', 'us'])
```

Versions

Function	Resolves to
`semver_gte(a, b)`, `semver_gt(a, b)`, `semver_lte(a, b)`, `semver_lt(a, b)`, `semver_eq(a, b)`	predicate: `a` against `b`, compared as versions

Both sides are read as semantic versions, so `2.10.0` is newer than `2.9.0`, and `1.0.0-rc.1` is older than `1.0.0`. A missing component reads as zero, so `2.9` is `2.9.0`. Build metadata does not change a version: `1.0.0+ci.7` and `1.0.0` are the same release and clear the same floor.

Anything that is not a version (`v2.9.0` , `latest` , a word) resolves to **nothing**, not to “older than the floor”, so a floor written as `semver_gte( … )` never admits it. Under `!` it does match, like everything else that could not be evaluated: `!semver_lt(header:x-app-version, '2.0.0')` matches a request whose version does not parse. See `!` and a missing value.

Together with `split` , that covers the client whose version is only in its User-Agent:

```
[[client.profile]]
name = "gp-cli"
when = "starts_with(header:user-agent, 'gp-cli/') && semver_gte(split(header:user-agent,
'/', 1), '0.7.0')"
version = "split(header:user-agent, '/', 1)"
```

Pseudonyms

Function	Resolves to	Nothing when
`pseudonymize(s)`	a keyed pseudonym of `s` : 43 characters of unpadded base64url	`s` is absent

`pseudonymize` replaces a value with a keyed digest (HMAC-SHA256) made with a secret key of your service that neither you nor your origin holds. Equal inputs give equal pseudonyms within the service, so a partition built on one still groups the same callers, while the cache, a `plain` origin and your request telemetry never see the value itself. The same input gives a different pseudonym on another service, and the pseudonym is not the `v1=` digest an `anonymized` origin receives.

Hash the part that identifies a person and keep the rest readable:

```
[[cache.policy.vary]]
name  = "x-org-user"
value = "auth:idp.org_id + ':' + pseudonymize(auth:idp.sub)"
```

- **Only in the values that partition the cache.** `pseudonymize` is accepted in `cache.policy.public` , `cache.policy.private` and `cache.policy.vary[].value` . Anywhere else **the deployment is refused**: a truth would compare the pseudonym with a value nobody can know, and `client.profile[].version` is read as a version.
- **A pseudonym is never replaced by the value itself.** The cache, a `plain` origin and your request telemetry see the pseudonym, not the input.
- **Insights stores a digest of the pseudonym.** With `telemetry.vary_values` at its default, a vary rule’s value is stored as its own `hmac-sha256:` digest, so a pseudonym is stored hashed again. Requests of one segment still group together, but the stored value is not the pseudonym a `plain` origin received. With `vary_values = "clear"` , Insights stores the pseudonym as it is.

Literals

Literal	Example
String	`'admin'`, `"admin"`
Integer	`5`, `-1`
Decimal	`1.5`
Boolean	`true`, `false`
List	`['eu', 'us']`

Strings take single or double quotes; `\'`, `\"`, `\\`, `\n` and `\t` are the escapes. They resolve in one left-to-right pass, and a backslash before any other character is kept as written (`'a\qb'` is `a\qb`). So `'a\\n'` is `a`, a backslash and `n`, never a newline.

A list is not a value. It exists for `in`, and evaluating one anywhere else resolves to nothing rather than quietly becoming part of a cache key.

`true` and `false` are lowercase whole words (`TRUE`, `False` and `trueish` do not parse). As a truth each is decided for every request and never undecidable, so an expression that is only a literal answers the same way for all traffic: `telemetry.record = "false"` records nothing. As a value each is its own spelling, the string a comparison resolves to, so `header:x-flag = true` is a text match against `"true"`. Requires a runtime that ships them; older runtimes refuse the expression at deploy.

Comparisons are not all alike

`=` and `≠` compare the two values as **strings**. `<`, `⩽`, `>` and `⩾` compare them as **numbers**, and if either side is not a number, the comparison resolves to nothing, which is not a match.

```
header:x-count > 5                  # true when the header is 42
header:x-ratio ⩾ 1.5                # decimals work
header:x-plan > 'gold'              # nothing: neither side is a number
header:x-app-version ⩾ '2.9.0'      # nothing, for every version ever sent
```

The last line is the one that costs people an afternoon. It parses, it deploys, and it never matches, because `2.9.0` is not a number. Nothing is logged, because an expression that resolves to nothing is an ordinary “did not match” rather than an error.

Ordering is deliberately not a string comparison. Lexically `'2.10.0' ⩾ '2.9.0'` is *false*, the opposite of what anyone writing it means, so a version floor built that way would refuse your newest release. **Compare versions with `semver_gte` instead**, which reads both sides as versions.

Whole numbers compare exactly, including ones past the signed 64-bit range, so an id orders correctly against its neighbor. A decimal, an exponent, or an integer longer than 39 digits moves both sides to floating point, where two neighboring values can round onto each other.

Resolving to nothing

A source that is not present on the request resolves to nothing, and that propagates:

- `a && b` resolves to nothing if either part does.
- `a + b` resolves to whichever parts did resolve, and to nothing only if none did. In a cache key it resolves to nothing if either part does; see combining sources in a cache key.
- `a || b` resolves to `b` if `a` does not, and to nothing if neither does.
- A comparison whose operands do not both resolve has **no answer**: it is neither a match nor a failure to match, and that is a third state the rest of this page calls *undecidable*.

For the settings that produce a value, nothing resolved is the whole story:

Setting	An expression that resolves to nothing
`public`	The entry is not partitioned by it.
`private`	A **private** response (origin `Cache-Control: private`, or a schema `PRIVATE` scope) is **not cached**, reported as `CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`: there is no way to keep one caller's copy from another's, so it fails closed. An unmarked response to a credentialed request is refused as `CACHE_SKIPPED_CREDENTIALED_REQUEST`. A public response is unaffected and cached as usual.
`vary[].value`	No header is written. A request without a credential is stored as the variant without it; one with a credential is not stored (`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`).
`client.profile[].version`	The client is recognized, with no version recorded against it.

> ⚠ **An empty string is a value**
>
> An expression that resolves to the **empty string** has resolved. A blank header, `?x=` in the query string, or a cookie sent as `name=` all produce one. That caller is stored under the empty-value variant, shared with every other caller whose value is empty, and a `private` policy or a named scope does not refuse the response for it. Derive a partition that has to separate callers from an `auth:` claim, whose value the caller cannot choose.

For the settings that produce a truth, the undecidable case is decided by the setting itself, and each one picks the direction that withholds rather than grants. See a request the expression cannot decide.

`!` and a missing value

`!` inverts whether an expression matched, and an expression that could not be evaluated for want of a source never matched or failed to match. There is nothing there to invert, so `!` passes the undecidable state through rather than turning it into a match:

```
!starts_with(header:user-agent, 'gp-cli/')   # undecidable for a request with no User-Agent
```

```
!(auth:idp.tier == 'free')                    # undecidable for every request without a
token
```

This is what makes the two spellings of one condition one condition. Both of

```
auth:idp.role ≠ 'admin'
!(auth:idp.role == 'admin')
```

are undecidable for a caller with no role, so they decide alike. They are the same expression to anyone reading them, and the language agrees.

A bare source is never undecidable

`header:x` asks whether the caller sent it, and that question always has an answer: a match when it resolved to anything other than the empty string or the exact string `false`, a no otherwise. The same reading applies to any value read as a truth, so a header sent as `x-flag: false` is a no, and `x-flag: 0` is a match. So `!header:x-preview` is exactly the absence test it looks like, and a bare source is what lets you decide the missing case yourself:

```
auth:idp.tier && !(auth:idp.tier == 'free')    # false for a caller with no tier
!auth:idp.tier || !(auth:idp.tier == 'free')   # true for a caller with no tier
```

`&&` and `||` over three values

An operand that cannot change the answer does not need one:

Expression	Result
`false && undecidable`	no match
`true && undecidable`	undecidable
`true || undecidable`	match
`false || undecidable`	undecidable

Either order: `undecidable && false` is also no match.

A request the expression cannot decide

Every setting that asks for a truth says what an undecidable expression means for it, and each picks the safe direction for what it does:

Setting	An undecidable expression
`cache.policy.bypass`	Does **not** bypass. The request is cached like any other.
`client.profile[].when`	Does **not** name the client. It stays unknown, and `client.unknown` decides it.

Setting	An undecidable expression
`response.*.rules[].when`	**Applies** the rule. Its `hide` or `only` list is enforced.
`telemetry.record`	**Records** the request. It is kept in Insights like any other.

The split is not an inconsistency. A setting whose match *grants* something, the origin or a named profile with its own limits, withholds it from a request carrying none of the context the expression names. A setting whose match *takes something away*, a `hide` list, takes it away: a filter that cannot tell whether a condition held must not be the one that reveals. A setting whose match *keeps data*, `telemetry.record`, keeps it: an expression over a header some callers omit must not make their traffic disappear from Insights.

To decide it in the expression instead of leaving it to the setting, say which way you mean, using a bare source as shown above.

The deploy warns about an unguarded `!`

Because the expression alone does not say whether the setting's direction is what you meant, the deployment reports a **warning** for every `!` applied to a predicate over a source nothing has required to be present, on the settings that ask for a **truth**. The warning names the setting, the expression and the sources, and shows both spellings that decide:

```
`client.profile[0].when` (`!(auth:idp.tier == 'free')`) negates a predicate over
`auth:idp.tier`, so it has no answer for a request that carries no such value: the
predicate
cannot be evaluated there, and negating something undecidable leaves it undecidable. Those
requests are decided by this key's default instead of by the expression: a
`cache.policy.bypass`
does not bypass them, a `[[client.profile]].when` leaves them an unknown client, a
`[[response.headers.rules]].when` applies the rule and hides, a `telemetry.record` records
them.
If that is what you meant, nothing needs to change. If not, say what those requests should
do by
requiring the value before negating it, writing `auth:idp.tier && ` in front of the
negation, as
in `header:x && !starts_with(header:x, 'prefix')`, or by admitting them explicitly with
`!header:x || ` in front of it.
```

The guard in front of the negation names your sources; the two `header:x` spellings after it are fixed examples. For the expression above, the admitting spelling is `!auth:idp.tier ||` .

It never fails the deployment. If the setting's direction is what you meant, leave the expression as it is: it deploys and behaves exactly as written.

Expression	Warned	Why
`!starts_with(header:user-agent, 'gp-cli/')`	yes	The function cannot be evaluated without the header.
`!(auth:idp.tier == 'free')`	yes	The comparison cannot be evaluated without the claim.
`header:x || !starts_with(header:x, 'a')`	yes	A bare source on the other side of `||` is not a guard: the right side runs exactly when the header is missing.
`header:user-agent && !starts_with(header:user-agent, 'gp-cli/')`	no	The conjunction requires the header first. Either side of `&&` works.
`!auth:idp.role || auth:idp.role ≠ 'admin'`	no	No predicate is negated.
`!auth:idp.role || !(auth:idp.role == 'admin')`	no	The missing case already matches through the left side.
`!header:x-preview`	no	Negating a bare source **is** the absence test.
`auth:idp.role ≠ 'admin'`	no	No `!` to look at, though it is undecidable for the same callers.
`public = "!starts_with(header:x, 'a')"`	no	A setting that asks for a value, where nothing is undecidable.

Only a bare source counts as a guard. `starts_with(header:x, 'a') && !ends_with(header:x, 'b')` is still warned about: not every predicate over a source requires it (`+` keeps whichever side resolved, `in` skips a missing candidate), and the check does not assume a guard it cannot see.

The `≠` row is worth being straight about. The warning looks for a `!`, and `auth:idp.role ≠ 'admin'` carries none, so it draws no warning even though it is now exactly the same expression as `!(auth:idp.role == 'admin')` and undecidable for exactly the same callers. A check that reported every undecidable expression would report almost all of them, since any comparison against a claim is undecidable for a caller without it. So this one flags the spelling most likely to be misread, not every case: treat a clean deployment as “nothing obvious”, not as “every expression decides every request”.

The last row is a different case. A setting that resolves a **value** has no undecidable state at all: `!` there yields the string `"true"` whenever its operand did not resolve, and that string is what the setting uses. There is no direction standing in for a missing answer, so there is nothing for this warning to be about, and requiring the value first would change what the setting resolves to rather than pin down an answer it was missing. What `!` does cost in a vary value, folding every caller missing the source into one bucket with everyone the predicate was false for, is a cardinality question.

Values versus truths

`bypass`, `client.profile[].when`, the `when` of a conditional response filter and `telemetry.record` are evaluated as **truths**; the rest are evaluated as **values**. This is not interchangeable, and it is the one place the language can surprise you.

Which one applies is decided by the setting you write the expression under, not by anything in the expression itself. The same text can mean two things in two places.

As a value, `&&` and `||` combine strings. As a truth, they combine truths, over the three states described under `!` and a missing value. Reading a value expression's output as a boolean would grant a bypass on the very expressions written to withhold it, so GraphPilot evaluates the two differently rather than converting between them. `auth:idp.role == 'admin'` as a value resolves to the string `"false"` for a caller who is not an admin, which is a value that resolved; as a truth it is a no.

Write `bypass` as a predicate. A service has one `bypass` key, so these are alternatives:

```
[cache.policy]
bypass = "header:x-preview-token"                                  # presence check
# bypass = "auth:idp.role == 'editor'"                             # or a verified claim
# bypass = "header:x-preview-token && auth:idp.role == 'editor'"   # or both
```

Errors

An expression is parsed when the service is deployed, and the whole expression must parse: a typo fails rather than silently becoming a shorter, different expression. That matters most for `bypass`, where a truncated predicate would be weaker than the one you wrote.

An expression that parses is still refused where it calls `pseudonymize` outside the values that partition the cache, and the error names every setting that does.

An expression that parses can still draw a warning for an unguarded `!`. Warnings are listed in the deployment's diagnostics and never fail it.

Cardinality

Everything here reads what the caller sent, and anything that reaches a cache key multiplies the number of entries your service holds. `split(header:user-agent, '/', 1)` as a vary value partitions your cache by every client version in the wild.

Prefer a predicate that collapses many values into one (`semver_gte( … )`, `in( … )`) over a value that carries them all.

What every expression resolves to

Everything in this section is measured against the evaluator rather than described, and kept in step with it by an automated test.

The caller sending these requests carries a verified `idp` token with `sub=u1`, `role=editor`, `tier=pro`, `org.id=o7`, the headers `user-agent: gp-cli/0.8.0`, `x-org: acme`, `x-blank:` (empty), `x-count: 42`, `x-version: 2.10.0`, the cookies `session=abc; theme=dark`, and the query string `?lang=de&lang=fr&empty=`.

A dash means the expression resolved to nothing.

Sources

Expression	As a value	As a truth
`auth:idp.role`	`"editor"`	match
`auth:idp.org.id`	`"o7"`	match
`auth:idp.missing`	–	no
`header:x-org`	`"acme"`	match
`header:x-blank`	`""`	no
`header:x-absent`	–	no
`cookie:session`	`"abc"`	match
`query:lang`	`"de"`	match
`query:empty`	`""`	no

A header the caller sent blank resolves to the empty string, which is a value that *resolved* and a truth that is *no*. A repeated query parameter gives the first occurrence.

What does not parse

The whole expression must parse, and a typo fails the deployment rather than quietly becoming a shorter one.

Expression	
`'a literal'`	resolves to `"a literal"`
`"double quoted"`	also a string literal, though in TOML you would have to escape the quotes
`true`	resolves to `"true"`, and as a truth matches every request
`false`	resolves to `"false"`, and as a truth matches no request
`auth:idp.role = 'editor'`	**does not parse.** Equality is `==`
`len(header:x-org) > 2`	resolves to `"true"`; a bare number is a valid operand

Joining values

The last column is what the same expression derives as a cache key (`private`, `public`, a vary `value`).

Expression	As a value	As a truth	As a cache key
`auth:idp.org.id + ':' + auth:idp.role`	`"o7:editor"`	match	`"2:o7:6:editor"`
`auth:idp.org.id + ':' + auth:idp.missing`	`"o7:"`	match	–

Expression	As a value	As a truth	As a cache key
`header:x-absent + header:x-org`	`"acme"`	match	–
`'tenant-' + auth:idp.org.id`	`"tenant-o7"`	match	`"tenant-o7"`
`auth:idp.role && auth:idp.tier`	`"editorpro"`	match	`"6:editor3:pro"`
`auth:idp.missing && auth:idp.tier`	–	no	–
`auth:idp.missing || auth:idp.tier`	`"pro"`	match	`"pro"`
`auth:idp.missing || header:x-absent`	–	no	–

> ⚠ **`+` and a missing part, outside a cache key**
>
> As a plain value, `+` skips what did not resolve, so the separator survives on its own. For a caller carrying none of the claims, `auth:idp.org.id + ':' + auth:idp.role` resolves to `":"`, a value that resolved. In a cache key the same expression resolves to nothing instead, so it no longer puts every such caller in one bucket.

Comparisons

Expression	As a value	As a truth
`auth:idp.role == 'editor'`	`"true"`	match
`auth:idp.role == 'admin'`	`"false"`	no
`auth:idp.missing == 'admin'`	–	undecidable
`auth:idp.missing != 'admin'`	–	undecidable
`header:x-count > '40'`	`"true"`	match
`header:x-count > '50'`	`"false"`	no
`header:x-org > '40'`	–	undecidable
`header:x-version >= '2.9.0'`	–	undecidable

The last two are the reason ordering is numeric and only numeric: `acme` and `2.10.0` are not numbers, so the comparison has no answer rather than a lexical one. Compare versions with `semver_gte` and its siblings.

Note the second row. A comparison that is *false* still resolved, to the string `"false"`. That is why a value expression's output cannot be read as a truth: as a value it is there, as a truth it is a no.

Functions

Expression	As a value	As a truth
`lower(auth:idp.role)`	`"editor"`	match
`upper(header:x-org)`	`"ACME"`	match
`len(header:x-org)`	`"4"`	match
`len(header:x-absent)`	–	undecidable
`starts_with(header:user-agent, 'gp-cli/')`	`"true"`	match
`starts_with(header:x-absent, 'gp-cli/')`	–	undecidable
`ends_with(header:x-org, 'me')`	`"true"`	match
`contains(header:user-agent, 'cli')`	`"true"`	match
`split(header:user-agent, '/', 1)`	`"0.8.0"`	match
`split(header:user-agent, '/', 9)`	–	undecidable
`in(auth:idp.role, ['editor', 'admin'])`	`"true"`	match
`in(auth:idp.role, ['admin'])`	`"false"`	no
`in(auth:idp.missing, ['admin'])`	–	undecidable
`semver_gte(header:x-version, '2.9.0')`	`"true"`	match
`semver_lt(header:x-version, '2.9.0')`	`"false"`	no
`semver_gte(header:x-absent, '2.9.0')`	–	undecidable
`semver_gte('latest', '2.9.0')`	–	undecidable
`pseudonymize(auth:idp.sub)`	`"azOiNEJTZ9KbWv1yQsvn34ZfGCsHSeYbH73G4fLf32Y"` (under an example key)	match

Expression	As a value	As a truth
`pseudonymize(auth:idp.missing)`	–	undecidable

`semver_gte(header:x-version, '2.9.0')` is a match where `header:x-version ⩾ '2.9.0'` is undecidable, on the same header and the same request. That is the whole reason the `semver_*` family exists.

Negation

Expression	As a value	As a truth
`!auth:idp.role`	`"true"`	no
`!auth:idp.missing`	`"true"`	match
`!header:x-blank`	`"true"`	match
`!(auth:idp.missing == 'admin')`	`"true"`	undecidable
`auth:idp.tier && !(auth:idp.tier == 'free')`	`"protrue"`	match
`!auth:idp.tier || !(auth:idp.tier == 'free')`	`"true"`	match

As a cache key, the fifth row derives `"3:pro4:true"`; every other row derives what it resolves to as a value.

> ⚠ **`!` in a setting that wants a value**
>
> Read as a value, `!` yields the string `"true"` for **anything that is not the string `"true"`**. The first row is the trap: `!auth:idp.role` resolves to `"true"` for a caller whose role is `editor`, and would resolve to `"true"` for every other role as well. As a `public` or `vary[].value` that is one bucket for everyone. Negation belongs in the settings that ask for a truth.

The same expressions, for a caller who sent none of it

Every source resolves to nothing and is a no, as you would expect. What is worth looking at is where that does **not** simply propagate:

Expression	As a value	As a truth
`auth:idp.org.id + ':' + auth:idp.role`	`":"`	match
`'tenant-' + auth:idp.org.id`	`"tenant-"`	match
`auth:idp.role == 'editor'`	–	undecidable
`!(auth:idp.missing == 'admin')`	`"true"`	undecidable
`!auth:idp.role`	`"true"`	match
`auth:idp.tier && !(auth:idp.tier == 'free')`	–	no

Expression	As a value	As a truth
`!auth:idp.tier || !(auth:idp.tier == 'free')`	`"true"`	match

The first two rows are the `+` trap again, now with nothing in front of the separator at all. As a cache key both resolve to nothing, so the trap is closed where it would share an entry. The last two are the guarded and the admitting spelling doing exactly what they are for: one excludes this caller, the other includes them, and neither leaves the answer to the setting.

Worked examples

One entry per person, which is what `private` normally wants:

```
[cache.policy]
private = "auth:idp.sub || cookie:session_id"
```

One shared entry per customer, with the customer taken from a verified claim:

```
[cache.policy]
public = "lower(auth:idp.org.id)"
```

> ⚠ **A header fallback is a header the caller chooses**
>
> `lower(auth:idp.org.id || header:x-customer-id)` looks like a harmless fallback. It is not: `header:` reads whatever the client sent, so a caller without a token can name any customer and read that customer's shared entries. Use a header fallback only when something in front of GraphPilot removes or overwrites that header on every request.

A composite partition, where both parts must be present or the expression resolves to nothing:

```
[cache.policy]
public = "auth:idp.org.id && '-' && auth:idp.locale"
```

Both claims are written with their length (`2:o7-2:de` for org `o7` and locale `de`), so no pair of claims can derive another pair's key. In a cache key `+` behaves the same way here: the expression resolves to nothing unless both claims are present.

A vary header your origin can partition on without learning the value:

```
[[cache.policy.vary]]
name           = "x-segment"
value          = "auth:idp.plan"
pass_to_origin = "anonymized"
```

A preview bypass for editors only:

```
[cache.policy]
bypass = "auth:cms.role == 'editor'"
```

See identifying callers for what each setting then does with the value.

REFERENCE

HTTP headers

GraphPilot reserves the `gp-` prefix for headers it sets itself. The namespace is protected in both directions: a client cannot send a `gp-` header (every one is dropped by prefix), and your origin cannot send one of the headers GraphPilot sets (each is removed by name). For reading these headers when something looks wrong, see troubleshooting.

Headers your client receives

GraphPilot sets ten response headers. They are the only `gp-` names GraphPilot itself puts on a response.

Header	Value	Controlled by	When it appears
`gp-cache`	`HIT`, `MISS`, `PASS` or `STALE`. See cache statuses.	`response.cache_headers` (default on)	Every response that went through the cache, including a `PASS`. Absent on a request refused before the cache (a `401` from authentication, for example), and on every response the cache would otherwise have looked up while the service runs in lean mode. In lean mode, a request that bypasses the cache still reports `PASS` with its reason.
`gp-cache-reason`	A stable code: `CACHE_PASS_*` or `CACHE_SKIPPED_*`. See codes.	`response.cache_headers`	On a `PASS`. Absent on a `HIT`, a `STALE`, and a `MISS` (which always means the response was stored).

Header	Value	Controlled by	When it appears
`gp-cache-hits`	How many times the copy on the cache server that answered has been served. Approximate: the entry can be stored on several servers, each counting on its own, so the value can go down between two reads. For hits per entry, use Insights.	`response.cache_headers`	On a `HIT` or `STALE`.
`gp-cache-age`	Seconds since the entry was written.	`response.cache_headers`	On a `HIT` or `STALE`.
`gp-cache-max-age`	The freshness lifetime (TTL) the entry was stored with, in seconds.	`response.cache_headers`	On a `HIT` or `STALE`.
`gp-cache-remaining`	Seconds of freshness left.	`response.cache_headers`	On a `HIT` or `STALE`.
`gp-origin-retries`	The number of repeats. See origin retries.	`[origin.retry]`; not affected by any `[response]` switch	When the origin request had to be repeated at least once.
`gp-surrogate-key`	The keys, space separated, whole keys only: the cache key (`ck:`) and root field keys (`q:`, `m:`) first, entity keys last.	`response.surrogate_key_header` (default off), `response.surrogate_key_header_max_bytes` (default `8192`)	When switched on and the response carries surrogate keys that fit the size limit.
`gp-surrogate-key-truncated`	How many keys `gp-surrogate-key` left out. If no key fits at all, this header is sent without `gp-surrogate-key`.	`response.surrogate_key_header`	When switched on and some keys did not fit.

Header	Value	Controlled by	When it appears
`gp-request-id`	The id GraphPilot recorded this request under, shown as the **Request ID** column in the portal's Insights and as **Entry ID** in the request's **Overview** tab. Minted per request, never stored with a cache entry.	`response.request_id_header` (default off)	When switched on, on every response.

Other headers GraphPilot writes or removes

Header	What GraphPilot does
`Cache-Control`	Written for the client when `cache.policy.downstream` is `if_cacheable` or `always`. Otherwise your origin's header passes through. A response GraphPilot produces itself (a rejection) carries `Cache-Control: no-store`.
`Allow`	Written on a `405` (`METHOD_NOT_ALLOWED`), naming the method the service accepts for that operation.
`Content-Encoding`	The edge compresses the response for the client according to the client's `Accept-Encoding`, on a hit exactly as on a miss.
`x-cache`, `x-cache-hits`	Removed. The underlying edge platform (Fastly) adds them to describe GraphPilot's own fetch from your origin, so they say nothing about the GraphPilot cache and could contradict `gp-cache`.
`surrogate-key`	Removed. GraphPilot has read your origin's `Surrogate-Key` header and its keys are part of `gp-surrogate-key`. See the Surrogate-Key header.

Reserved names on an origin response

Your origin cannot set the `gp-` headers above, or the ones GraphPilot uses between its edge locations. GraphPilot removes them, by name, from every origin response before anything else reads it, so an origin cannot make its own behavior look like GraphPilot's or get one of them stored in a cache entry. Other `gp-` names your origin sends are left alone.

Switching the headers on and off

Four keys under `[response]` decide whether and how much a client sees of these headers.

Key	Default	Controls
`response.cache_headers`	`true`	The whole cache status family: `gp-cache`, `gp-cache-reason`, `gp-cache-hits`, `gp-cache-age`, `gp-cache-max-age` and `gp-cache-remaining`.
`response.surrogate_key_header`	`false`	`gp-surrogate-key` and `gp-surrogate-key-truncated`.
`response.surrogate_key_header_max_bytes`	`8192`	How long the value of `gp-surrogate-key` may get. See size limit.
`response.request_id_header`	`false`	`gp-request-id`. See request ids.

Fact	Detail
Independent of each other	Each switch changes only what leaves GraphPilot.
The cache is unaffected	A response is stored, served and purged the same way whether a client sees the headers or not. Surrogate keys are still written to the entry, so purging by key works with `surrogate_key_header = false`.
Telemetry is unaffected	Request telemetry records the cache status, the reason code and the keys either way.
Applied on delivery	Like `[response.headers]`, the switches also govern a hit served from an entry written before you changed them.
`gp-origin-retries`	Not affected by any switch.
The only way to withhold a `gp-` header	`[response.headers]` cannot hide one: the namespace is protected, and naming one there refuses the deployment.

Size limit

`gp-surrogate-key` stops at `response.surrogate_key_header_max_bytes`, 8,192 bytes by default.

Rule	Detail
What it measures	The value of `gp-surrogate-key` alone, not all headers of the response together. The default is half of the 16 KiB Node.js accepts for all headers of a response, so the other headers still fit.
Whole keys only	A key is never cut in half.
Order	The cache key (`ck:`), then the root field keys (`q:`, `m:`), then type keys, then static keys, keys from your origin's `Surrogate-Key` header and empty-list keys, and entity keys last. When the list is cut, the entity keys go first.
Keys left out	Counted in `gp-surrogate-key-truncated`. Without that header, the list is complete. If not even one key fits, `gp-surrogate-key` is left off and only `gp-surrogate-key-truncated` is sent.
One header	The keys are never split across several header lines (RFC 9110, section 5.3).

Rule	Detail
Accepted values	`1` to `128000`. The upper bound is Fastly's limit for a single response header; past it the response would fail with a `503`, so a larger value refuses the deployment.
The entry is unaffected	The entry is tagged with every key, and a purge by any of them works whether it appeared in the header or not.

Headers a client can send

Everything a client sends reaches your origin unless a row below or your `[origin.headers]` filter says otherwise.

Header	What GraphPilot does with it	Reaches your origin
`gp-client-name`, `gp-client-version`	Kept for client profiles: a policy expression reads them as `header:gp-client-name` and `header:gp-client-version`. Removed from the request before it is forwarded.	No
`GP-Correlation-ID`, or the name set in `telemetry.correlation_header`	Its value is recorded on the request's telemetry row and on every log line for the request. Nothing is generated when it is absent. See correlating logs.	Yes, unchanged, under the default name too. When a client sends it more than once, only the first value is forwarded, the one telemetry records. The value is whatever the client sent and is not validated, so treat it as untrusted input even though the default name is in the `gp-` namespace. To keep it out of that namespace, set `telemetry.correlation_header` to a name such as `x-correlation-id`.
Any other `gp-` header	Dropped on arrival and logged as `INGRESS_RESERVED_HEADER_DROPPED`.	No
`Cache-Control`	Its `no-store`, `no-cache` and `max-age=0` directives are read under `[cache.request_directives]`.	Yes
`Authorization`, or the header an `[auth.providers.<name>]` block reads	Verified by that provider; its claims are available as `auth:<name>.<claim>`. A credential makes the request credentialed for caching. A `Bearer` prefix is stripped before verification.	Yes
`Cookie`	A cookie not on `cache.cookies.allow` makes the request credentialed.	Yes, minus `[origin.cookies]`

Header	What GraphPilot does with it	Reaches your origin
`Origin`, `Access-Control-Request-Method`, `Access-Control-Request-Headers`	On a cached CORS preflight, part of the entry's key. See CORS.	Yes
A header a `[[cache.policy.vary]]` entry names	Removed and logged as `INGRESS_CLAIMED_HEADER_DROPPED`; the entry writes its own derived value under that name.	Only as `pass_to_origin` allows

Headers dropped on the way in

Every request header starting with `gp-` is removed when the request arrives, before any setting or policy expression reads it, and each one is logged as `INGRESS_RESERVED_HEADER_DROPPED` with its name. `gp-client-name` and `gp-client-version` are read first, as the table above says, and removed all the same. The correlation header is the one exception: it is set aside before the removal and written back afterwards, so it reaches your origin and is not logged as dropped. A name GraphPilot acts on itself (`gp-signature`, `gp-timestamp`, `gp-client-ip`, the client names, or any other name GraphPilot uses internally) is never excepted, even when configured as the correlation header.

The client's own `Cache-Control`

What the directives a client sends on its own request do (RFC 9111, section 5.2.1). Each is a switch under `[cache.request_directives]`. A hard reload in a browser sends `no-cache`.

Request directive	Key	Default	When honored
`no-store`	`no_store`	`"honor"`	The cache is still read, so the request can be answered from an entry another request stored. Only the response to this request is not written: `CACHE_SKIPPED_REQUEST_NO_STORE`.
`no-cache`, or `max-age=0`	`no_cache`	`"ignore"`	The request goes straight to your origin with no lookup, and its response is not written back, the same path a bypass takes: `PASS`, `CACHE_PASS_REQUEST_NO_CACHE`.

Default	Why
`no_cache = "ignore"`	Any caller can send it. Honored, it lets a client put every one of its requests past the cache and onto your origin. Even when honored, the response is not written back, so it never overwrites the entry other clients read.
`no_store = "honor"`	The lookup still happens, so it never causes an origin request that would not have happened anyway; at most one entry is not written.

A value other than `"honor"` or `"ignore"`, or an unknown key in the section, is refused when the configuration is read.

Origin headers GraphPilot reads

Header on your origin's response	What GraphPilot does
`Cache-Control`	The only header that decides a lifetime. Which directives are read, and in which order, is on lifetime from the origin header. `stale-if-error` is parsed and not acted on.
`Surrogate-Key`	Keys to tag the entry with; on a response to a write, keys to purge. See the Surrogate-Key header. Removed before the response reaches the client.
`Vary`	Every request header it names becomes part of the entry's key. `Vary: *` is never stored (`CACHE_SKIPPED_VARY_WILDCARD`). See your origin's `Vary` header.
`Set-Cookie` , `Set-Cookie2`	A cookie not on `cache.cookies.allow` refuses storing (`CACHE_SKIPPED_SET_COOKIE`).
`WWW-Authenticate` , `Proxy-Authenticate` , `Authentication-Info` , `Proxy-Authentication-Info` , `Clear-Site-Data` , `Set-Login`	Refuse storing (`CACHE_SKIPPED_AUTHENTICATION_HEADER`), unless named in `cache.headers.allow` .
`Content-Encoding`	A compressed body is decoded first (`gzip` , `deflate` , `br` , `zstd`). See compression.
`Surrogate-Control`	Ignored.
`Expires`	Ignored.
`Age` , `Date` , `ETag` , `Last-Modified`	Not consulted. An entry's age comes from the cache itself.
The `gp-` headers GraphPilot sets	Removed. See reserved names.

Which of these refuse a response whatever its `Cache-Control` says is on when a response is cached.

Request headers your origin receives

GraphPilot changes a few headers on the request before forwarding it. Everything else arrives as the client sent it, minus the forwarding headers a client sent and what your `[origin.headers]` and `[origin.cookies]` filters remove.

Header	What your origin receives	Controlled by
`Host`	The hostname of your service's origin URL, not the hostname the client called. The same name is sent for TLS (SNI) and checked against the origin's certificate.	The origin URL. See origin connection.
`gp-timestamp`	The Unix time, in seconds, at which GraphPilot signed the request.	Always sent. See request signing.

Header	What your origin receives	Controlled by
`gp-signature`	`v1=` followed by the Base64 HMAC-SHA256 signature over the method, path and query, timestamp and body hash.	Always sent. A request GraphPilot cannot sign is not forwarded.
`GP-Correlation-ID`, or the name set in `telemetry.correlation_header`	The value the client sent, unchanged and not validated, and only the first one when it was sent more than once. It is not covered by `gp-signature`, so treat it as untrusted input even under the default `gp-` name. Nothing is sent when the client sent none.	`telemetry.correlation_header`; an `[origin.headers]` filter can remove it.
`Accept-Encoding`	Always `identity`, replacing what the client sent. GraphPilot reads the body to derive surrogate keys and strip injected fields, and compresses toward the client itself. An origin that compresses anyway is decoded (`ORIGIN_RESPONSE_DECODED`) or answered with `502`. See codes.	Not configurable. An `[origin.headers] only` list does not remove it.
The name you choose	The GraphPilot request id, the same value as `gp-request-id`. A client-sent header of the same name is replaced.	`origin.request_id_header` (default unset, so nothing is sent). A `gp-` name refuses the deployment.
`gp-client-ip`	The client's IP address, IPv4 or IPv6, as the edge saw the connection; on a shielded request, the address the first edge location saw, never a GraphPilot address. Not sent when the address is unknown. A client cannot set it, since every `gp-` header a client sends is dropped on arrival. Like the other headers it is not part of `gp-signature`, so trust it on a request whose signature verifies.	`origin.forward_client_ip`, default `false`, so nothing is sent. An `[origin.headers] only` list does not remove it. `telemetry.client_ip` does not change it either, but it decides whether a `[telemetry.headers]` rule may record it: only under `client_ip = "full"`.
Every `X-Forwarded-*` header, `Forwarded`, `X-Real-IP` and the other forwarding headers	Nothing by default: the values a client sent are removed. With `"pass"`, the values exactly as the client sent them. GraphPilot never writes these headers itself.	`origin.forwarded_headers`, default `"strip"`.
A `[[cache.policy.vary]]` header	Nothing by default. With `pass_to_origin = "anonymized"`, a keyed digest (`v1=` plus base64url); with `"plain"`, the derived value.	`pass_to_origin`

Header	What your origin receives	Controlled by
`Content-Type`, `Content-Length`	`Content-Length` is restated for the body actually sent. A `GET` whose operation GraphPilot rewrote (a persisted query hash resolved to its document) is sent as `POST`, or `QUERY` under `method_rewrite = "post_to_query"`, with a JSON body and `Content-Type: application/json`.	`origin.method_rewrite`

Nothing else in the `gp-` namespace reaches your origin: apart from the correlation header above, GraphPilot strips the whole namespace before it signs, then adds `gp-timestamp` and `gp-signature`, and `gp-client-ip` only when you turn on `origin.forward_client_ip`. A fresh cache hit is answered at the edge, so no request, and no request id, reaches your origin for it.

Forwarding headers

Forwarding headers tell a server where a request came from and which URL it asked for. A proxy in front of a server normally writes them. GraphPilot writes none of them, so on a request that reaches the edge every value in them is one the client chose.

What counts as a forwarding header

- **Every header whose name starts with** `X-Forwarded-`, in any case: `X-Forwarded-For`, `X-Forwarded-Host`, `X-Forwarded-Proto`, `X-Forwarded-Port`, `X-Forwarded-Prefix`, `X-Forwarded-Client-Cert`, and any other name with that prefix. `X-Forwarded` without the trailing dash is an ordinary header.
- **These ten names**, matched whole and in any case: `Forwarded`, `X-Real-IP`, `X-Client-IP`, `Client-IP`, `True-Client-IP`, `CF-Connecting-IP`, `Fastly-Client-IP`, `X-Original-URL`, `X-Rewrite-URL`, `X-Original-Host`.

Why they are removed

An origin that reads them gets one of two problems:

- **A spoofed address.** A framework configured to trust a proxy (Express `trust proxy`, Rails, Symfony, nginx `real_ip`) takes the client's address from `X-Forwarded-For` or `Forwarded`. A rate limit or an IP allowlist built on it uses an address the client picked.
- **A poisoned cache entry.** An origin that builds absolute URLs from `X-Forwarded-Host` or `X-Forwarded-Proto` answers with the client's URLs. These headers are not part of the cache key, so that answer is stored and served to every later caller.

When they are removed

With `origin.forwarded_headers` at its default, `"strip"`, every forwarding header is removed right before the request is sent to your origin, after your `[origin.headers]` filter has run. Only the request to your origin changes:

- GraphPilot itself reads the request as the client sent it. A policy expression such as `header:x-forwarded-for` still resolves, a `[[cache.policy.vary]]` entry can still derive a value from one, and

`telemetry.headers.request` records the headers the client sent. `telemetry.headers.origin_request` records what your origin received, so the forwarding headers are absent there.

- On a shielded service, the edge location passes the headers on to the shield, and the shield removes them before it calls your origin.
- A cache hit never reaches your origin, so nothing is removed there.

Removing further headers

The strip covers forwarding headers only. To keep other headers from your origin, list them in `[origin.headers]`. A trailing `*` matches a prefix:

```
[origin.headers]
hide = ["x-internal-debug", "x-envoy-*"]
```

An `[origin.headers]` `only` list that keeps a forwarding header while the strip is on **refuses the deployment**, because it asks to keep a header the strip removes. That applies to an entry naming one (`x-forwarded-for`, `True-Client-IP`) and to the pattern `x-forwarded-*`. A broader pattern such as `x-*` is accepted: it keeps your other `x-` headers, and the forwarding headers among them are still removed.

Passing them through

With `origin.forwarded_headers = "pass"`, the headers reach your origin exactly as the client sent them, and both problems above are yours to handle: do not treat the values as facts about the connection, and put any header your response depends on into the cache key with a `[[cache.policy.vary]]` entry of the same name:

```
[origin]
forwarded_headers = "pass"

[[cache.policy.vary]]
name = "x-forwarded-host"
value = "header:x-forwarded-host"
pass_to_origin = "plain"
```

The client's address

Read the client's address from `gp-client-ip`, which no client can set. Turn on `origin.forward_client_ip` and trust the header on a request whose signature verifies. It is independent of `origin.forwarded_headers`: it is sent in both modes.

Request ids

Every request is recorded under an id, shown as the **Request ID** column in Insights and as **Entry ID** in the request's **Overview** tab. Two independent settings decide who else learns it:

Setting	Who sees the id	Rules
`response.request_id_header = true`	The caller, as `gp-request-id`	Off by default.
`origin.request_id_header = "<name>"`	Your origin, under the name you choose, commonly `x-request-id`	A `gp-` name, or one that is not a valid header name, refuses the deployment. An `[origin.headers]` `only` list does not remove it. A client-sent header of the same name is replaced. Do not reuse the name of your `telemetry.correlation_header`, or the client's correlation id is replaced on its way to your origin.

An id GraphPilot reported that your origin has no record of is the normal case for a fresh cache hit, not a lost request. `gp-cache` tells the two apart. Following one request across both logs is on correlating logs.

Reserved path

Path	What GraphPilot does
`POST /__gp/purge`	Reserved: answered by GraphPilot itself, on every hostname of the service, and never forwarded to your origin. A request GraphPilot did not send is answered with `401` and `{"error":"unauthorized"}`, and counts as one request unit. To purge, use the API or CLI. Only `POST` to exactly this path is reserved; other methods and other paths under `/__gp/` are forwarded as usual.

CORS

CORS preflights, and the methods you never cache

A browser sends an `OPTIONS` preflight before a cross-origin `POST`. `cache.policy.uncacheable_methods` holds `OPTIONS` by default, so preflights skip the cache and reach your origin (`CACHE_PASS_UNCACHEABLE_METHOD`).

With `uncacheable_methods = []`	Detail
A preflight becomes a cache candidate	It still needs a response policy that permits storing it. A preflight answered without a `Cache-Control` is not stored.
Keyed per question	GraphPilot adds `Origin`, `Access-Control-Request-Method` and `Access-Control-Request-Headers` to the entry's vary rule, whatever your origin's own `Vary` says, so one site is never handed the answer computed for another. Every browser on one site asking the same question reads the same entry.
Revocation	A cached preflight keeps saying yes for its freshness lifetime. Revoking a site's access means purging or waiting it out, the same question as authenticated requests.
Little to win	When your origin's `Access-Control-Max-Age` is already generous, browsers cache the preflight themselves and the edge sees few.

With `uncacheable_methods = []`	Detail
Method names	Written in capitals. A name that is not a valid HTTP method is refused when the configuration is read.

> ⚠ **A preflight is one answer per question**
>
> A preflight response depends on the caller's `Origin` and on what it asks for. GraphPilot keys it on all three request headers so that an entry never grants one site the cross-origin permissions you gave another.

Edge rejections

A response GraphPilot produces itself carries none of the CORS headers your origin would have sent, so a browser blocks it and your app sees a network error instead of the code. That covers every response GraphPilot writes instead of your origin's: the rejections in codes (such as a `400`, `401`, `403`, `404`, `405`, `413` or `415`), the `500`, `502` and `504` GraphPilot answers when it cannot get a response from your origin, and the persisted query answers `PERSISTED_QUERY_NOT_FOUND` and `PERSISTED_QUERY_NOT_SUPPORTED`. List the origins that may read them in `[response.cors]`:

gpilot.toml

```
[response.cors]
allow_origins     = ["https://app.example.com"]
allow_credentials = false
```

When a request's `Origin` matches a listed origin (case-insensitively), a response GraphPilot produced itself carries:

Header	Value
`Access-Control-Allow-Origin`	The request's own `Origin`, or `*` when the list is `["*"]`
`Vary`	`Origin`
`Access-Control-Allow-Credentials`	`true`, only with `allow_credentials = true`
`Access-Control-Expose-Headers`	`gp-request-id`, only with `response.request_id_header` on

A request without an `Origin`, or from an origin not on the list, gets none of them. A response from your origin, a cache hit included, carries exactly the CORS headers your origin sent, and a preflight (`OPTIONS`) still goes to your origin.

Related

- Troubleshooting: the checks that read these headers.
- Cache statuses: what `HIT`, `MISS`, `PASS` and `STALE` mean.

- Codes: every value `gp-cache-reason` can carry.
- Request signing: verifying `gp-signature` on your origin.

REFERENCE

Error and log codes

GraphPilot reports what it did with a stable code rather than a sentence. This page lists the codes of the edge, the telemetry values recorded beside them, the GraphPilot API's error codes and the CLI's exit codes.

> ⚠ **Two different FORBIDDEN codes**
>
> The edge's `FORBIDDEN` is a `403` on a request to your service: the operation is not on its allowlist. The GraphPilot API's `FORBIDDEN` is an error in a `200` GraphQL response: the credential lacks a permission. They share a name and nothing else.

Edge codes

An edge code appears in three places, so a support thread can start from any of them:

- on the response, as `gp-cache-reason` for the cache codes, or in the error body for the codes that reject a request,
- in your **logs**: a decision or refusal is recorded on the request itself and listed under **Edge codes** at the end of the request's **Overview** tab in Insights, and a warning or error is one of the log lines, read through the API's `Service.logs` (both stored under the same rules as telemetry, below),
- in your **request telemetry**, which the portal's Insights view reads, stored under the rules on what Insights needs.

> ⓘ **Note**
>
> Treat this list as open. New codes are added as the product grows, so anything consuming them must pass an unknown code through and fall back to this reference rather than failing on it. A code, once published, does not change meaning.

Reading a code

Some codes **refine** another one. `REQUEST_BODY_TOO_LARGE` refines `PAYLOAD_TOO_LARGE`: one request can log both, as a cause and its outcome, and they share a `request_id`. A view that wants a single line per event can drop the parent and keep the child.

A refining code is **for your logs, not for the caller**. The client is told the parent, because its situation and its fix are the same whichever of the two ceilings answered, and which one did is something only you can see. Do not switch on a refining code in client code: it never arrives there. The tables below mark each one **log only**.

Where a row below says what a code's **detail** names (a header, a cookie, a client), that detail is shown beside the code: under **Edge codes** in the request's **Overview** tab for a decision or refusal, or on the log line for a warning or error.

The request was rejected

These end the request with the status shown. The caller receives the code in the body: for a GraphQL request, as a GraphQL error carrying it in `extensions.code`; for any other request, as an `application/problem+json` body carrying it in `title`. Rows marked **log only** appear in your logs and telemetry, while the caller is told the code they refine.

Your origin is not called for any of them, except `BAD_GATEWAY`, `GATEWAY_TIMEOUT` and the codes that refine them, which are produced after the request was sent on, to your origin or, with shielding, to the shield, and no usable answer came back.

Code	Status	Refines	Meaning
`UNAUTHENTICATED`	`401`		A credential GraphPilot could not accept.
`AUTH_INVALID_TOKEN`	`401`	`UNAUTHENTICATED`	The signature did not verify, the token expired, it carries no `exp` and the provider does not set `allow_missing_exp`, or a required claim did not match.
`AUTH_UNKNOWN_KEY_ID`	`401`	`UNAUTHENTICATED`	The token names a key GraphPilot does not have, and refreshing did not produce it.
`AUTH_KEY_MATERIAL_UNAVAILABLE`	`401`	`UNAUTHENTICATED`	No key material at all for this provider, affecting every request.
`FORBIDDEN`	`403`		The operation is not on this service's allowlist.
`INTROSPECTION_DISABLED`	`403`		The service does not answer schema introspection. Does not apply to operations sent over a subscription socket.
`METHOD_NOT_ALLOWED`	`405`		The operation may not be run with the method the request used. The response carries an `Allow` header naming the method to use.
`PAYLOAD_TOO_LARGE`	`413`		The request body is over a limit this service sets, checked before the body is parsed. This is what the caller is told for either of the two rows below; the message carries no numbers, so a caller learns only that it sent too much.
`REQUEST_BODY_TOO_LARGE`	`413`	`PAYLOAD_TOO_LARGE`	**Log only.** The body was over `request.max_body_bytes`, the service-wide ceiling, checked before the client is even identified.
`CLIENT_BODY_TOO_LARGE`	`413`	`PAYLOAD_TOO_LARGE`	**Log only.** The body cleared the service-wide ceiling but was over the `max_body` set on the client profile that matched it. Its detail names the client.

Code	Status	Refines	Meaning
`UNSUPPORTED_GRAPHQL_BODY`	`415`		The body sent to a GraphQL path is not a single operation GraphPilot can check (a JSON batch, a multipart upload, text, no content type, or JSON that names no operation), and the service does not forward that kind unchecked. See bodies on a GraphQL path.
`INGRESS_UNSUPPORTED_GRAPHQL_BODY`	`415`	`UNSUPPORTED_GRAPHQL_BODY`	**Log only.** Names the `kind` of body that was refused (`batch`, `multipart` or `other`); listing that kind in `request.forward_unchecked_bodies` forwards such bodies to your origin unchecked.
`INVALID_GRAPHQL_REQUEST`	`400`		The request on a GraphQL path carries an operation GraphPilot cannot read unambiguously, so it is refused rather than forwarded unchecked: your origin might read it differently, and none of GraphPilot's checks would then have seen what it runs. The message says what was wrong with the request's form.
`INGRESS_UNREADABLE_GRAPHQL_REQUEST`	`400`	`INVALID_GRAPHQL_REQUEST`	**Log only.** `reason` says what could not be read: `duplicate_key` (the JSON body repeats a key within one object), `too_deep` (the JSON body nests more than 128 levels), `repeated_parameter` (a `GET` repeats `query`, `operationName`, `variables` or `extensions`), `unreadable_parameter` (a `GET`'s `variables` or `extensions` is not JSON, repeats a key, or nests too deep), `query_string_override` (a `POST`, or a `GET` with a body, carries `query`, `variables` or `extensions` in its URL, or an `operationName` that differs from the body's), or `coded_body` (the body carries a `Content-Encoding` such as `gzip`, `deflate`, `br` or `zstd`: a body on a GraphQL path must be sent uncompressed; `identity` counts as uncompressed). Refused whatever `request.forward_unchecked_bodies` lists.
`NOT_A_GRAPHQL_PATH`	`404`		The request carries a GraphQL operation on a path that is not one of this service's GraphQL paths (`request.graphql_paths`), where the GraphQL checks do not run, so it was refused rather than forwarded unchecked. Sent while at least one of those checks is in force, which by default it is. The message names the GraphQL paths. A `query` that is not a GraphQL document, such as a search term, is never refused. See GraphQL on other paths.

Code	Status	Refines	Meaning
`INGRESS_OFF_PATH_GRAPHQL`	`404`	`NOT_A_GRAPHQL_PATH`	**Log only.** `form` says where the operation was found: `json_body` (a JSON body whose top-level `query` is a GraphQL document, or that carries `extensions.persistedQuery`), `document_body` (an `application/graphql` body), `form_body` (an `application/x-www-form-urlencoded` body) or `query_string` (the URL's `query` or `extensions` parameter). `protections` lists the checks the operation would have gone around: `introspection_block`, `trusted_documents`, `client_persisted_query_required`, `max_query_depth`, `max_document_bytes`, `max_selections`. Point the client at a GraphQL path, or add this path to `request.graphql_paths` if your origin serves GraphQL there.
`GRAPHQL_INVALID_QUERY`	`400`	`INVALID_GRAPHQL_REQUEST`	**Log only.** The GraphQL document could not be parsed. A leading byte-order mark is skipped first. A block string with a backslash directly before a quote is refused here too, because parsers disagree on where such a string ends.
`GRAPHQL_EMPTY_QUERY`	`400`	`INVALID_GRAPHQL_REQUEST`	**Log only.** The operation selects no field your deployed schema defines.
`GRAPHQL_AMBIGUOUS_OPERATION`	`400`	`INVALID_GRAPHQL_REQUEST`	**Log only.** The document defines several operations and the request named none, so none could be selected.
`GRAPHQL_OPERATION_NOT_FOUND`	`400`	`INVALID_GRAPHQL_REQUEST`	**Log only.** The request named an operation the document does not define.
`QUERY_DEPTH_LIMIT_EXCEEDED`	`400`		The operation nests deeper than this service allows. The message gives the depth and the limit. Also returned, whatever `request.max_query_depth` says and even with it set to `0`, for a document nested more than 128 levels deep, counted on its brackets before it is parsed or on its fields with fragments expanded: the ceiling GraphPilot keeps for itself so a document cannot exhaust its stack. Does not apply to operations sent over a subscription socket.
`DOCUMENT_SIZE_LIMIT_EXCEEDED`	`400`		The operation's document is longer than this service allows (`request.max_document_bytes`, or a lower client profile value). The message gives the length and the limit. See query document limits. Does not apply to operations sent over a subscription socket.
`DOCUMENT_SELECTION_LIMIT_EXCEEDED`	`400`		The operation makes more selections than this service allows (`request.max_selections`, or a lower client profile value), counted with fragment spreads expanded. The message gives the count and the limit. Does not apply to operations sent over a subscription socket.

Code	Status	Refines	Meaning
`RESERVED_FIELD_PREFIX`	`400`		The document uses an alias, or a field name without an alias, that starts with `origin.field_prefix` (`__gp_` by default). GraphPilot names the fields it adds with that prefix and removes everything carrying it from the response, so a key of your own there would disappear from the answer. Every operation and fragment in the document is checked, including ones the request does not run. The message names the key and the prefix; rename the alias. An empty `field_prefix` turns the check off. See selection set expansion.
`UNKNOWN_CLIENT`	`403`		The request matched no client profile this service declares, and `client.unknown` is `"reject"`.
`CLIENT_VERSION_UNSUPPORTED`	`403`		The client is declared, but the version it reported is below that profile's `min_version`. The required version is in the message.
`PERSISTED_QUERY_REQUIRED`	`403`		The client's profile sets `persisted_query = "required"` and the payload was not a hash on its own. A document is refused even when a hash travels beside it, which is what an APQ registration looks like. Does not apply to operations sent over a subscription socket.
`INTERNAL_SERVER_ERROR`	`500`		GraphPilot failed to process the request. The message says nothing about why.
`BAD_GATEWAY`	`502`		The origin could not be reached or did not answer usably.
`ORIGIN_RESPONSE_DECODED_TOO_LARGE`	`502`	`BAD_GATEWAY`	**Log only.** Your origin compressed its response although GraphPilot asked for an uncompressed one, and the body would decode to more than GraphPilot processes.
`ORIGIN_RESPONSE_ENCODING_CORRUPT`	`502`	`BAD_GATEWAY`	**Log only.** Your origin compressed its response although GraphPilot asked for an uncompressed one, and the body did not decode under the coding its `Content-Encoding` named.
`ORIGIN_RESPONSE_ENCODING_UNSUPPORTED`	`502`	`BAD_GATEWAY`	**Log only.** Your origin compressed its response with a coding GraphPilot cannot decode. It decodes `gzip`, `deflate`, `br` and `zstd`.
`SHIELD_RESPONSE_UNDECODABLE`	`502`	`BAD_GATEWAY`	**Log only.** With shielding, the shield answered this edge location with a compressed body although the edge asked for an uncompressed one, and the edge could not decode it. `reason` says why (`unsupported`, `corrupt` or `too_large`). The edge location does not then call your origin itself.

Code	Status	Refines	Meaning
`SHIELD_UNAVAILABLE_NO_FALLBACK`	`502`		**Log only.** The caller receives `BAD_GATEWAY`. With shielding, a mutation could not be sent through the shield, so this edge location answered `502` rather than send it to your origin itself, which could run it twice and would purge this location's cache before the shield's. `reason` says why, with the same values as `SHIELD_UNAVAILABLE_FALLBACK`. A request whose operation the edge cannot see (a persisted query hash without a document) is treated the same way when the failure came after the request may have left the edge location, since it may be a mutation. `shield_result` is `UNAVAILABLE`.
`GATEWAY_TIMEOUT`	`504`		With shielding, the request was sent to the shield and no usable response came back: the hop exceeded its time budget, or the connection dropped or the response was incomplete, oversized or malformed. Applies to queries and mutations alike. The edge location does not then call your origin itself, so your origin is never called twice for the same request. `shield_result` is `TIMEOUT`. See shielding and the origin connection.

A body sent to a GraphQL path (`request.graphql_paths`) is read as one operation: JSON carrying a `query` or a persisted query hash, or an `application/graphql` document, which is checked like any other operation. A JSON body that repeats a key or nests more than 128 levels deep, and a document that does not parse, are refused with `400` and `INVALID_GRAPHQL_REQUEST`. Any other body is refused with `415` and `UNSUPPORTED_GRAPHQL_BODY` before it reaches your origin, unless its kind is listed in `request.forward_unchecked_bodies`. See persisted queries. A GraphQL operation sent to any other path is refused with `404` and `NOT_A_GRAPHQL_PATH`; see GraphQL on other paths.

GraphPilot sends every origin request with `Accept-Encoding: identity`, because it has to read the response body to derive surrogate keys and strip the fields it injected. The edge compresses toward the client itself. An origin that compresses anyway costs a decode (`ORIGIN_RESPONSE_DECODED`, below) or, when decoding fails, a `502`. See response headers.

See authentication, limits and origin timeouts for what each of these is protecting.

The response was not cached

Every code here arrives on `gp-cache-reason` alongside `gp-cache: PASS`. `MISS` means the response was stored, so a response that was not stored never reports `MISS`, whatever the reason (see which status a refused store reports). The response is still delivered to the client in full; only storing was refused. One code in this group, `CACHE_SKIPPED_NO_ORIGIN_RESPONSE`, never reaches `gp-cache-reason`, because no response reaches the client on its path: it appears only in log lines and request telemetry. The order these are evaluated in is on when a response is cached.

Code	Meaning	What to do
`CACHE_SKIPPED_UNCACHEABLE_OPERATION`	The operation was a mutation or a subscription. Only a query is ever cached, whatever the origin's `Cache-Control` said.	Nothing. This is by design.
`CACHE_SKIPPED_NO_CACHE_POLICY`	Neither an origin `Cache-Control` header nor a schema cache policy applied.	Send `Cache-Control` from your origin, or annotate the schema with `@cacheControl`.
`CACHE_SKIPPED_REQUEST_NO_STORE`	The request's own `Cache-Control` said `no-store`, so its response was not cached. The cache was still read.	Nothing, unless the client should not send it. Set `cache.request_directives.no_store = "ignore"` to cache such responses anyway. See the client's own `Cache-Control`.
`CACHE_SKIPPED_ENTRY_NOT_REPLACED`	The cache already held an entry for this key that the request did not use, because it was older than the matching client profile's `max_age` or could not be read, and the entry was not replaced. An unreadable entry is normally overwritten with the origin's answer (reported as `MISS`); this code means the overwrite was not possible. The response was served from your origin without being stored, and the entry is unchanged.	Nothing for a client with a `max_age`: this is what the cap does.
`CACHE_SKIPPED_NO_ORIGIN_RESPONSE`	The cache had no usable entry, but no origin response came back to store: the origin request failed, or GraphPilot answered the request itself before reaching the origin. Nothing was cached. Log and telemetry only, never on `gp-cache-reason`.	Look at the request's other codes: an origin failure (`BAD_GATEWAY`, a timeout) or a refusal explains it.
`CACHE_SKIPPED_NO_MAX_AGE`	The deciding policy stated no freshness: the origin response carried neither `max-age` nor `s-maxage`, or the schema policy resolved no positive `maxAge`. Checked before the discriminators, so it is also what a `private` response with no `max-age` reports.	Add `max-age` or `s-maxage` at the origin, or a `maxAge` to every type the query selects.
`CACHE_SKIPPED_NO_STORE`	The origin response was marked `no-store`.	Nothing, unless the marking is wrong.
`CACHE_SKIPPED_NO_CACHE_DIRECTIVE`	The origin response was marked `no-cache`, which forbids reuse without revalidating with the origin first.	Use `max-age` to cache it, `no-store` not to.

Code	Meaning	What to do
`CACHE_SKIPPED_ERROR_STATUS`	Your origin answered with a `4xx` or `5xx` status that `cache.policy.store_error_statuses` does not list, so the response was not stored, whatever its `Cache-Control` said.	Nothing, unless the error is meant to be cached. To store one on purpose, such as a `404` for a resource that does not exist, list its status in `cache.policy.store_error_statuses`.
`CACHE_SKIPPED_UNPARSEABLE_GRAPHQL_RESPONSE`	The response to a GraphQL request was not JSON GraphPilot could parse, or was larger than it parses. It was delivered to the client unchanged but not stored, because neither GraphQL errors nor surrogate keys could be read from it.	Check that your origin answers GraphQL requests with a JSON body.
`CACHE_SKIPPED_GRAPHQL_ERRORS`	Your origin returned a GraphQL error. An errored response describes the moment it was produced rather than the data, and caching it would serve that failure to every later request for the same query until the entry expired.	Nothing, unless the error is unexpected. Fix it at the origin.
`CACHE_SKIPPED_PRIVATE_WITHOUT_DISCRIMINATOR`	The response is partitioned, by an origin `private` or by the schema's scope for the query (also when the origin header decided the lifetime), but at least one discriminator it is keyed on was not available, so it could not be filed. On the schema path its detail separates names the request did not carry from names nothing configures. Also reported when the request carried a credential and a `[[cache.policy.vary]]` rule that partitions the response resolved to nothing or to the empty string; its detail names those rules under `unresolved`.	For a private response, configure `cache.policy.private` and check that its expression resolves for this request. For a named scope, check that every name it lists has a `[[cache.policy.vary]]` entry whose expression resolves.
`CACHE_SKIPPED_UNRESOLVED_SELECTION`	The query selected a field, or a fragment on a type, that the deployed schema does not declare, at any depth, usually because your origin was deployed before the schema was. Who may share that field's value is unknown, so the response was not stored, whatever the origin's `Cache-Control` said.	Deploy the schema your origin runs. See deploy the schema before the origin.

Code	Meaning	What to do
`CACHE_SKIPPED_CREDENTIALED_REQUEST`	The request carried a credential (`Authorization`, a `Cookie` naming at least one cookie that is not on `cache.cookies.allow`, or the header an `[auth.providers.<name>]` block is configured to read), the origin never said the response may be held by a shared cache, and no discriminator identified the caller. Its detail names which header carried the credential, and for `cookie` also names the first blocking cookie, unless the header's value carries no readable cookie name at all.	Mark the response `public` or `s-maxage` if it is the same for everyone, configure a private policy, or add the cookie to `cache.cookies.allow` if it carries no session, as App Engine's `GAESA` does by default. See authenticated requests.
`CACHE_SKIPPED_SET_COOKIE`	The response carried `Set-Cookie` or `Set-Cookie2` setting at least one cookie that is not on `cache.cookies.allow`. A cache entry would re-assert that cookie on every hit until it expired. Its detail names the blocking cookie, not just the header.	Stop setting cookies on reads, mark the response `no-store`, or add the cookie to `cache.cookies.allow` if it carries no session. This is the most common cause of a service that caches nothing.
`CACHE_SKIPPED_AUTHENTICATION_HEADER`	The response carried a header stating something about one caller's session: an authentication challenge, `Clear-Site-Data`, or `Set-Login`.	Send it only on the responses that need it.
`CACHE_SKIPPED_VARY_WILDCARD`	The response carried `Vary: *`, so no cache entry could legitimately match a later request.	Name the headers you actually vary on.
`CACHE_SKIPPED_NO_SURROGATE_KEYS`	The response carried no surrogate keys.	Rare. Check that the request reached the GraphQL layer at all.
`CACHE_SKIPPED_SURROGATE_KEYS_TOO_LARGE`	The surrogate keys exceeded the 16,000-byte cache header limit.	Paginate the response. See how many keys fit on one response.
`CACHE_SKIPPED_SERIALIZATION_FAILED`	The response could not be encoded for storage.	Nothing. This is an internal fault, not a configuration problem.
`CACHE_SKIPPED_INSERT_FAILED`	The cache storage refused the write.	Nothing. This is an internal fault.
`CACHE_SKIPPED_SHIELD_ENTRY_EXHAUSTED`	With shielding, the response came from the shield, which had already held it for its whole max-age, so no lifetime was left for a copy at this edge location. An entry never lives longer in total than your origin allowed.	Nothing. This is by design.

Code	Meaning	What to do
`CACHE_SKIPPED_SHIELD_RESPONSE_NOT_STORABLE`	With shielding, the shield's response was served but not stored at this edge location: it came back `STALE` or `PASS`, or its metadata or age could not be read. A stale or uncacheable response is never copied to every edge location.	Look at the shield's own reason for why the shield did not store it: the portal shows it beside this code. When `shield_result` is `HIT` or `MISS`, `SHIELD_META_UNUSABLE` on the same request says what could not be read.

A response with a `4xx` or `5xx` status is not stored by default, even when its `Cache-Control` permits it, and reports `PASS` with `CACHE_SKIPPED_ERROR_STATUS`. A status listed in `cache.policy.store_error_statuses` is stored under the ordinary rules instead. See an error status is not stored unless you opt in.

The cache was not consulted

Reported alongside `gp-cache: PASS`, like the codes above. Most of these passes are decided before a lookup; `CACHE_PASS_READ_DISABLED` follows a lookup that failed. Either way the request went to your origin and its response was not stored.

Code	Meaning
`CACHE_PASS_BYPASS_POLICY`	The service's cache bypass policy matched this request.
`CACHE_PASS_UNCACHEABLE_METHOD`	The request method is listed in `cache.policy.uncacheable_methods`, which defaults to `OPTIONS`.
`CACHE_PASS_WEBSOCKET_UPGRADE`	A WebSocket upgrade is never cacheable and always reaches the origin.
`CACHE_PASS_MISSING_CACHE_KEY`	The request produced no cache key. Also for a GraphQL request whose body carries a document together with a persisted query hash that is not that document's hash: it is never looked up or stored, and runs as before.
`CACHE_PASS_READ_DISABLED`	The cache could not be read, so the request went to the origin.
`CACHE_PASS_CLIENT_CACHE_OFF`	The client profile that matched this request sets `cache = "off"`.
`CACHE_PASS_REQUEST_NO_CACHE`	The request's own `Cache-Control` said `no-cache` or `max-age=0`, and `cache.request_directives.no_cache` is `"honor"`. Off by default, because any caller can send it.

Something succeeded

These are not problems. They appear in logs and telemetry, not as a `gp-cache-reason`.

Code	Refines	Meaning
`CACHE_STORED`		The response was written to the cache.
`CACHE_STORED_PRIVATE`	`CACHE_STORED`	The request carried a credential and the origin never said the response may be held by a shared cache, so the entry was partitioned under the private cache policy rather than shared.
`CACHE_STORED_WITHOUT_STALE_WINDOW`	`CACHE_STORED`	The origin response carried `stale-while-revalidate` together with `must-revalidate` or `proxy-revalidate`. Those forbid serving it once stale, so the entry was stored for its max-age only and will not be served `STALE`. Its detail carries the dropped window as `stale_while_revalidate`.
`CACHE_POLICY_ORIGIN_HEADER`		Your origin sent a `Cache-Control` header, so it decided the response, outranking the schema's `@cacheControl` policy. `overrode_schema_policy` says whether there was a schema policy to outrank, and `cache_control` carries the header verbatim.
`CACHE_POLICY_ORIGIN_HEADER_IGNORED`		Your origin sent a `Cache-Control` that constrains neither storage nor freshness (a bare `private`, for example), so it decided nothing and the schema's `@cacheControl` policy governed instead. A `private` on such a header still partitions the entry. Turned off with `cache.ignore_origin_header_without_freshness = false`.
`CACHE_POLICY_SCHEMA`		The schema's `@cacheControl` policy decided the response. `max_age` is the lowest across every field the query selected, and `max_age_set_by` names the fields that set it (only those that declared the value rather than inherited it, capped and marked `and more` when there were further ones). Not restored on a cache hit.
`CACHE_PURGED`		Surrogate keys were purged from the cache. See automatic invalidation.
`ORIGIN_RETRIED`		A failed origin request was sent again, per the configured retry policy.
`SHIELD_UNAVAILABLE_FALLBACK`		With shielding, the request could not be sent through the shield, or what came back was not a shield response, so this edge location went to the origin itself. A query falls back on any of these failures. A mutation never falls back: it is answered `502` instead (`SHIELD_UNAVAILABLE_NO_FALLBACK`), and so is a request whose operation the edge cannot see once it may have left this location. A failure after the shield may have started on the request is a `504` (`GATEWAY_TIMEOUT`) instead, for every request.
`SHIELD_META_UNUSABLE`		With shielding, the shield's response was served to the client, but this edge could not read what it needs to store the response here (its metadata was missing, too large, or did not decode), so it was passed through without being cached at this location.

Code	Refines	Meaning
`SHIELD_PURGE_NOT_CARRIED`		With shielding, the shield purged a mutation's surrogate keys but could not tell this edge location which, so the edge did not purge them again on the way back. `reason` says why (`too_large`, `unencodable` or `undecodable`). This location's copy of those entries is then cleared only when the shield's purge reaches it.
`SHIELD_HOP_REJECTED`		A request arrived claiming to come from another edge location but did not verify as such, or arrived at a location other than the shield it names, so it is served as an ordinary request rather than as a hop from another edge location. When it came from another edge location, the shield records it as a hop entry rather than as a client request; like every request the shield runs, it counts at the shield toward request units.
`ORIGIN_RESPONSE_DECODED`		Your origin compressed its response although GraphPilot asked for an uncompressed one, so GraphPilot decoded it first. The response is keyed and cached as usual; `encoded_bytes` and `decoded_bytes` carry both sizes. Configure your origin to honor `Accept-Encoding: identity` to save the work.
`INGRESS_RESERVED_HEADER_DROPPED`		A header in the reserved `gp-` namespace that the client sent was dropped. Its detail names the header. See headers.
`INGRESS_CLAIMED_HEADER_DROPPED`		A header that a `[[cache.policy.vary]]` entry claims was dropped from the client's request, so a caller cannot choose its own value. Its detail names the header, never its value.
`CLIENT_MIN_VERSION_UNREADABLE`		A client profile sets a `min_version` that is not a valid version, so the floor is ignored and no request is refused for it. Your traffic is unaffected; the setting is not doing anything.
`CLIENT_PERSISTED_QUERY_UNENFORCEABLE`		A client profile requires persisted queries while `persisted_query.enabled` is `false` (for example with `preset = "off"`, the default), so the service offers the client nothing it could send. The requirement is ignored rather than refusing all of that client's traffic.
`CACHE_CLIENT_MAX_AGE_MISS`		A cached entry was older than the `max_age` on the matching client profile, so that one request went to the origin. The response reports `PASS` with `CACHE_SKIPPED_ENTRY_NOT_REPLACED`. The entry is unchanged and still serves other clients.

Persisted queries

See persisted queries for the switches these belong to.

Code	Status	Refines	Meaning
`PERSISTED_QUERY_NOT_SUPPORTED`	`200`		The request carried a hash, and this service does not resolve persisted queries (`persisted_query.enabled` is `false`). Sent to the client as a GraphQL error, with HTTP `200` so that Apollo clients read it rather than treating it as a transport failure.
`PERSISTED_QUERY_NOT_FOUND`	`200`		A persisted query hash is not registered at this location. Under APQ this is the normal signal for the client to resend the document. Sent as a GraphQL error with HTTP `200` , for the same reason.
`PERSISTED_QUERY_REGISTERED`			A client registered a persisted query with this service.
`PERSISTED_QUERY_HASH_MISMATCH`			A persisted query was offered under a hash that is not its own and was not registered. The operation still runs, so the client receives your origin's status.
`PERSISTED_QUERY_MUTATION_OVER_GET`	`405`	`METHOD_NOT_ALLOWED`	**Log only.** The caller receives `METHOD_NOT_ALLOWED` . An operation sent as a `GET` , as a document or as a persisted query hash, is not a query.
`PERSISTED_QUERY_REFUSED`	`403`	`FORBIDDEN`	**Log only.** The caller receives `FORBIDDEN` . An operation that is not in this service's registry was refused.
`PERSISTED_QUERY_REGISTRY_UNAVAILABLE`	`403`	`FORBIDDEN`	**Log only.** The caller receives `FORBIDDEN` . The registry could not be read, so the operation was refused.
`PERSISTED_QUERY_UNTRUSTED_OPERATION`			An operation ran that `only_registered` would have refused. Emitted while `audit_unregistered` is on, so you can see what your manifest is still missing.

Telemetry values

A few fields of request telemetry carry stable values of their own rather than codes. They are what the **Lookup** and policy details in the Insights request detail are drawn from.

`cache_lookup`

What the cache lookup found. Empty when no lookup ran (a pass decided before the lookup, or lean mode).

Value	Meaning
`FOUND`	An entry was found and answered the request, as a `HIT` or a `STALE` .
`NOT_FOUND`	The cache held no usable entry for the key.
`UNREADABLE_NO_LENGTH` , `UNREADABLE_BODY` , `UNREADABLE_METADATA`	An entry was found but could not be read. It is replaced with your origin's answer; there is nothing to do.

Value	Meaning
`DECLINED_CLIENT_MAX_AGE`	An entry was found and was older than the `max_age` of the client profile that asked. The entry is left as it is.
`FAILED`	The lookup itself failed: the cache was not reached (`CACHE_PASS_READ_DISABLED`).

`cache_policy_source`

Which policy decided whether the response was stored, recorded whether the decision was to store or to refuse.

Value	Meaning
`ORIGIN_HEADER`	Your origin's `Cache-Control` decided, outranking any schema policy.
`SCHEMA_POLICY`	The schema's `@cacheControl` policy decided.
`NONE`	Neither applied: no origin header that decides, and no schema hint for what the query selected.

`shield_pop` , `shield_result` and `shield_node`

With shielding switched on, three further fields record the hop to the shield. `shield_pop` is the shield location's alias, `shield_node` the individual node within it that answered, and `shield_result` how the hop ended:

Value	Meaning
`HIT`	The shield answered from its own cache.
`MISS`	The shield had no usable entry, fetched the response from your origin and stored it.
`STALE`	The shield answered a stale entry, per its own stale handling.
`PASS`	The shield stored nothing for this request: it did not use its cache, did not store your origin's response, or answered without its cache, for example because your origin failed or the request was refused there. The shield's own reason is recorded on its entry.
`FALLBACK`	The request could not be sent through the shield, so this edge location went to the origin itself. See `SHIELD_UNAVAILABLE_FALLBACK` .
`TIMEOUT`	No usable response came back from the shield: the hop exceeded its time budget, or the connection dropped or the response was incomplete, oversized or malformed. The caller receives `504` . See timeouts on a shielded service.
`UNAVAILABLE`	A mutation could not be sent through the shield, so this edge location answered `502` rather than send it to your origin itself. See `SHIELD_UNAVAILABLE_NO_FALLBACK` .

`shield_node` is empty when no shield answered: on `FALLBACK` , `UNAVAILABLE` and `TIMEOUT` .

The portal's Insights view shows them in the request detail panel, and can filter the list by the shield result.

GraphPilot API codes

The GraphPilot API reports errors in the GraphQL response format, in an ordinary `200` response. The exception is a credential that is presented and refused: an `sk_` key that is not a usable key, and any other bearer value that does not verify as a sign-in token, are refused with `401` and `UNAUTHENTICATED` before any operation runs, an `sk_` key that could not be checked at that moment with `503` and `SERVICE_UNAVAILABLE`, and an unrecognized `sk_` key sent over the validation budget with `429` and `TOO_MANY_REQUESTS`. See when a credential is refused. `POST /purge`, the REST form of two purge mutations, carries the same codes and answers each with a matching HTTP status instead of `200`.

Most errors carry their code in `extensions.code`; some refusals carry it in `extensions.error_code` instead. Read both, and branch on the code.

```
{
  "errors": [
    {
      "message": "Not authenticated to an organization",
      "extensions": { "code": "UNAUTHENTICATED" }
    }
  ],
  "data": null
}
```

`extensions.code`

Code	Meaning
`UNAUTHENTICATED`	The request carried no valid credential, or the credential is not scoped to an organization. In a `200` for a missing credential; in a `401` for an `sk_` key that is not a usable key and for any other bearer value that does not verify.
`SERVICE_UNAVAILABLE`	An `sk_` key could not be checked at that moment. In a `503` with `Retry-After`; it says nothing about the key, so send the request again.
`FORBIDDEN`	The credential is valid but lacks the permission the operation needs. Some operations name the missing `subject` and `action` in `extensions`. Also returned when an API key asks for or spends a confirmation code, and when the service did not accept the purge request's signature, which a redeploy of the service resolves.
`NOT_FOUND`	The thing addressed does not exist in the caller's organization. A service in another organization is reported the same way, as `Service not found`.
`BAD_REQUEST`	The input was refused, for example a surrogate key that is too long, too many keys in one purge, or a `purgeByQueryHash` with a `variablesHash` but no `operationName` for which your request telemetry holds no operation name (pass `operationName`, or `null` for an anonymous operation). The message says which.

Code	Meaning
`BAD_USER_INPUT`	An argument over a documented bound, such as a persisted query push over 100 documents or a document over 100,000 bytes. The message names the bound, so the caller can chunk.
`PRECONDITION_FAILED`	The operation cannot run in the current state, for example a purge against a service that has never been deployed or has no active hostname, or a deployment that asks for the newest schema or configuration before one was uploaded (send `withoutSchema` or `withoutConfig` to deploy without it).
`OPERATION_LIMIT_EXCEEDED`	Over HTTP in a `400`: the operation's document is over one of the API's operation limits, checked before it is validated. `extensions.limit` names the limit (`bytes`, `tokens`, `depth`, `selections` or `aliases`) and `extensions.max` its value.
`INVALID_STATUS`	`cancelDeployment` on a deployment that is no longer `STARTED` or `PENDING`.
`CONFIG_VERSION_NOT_ALLOWED`	`createDeployment` with both `withoutConfig` and `configVersion`. Send one of them.
`SERVICE_SETUP_NOT_FOUND`	`applyServiceSetupSuggestion` or `keepServiceAsPlainHttp` for a service that has no setup.
`SERVICE_SETUP_STATE_CONFLICT`	A setup mutation that the setup's current state does not allow, for example applying a suggestion while the first deployment is still running. Read `Service.setup` and its `canRetryDetection`, `canApplySuggestion` and `canKeepAsPlainHttp`.
`SERVICE_SETUP_SUGGESTION_INCOMPLETE`	`applyServiceSetupSuggestion` for a suggestion without a schema, because introspection did not answer. Read the schema with `gpilot schema pull` and deploy it.
`SETUP_WOULD_OVERWRITE_CONFIG`, `SETUP_WOULD_OVERWRITE_SCHEMA`	`applyServiceSetupSuggestion` for a service whose newest configuration or schema upload is your own, not one the setup wrote. The setup never replaces your files: set `request.graphql_paths` in your own `gpilot.toml` and run `gpilot deploy`. Retrying the detection and keeping plain HTTP stay allowed.
`ASSET_CHANGED_CONCURRENTLY`	`createDeployment` for a service whose schema source is **Introspect at deploy**, or `applyServiceSetupSuggestion`: an upload of the service's configuration or schema landed while GraphPilot was writing the same file, and nothing was written. Send the same request again. After `applyServiceSetupSuggestion` the setup stays in `SUGGESTION_READY`.
`SERVICE_NOT_LIVE`	`retryServiceSetupDetection` before any deployment of the service completed: detection runs through the service's own hostname, which needs a live deployment.
`ORIGIN_TIMEOUT_EXCEEDS_PLAN`	`createDeployment` with a configuration whose origin timeout is above its ceiling on your plan. Nothing is created. The message names the key, the value and the ceiling. See origin timeouts.
`RUNTIME_VERSION_YANKED`	The runtime version named in the request (`createDeployment` with `coreVersion`, or `setServiceRuntimePin`) was withdrawn. The message names the version and, where one was given, the reason. Pick another version.

Code	Meaning
`RUNTIME_VERSION_WITHDRAWN`	`createDeployment` without `coreVersion` for a service pinned to a runtime version that was withdrawn. A withdrawn version can no longer be deployed, so nothing is created. The message names the pinned version and, where one was given, the reason. Pin a supported version or remove the pin.
`INVALID_CONFIRMATION_CODE`	The confirmation code was never issued for this action and subject, has expired, has run out of attempts, or is wrong. The four cases read the same. Request a new code.
`TOO_MANY_REQUESTS`	Too many confirmation codes requested: wait a minute and try again. Also, in a `429` with `Retry-After`, an `sk_` key the API has not seen recently, sent while its client is over the validation budget for unrecognized keys. The key was not checked, so it says nothing about the key: send the request again after `Retry-After`.
`ACCOUNT_LOCKED`	The billing account funding the organization is locked, so every mutation outside billing is refused (queries still answer). `extensions.reason` says why: `PAYMENT_FAILED` (a payment is overdue: pay the outstanding invoice), `FREE_TIER` (a Glider (Free) account was locked under fair use this month: the lock lifts at the start of the next month (UTC), or at once on a paid plan once no invoice is outstanding) or `MANUAL` (contact support). `extensions.fields` lists the refused root fields. `POST /purge` answers it with `403`.
`ORGANIZATION_SLUG_TAKEN`	Another organization already uses that slug.
`ORGANIZATION_QUOTA_EXCEEDED`	`createOrganization` when your billing account already funds as many organizations as it may, 3 on every plan. Nothing is created. Delete an organization, or contact support to fund more. See organizations per billing account.
`TERMS_NOT_ACCEPTED`	`createOrganization`, or `subscribePlan` for a paid plan, without an acceptance of the terms in force. `extensions.reason` is `missing` (no acceptance sent) or `outdated` (another version than `legalTerms.version`), and `extensions.currentVersion` names the version to accept. Nothing is created or ordered. See terms acceptance.
`BUSINESS_CONFIRMATION_REQUIRED`	`subscribePlan` for a paid plan without the confirmation of ordering as a business (`extensions.reason` `not_confirmed`). Nothing is ordered.
`ACCOUNT_LOCKED`	A mutation outside billing while the billing account that funds the organization is locked. `extensions.reason` is `PAYMENT_FAILED` or `MANUAL`, and `extensions.fields` names the refused root fields. The whole operation is refused; queries are never refused. Over HTTP and WebSocket alike.
`ACCOUNT_DELETION_IN_PROGRESS`	Any mutation, billing included, in an organization that is being deleted, except a support request. `extensions.scope` is `ACCOUNT` (the account is being deleted) or `ORGANIZATION` (the organization is), and `extensions.fields` names the refused root fields.
`OUTSTANDING_INVOICE`	A plan order while an invoice is still unpaid. Pay the outstanding invoice first.
`PAYMENT_IN_PROGRESS`	A plan order while an earlier order is still being completed. Confirm its payment from the invoice if the bank asks, otherwise wait a minute and try again.

Code	Meaning
`PLAN_CHANGE_PENDING`	A plan order while an earlier plan change is waiting for the bank to confirm its payment. Confirm it from the invoice, or try again once it is discarded after about 23 hours. See when your bank asks you to confirm the payment.
`PAYMENT_FAILED`	The payment method was declined.
`RATE_LIMITED`	The payment provider is rate limiting. Try again later.
`UNAVAILABLE`	The payment provider could not be reached.
`INVALID_RETURN_URL`	A billing checkout named a return URL the portal does not own.
`DOMAIN_ALREADY_ATTACHED`	The hostname is already attached to a service.
`DOMAIN_INVALID_FORMAT`	Not a valid hostname.
`DOMAIN_QUOTA_EXCEEDED`	The plan's custom domain allowance, pooled across every organization the billing account funds, is used up.
`DOMAIN_NOT_IN_PLAN`	Custom domains are not included in the plan.
`DOMAIN_NOT_VERIFIED`	The organization has not verified a domain that covers this hostname.
`DOMAIN_NOT_RETRYABLE`	Only a failed domain can be retried.
`SERVICE_DOMAIN_LAST_ACTIVE`	A service must keep at least one active domain.
`SERVICE_DOMAIN_IS_PRIMARY`	A service must have exactly one primary domain, so the primary cannot be removed.
`SERVICE_DOMAIN_PRIMARY_NOT_ACTIVE`	The primary domain must be active.
`SERVICE_DOMAIN_PLATFORM_PERMANENT`	The platform domain can be disabled but not removed.
`INTERNAL_ERROR`, `INTERNAL_SERVER_ERROR`	A failure on the GraphPilot side. The message stays generic; retrying is the available action.

`extensions.error_code`

Some refusals carry `extensions.error_code`, and where it applies `field` (the input field) and `description`.

Code	Meaning
`INSUFFICIENT_PERMISSIONS`	A read or write of organization data without a valid credential or permission. Message `Insufficient permissions`.
`SERVICE_QUOTA_EXCEEDED`	The plan's service limit is reached. See quotas.

Code	Meaning
`SUBDOMAIN_ALREADY_EXISTS`	Another service of the organization already has this subdomain. For a service directly under `graphpilot.cloud` (created before hostname labels), any other such service.
`SUBDOMAIN_INVALID_LENGTH`	A subdomain must be between 2 and 63 characters.
`SUBDOMAIN_INVALID_FORMAT`	A subdomain may contain only lowercase letters, numbers and hyphens.
`SUBDOMAIN_RESERVED`	The subdomain is reserved. Only for a service directly under `graphpilot.cloud`; under a hostname label every valid name is allowed.
`ORIGIN_URL_INVALID`	The origin URL must be a valid HTTP or HTTPS URL.
`COLOR_INVALID_FORMAT`	A service color must be a hex color code.
`ORIGIN_URL_PATH_NOT_ALLOWED`	The origin URL may name only the scheme, host and port, such as `https://api.example.com`. Remove the path, query or fragment; the paths served as GraphQL are set with `request.graphql_paths`.
`SERVICE_CAPACITY_EXHAUSTED`	New services cannot be created right now, through no fault of the request. GraphPilot is notified automatically. Try again later.
`DOMAIN_ALREADY_ATTACHED`, `DOMAIN_INVALID_FORMAT`	As in `extensions.code`, for a write that was refused.
`UNIQUE_VIOLATION`, `CHECK_VIOLATION`, `FOREIGN_KEY_VIOLATION`, `NOT_NULL_VIOLATION`	A value already exists, is invalid, references a record that does not exist, or is missing. The message says which.

Deployment failures

A deployment that fails after it was created reports why in `Deployment.errorMessage`, in the portal and in `gpilot`. A failure to load an auth provider's key set starts with one of these codes, names the provider and every source that was tried, and stops the deployment before it is activated, because the new version would otherwise refuse every token for that provider with `401`.

A deployment that ends `CANCELLED` because GraphPilot stopped it carries a reason in `Deployment.errorMessage` too, without a code: `Replaced by a newer deployment of this service before it went live.` when a newer deployment of the same service replaced it, and `Cancelled before it went live.` otherwise. A cancellation that was requested records none.

Code	Meaning
`JWKS_URL_REFUSED`	The provider's `jwks_url` is not a URL GraphPilot fetches: it does not parse, its scheme is not `https`, it has no host, or it carries credentials. Plain `http` is accepted only on `localhost`, `127.0.0.1` and `[::1]`; on any other host it is refused with `a key set must be served over https`. Fix the URL and deploy again.

Code	Meaning
`JWKS_FETCH_FAILED`	No source answered with a usable response (a failed connection, a non-`2xx` status, or a body that is too large). The step is retried, since the issuer may be back a moment later.
`JWKS_NO_USABLE_KEY`	A source answered, but what it served holds no key the provider can verify with: not JSON, no `keys`, an empty set, or only keys of another type than the provider's algorithms need.

CLI exit codes

The `gpilot` CLI's exit codes are stable, so scripts can branch on them.

Code	Meaning
`0`	Success, or you canceled at a confirmation prompt, or you stopped watching a deployment (Ctrl+C, then "stop watching").
`1`	General error: invalid input, something not found, a command that refused to run without `--yes`, or a deployment that failed, was canceled from somewhere else, or whose wait timed out.
`2`	Usage error: an unknown command, flag or value.
`3`	Authentication failed: no usable credential, or the GraphPilot API rejected it (HTTP `401`). A permission the credential lacks is `4`, not `3`.
`4`	Network or GraphPilot API error, including a GraphQL error in the API's answer.
`5`	Not used. A new deployment cancels one in progress instead of conflicting with it, so no command returns it, and it will not be given a new meaning.
`6`	This CLI version is too old. Upgrade it (see staying up to date).
`7`	The deployment's outcome is unknown: watching it ended before it reported one (the progress stream closed, or could not be opened), and checking the deployment until `--timeout` ran out did not settle it. It is still open, or every check failed. Check with `gpilot status`.
`130`, `143`	`gpilot service create --deploy` was stopped while following the setup: Ctrl+C gives `130`, `SIGTERM` gives `143` (Unix). It prints its normal output first (with `-o`, the one document, signing key included), and the setup goes on at GraphPilot; follow it with `gpilot service setup --service <subdomain>`.

REFERENCE

Permissions

A permission is named `<resource>:<action>`. A member's role carries a set of them, and so does an API key. The GraphPilot API checks them on every operation, and the portal hides or disables most controls a member may not use. Which role holds which permissions is in the table under members and roles.

Each resource has two levels: `view` reads it, and `edit` changes it and includes `view`. A role still lists both: a key can only be given permissions the member creating it holds by name, so a role with `service:edit` alone could not create a read-only key. Three permissions stand on their own because of what they do: deleting a service, deleting the organization, and purging.

Permissions

Permission	Allows
`service:view`	Seeing the organization's services and reading them through the API, including `ipRanges`, `purges`, custom domains, telemetry and the persisted query registry. Without it, services are not visible and lists come back empty.
`service:edit`	Everything `service:view` allows, and creating a service (within the plan's allowance), changing its settings, origin, custom domains, alert rules and persisted query manifest, uploading the schema and configuration every deployment starts with, reading the deployed configuration back, and seeing (masked), rotating and activating its signing key. It does not include deleting a service or purging.
`service:delete`	Deleting a service, with a confirmation code, and resetting a service's history. Includes seeing the organization's services, not their custom domains.
`deployment:view`	Seeing deployments (**Deployments**), what each one built, its diagnostics and configuration analysis, and the portal's **Schema** and **Config** pages. Through the API, the live deployment's schema and analysis on the service (`Service.deployedSchema`, `Service.schemaAnalysis`, `Service.configAnalysis`) need only `service:view`.
`deployment:edit`	Everything `deployment:view` allows, and starting a deployment, canceling one that is still in progress, and tearing down a service's deployed edge infrastructure while keeping the service.
`cache:purge`	Every purge. `service:edit` does not include it.
`organization:edit`	Changing the organization's settings.
`organization:delete`	Deleting the organization, with a confirmation code.
`billing:view`	Seeing the plan, payment method and billing address.
`billing:edit`	Everything `billing:view` allows, and changing the plan, payment method and billing address.

Earlier permission names

Roles and API keys set up before October 2026 can carry the earlier, finer-grained names. They keep working for now, each allowing what it allowed before, and will stop being accepted once every role and key has moved to the names above. A key that carries one should be replaced with a key that carries the new name.

Earlier name	Now part of
`service:create`, `service:view-signing-key`, `service:rotate-signing-key`, `service:activate-signing-key`	`service:edit`

Earlier name	Now part of
`service:manage`	`service:edit` and `service:delete`
`deployment:create`, `deployment:delete`, `deployment:cancel-deployment`, `deployment:manage`	`deployment:edit`
`organization:manage`	`organization:edit` and `organization:delete`
`billing:manage`	`billing:edit`

Your own account is not governed by permissions: anyone can delete their own account. See your account.

What an API key can carry

An API key can carry these six permissions:

`service:view`, `service:edit`, `deployment:view`, `deployment:edit`, `cache:purge`, `billing:view`

Everything a person decides, such as organization settings, billing changes or deleting a service, is not available to a key. A key holding `service:edit` cannot see, rotate or activate the signing key: those parts of `service:edit` are removed from every API key. A key holding `service:edit` can read the deployed configuration back, as a member can. Deleting a service, an organization or an account also needs a confirmation code, which an API key cannot request or spend (`FORBIDDEN`). The earlier names `service:create`, `deployment:create` and `deployment:delete` are still accepted on a key while they are being phased out.

How a missing permission shows

Where	What happens
A read through the API	Filtered, not refused: a key without `service:view` sees no services, not an error.
A mutation through the API	Refused with `FORBIDDEN`. See codes.
The `gpilot` CLI	The API's error is printed and the CLI exits with `4`. See exit codes.
The portal	The control is hidden or disabled; a disabled one says why when you point at it.

Jobs

What each job needs. Every CLI command that takes `--service` looks the service up first, which needs `service:view`.

Job	CLI	API	Permissions
Deploy	`gpilot deploy`	`createSchemaUploadUrl`, `createConfigUploadUrl`, `createDeployment`, `deploymentProgress`	`service:edit` (uploads the schema and configuration) and `deployment:edit` (starts the deployment and waits for the result)
Cancel a deployment	`gpilot deployment cancel`	`cancelDeployment`	`deployment:edit`
Tear down the edge infrastructure	`gpilot service destroy`	`destroyDeployment`	`deployment:edit`
Pin or unpin the runtime version	`gpilot service runtime pin`, `gpilot service runtime unpin`	`setServiceRuntimePin`	`service:edit` and `deployment:edit` (changing the pin can redeploy the service, and decides what its next deployment builds)
Read a deployment's diagnostics	`gpilot deploy` output	`Deployment.diagnostics`, `Deployment.configAnalysis`	`service:view`, `deployment:view`
Read the deployed configuration	`gpilot config get`, the portal's **gpilot.toml** views	`createConfigDownloadUrl`, `Deployment.configDocument`	`service:edit`, for members and API keys alike. Every resolved setting, secrets redacted, needs only `deployment:view` (`configAnalysis`, the portal's **Config** values).
Purge the cache	`gpilot purge`	`purgeBySurrogateKey`, `purgeByQueryHash`, `purgeByRequest`, `purgeAll`, `POST /purge`	`cache:purge`
Read the purge history		`Service.purges`	`service:view`
Push persisted queries	`gpilot pq push`	`pushPersistedQueries`	`service:edit`
Revoke persisted queries	`gpilot pq rm`, `gpilot apq rm`	`removePersistedQueries`	`service:edit`
List persisted queries	`gpilot pq list`, `gpilot apq ls`	`Service.persistedQueries`, `Service.clientRegisteredQueries`	`service:view`

Job	CLI	API	Permissions
Create a service	`gpilot service create`	`createService`	`service:edit`
Read request telemetry	`gpilot requests`	`Service.requests`, `serviceRequestsStream`	`service:view`; some filters also need a plan, see Insights
Read the origin IP ranges		`Service.ipRanges`	`service:view`
Manage custom domains		`addServiceDomain` and the other domain mutations	`service:edit`
Switch an alert rule on or off		`setAlertRuleEnabled`	`service:edit`; reading the rules (`Service.alertRules`) needs `service:view`
See the signing key, masked		`Service.signingKey`, `Service.pendingSigningKey`	`service:edit`
Rotate the signing key, or cancel a rotation		`rotateServiceSigningKey`, `cancelServiceSigningKeyRotation`	`service:edit`
Activate a pending signing key		`activateServiceSigningKey`	`service:edit`
Delete a service	`gpilot service delete`	`requestActionConfirmation`, `deleteService`	`service:delete` and a confirmation code; a member, not an API key
Reset a service's history		`resetServiceHistory`	`service:delete`; a member, not an API key

The signing-key rows are for members only: an API key that carries `service:edit` still cannot do them.

REFERENCE

Limits

What each plan includes is on quotas (request units, services, custom domains) and on the pricing page (the numbers per plan), and the origin timeout ceilings are on what your plan allows. This page lists the technical limits.

Request limits

Limit	Default	Setting	Over it
Request body	8 MiB (8,388,608 bytes)	`request.max_body_bytes`	`413` , `PAYLOAD_TOO_LARGE`
Query document length	64 KiB (65,536 bytes)	`request.max_document_bytes`	`400` , `DOCUMENT_SIZE_LIMIT_EXCEEDED`
Selections per operation	4,000	`request.max_selections`	`400` , `DOCUMENT_SELECTION_LIMIT_EXCEEDED`
Query depth	10 levels	`request.max_query_depth`	`400` , `QUERY_DEPTH_LIMIT_EXCEEDED`
Introspection	Refused	`request.introspection`	`403` , `INTROSPECTION_DISABLED`
JSON nesting in a request body	128 levels		`400` , `INVALID_GRAPHQL_REQUEST`
Requests in flight, requests per second	No limit		

Every limit is inclusive: a request exactly at the limit is served. A client profile can set a second, lower ceiling for each of the first four; the stricter of the two decides. The document, selection and depth limits and the introspection block apply to a parsed GraphQL operation only.

These limits apply to HTTP requests that reach the edge. Messages inside a WebSocket are not checked against any of them, and there are no separate limits on aliases, directives, tokens or query cost. See what the edge does not check for what to enforce at your origin instead.

Request body

A body over the limit is refused before it is parsed and before it reaches your origin:

```
{
  "errors": [
    {
      "message": "The request body is too large.",
      "extensions": { "code": "PAYLOAD_TOO_LARGE" }
    }
  ]
}
```

Rule	Detail
Checked on every request	A cache hit included.

Rule	Detail
What is measured	The bytes the edge received, not the `Content-Length` the client claimed.
What the caller learns	Only that it sent too much. The limit and the refused size are in your logs (`REQUEST_BODY_TOO_LARGE` , or `CLIENT_BODY_TOO_LARGE` for a client profile's `max_body`).
Changing it	Raise or lower it with an integer. The limit cannot be removed: see `request.max_body_bytes` .
Why 8 MiB	The body is parsed into a JSON tree and rewritten, so a large `variables` object costs several times its size in memory. 8 MiB leaves a request room to be answered inside what one edge execution may spend (128 MB of memory, see edge platform limits).

Query document

The cost of a request is dominated by the document, not by the payload around it. A large `variables` object is parsed once and passed on; every field selection is parsed, planned and printed back out, so a document of thousands of aliased selections costs far more than its size suggests. The body limit does not bound that cost, so the document has two limits of its own.

Setting	Default	Checked	Refused with
`request.max_document_bytes`	`65536` (64 KiB)	The length of the `query` string, before it is parsed	`DOCUMENT_SIZE_LIMIT_EXCEEDED`
`request.max_selections`	`4000`	The number of selections, on the parsed document, before the query plan is built	`DOCUMENT_SELECTION_LIMIT_EXCEEDED`

`0` turns either limit off. 64 KiB is past any document a product ships, hand-written or generated from colocated fragments. 4,000 selections is past what an ordinary product page selects in one operation, and the standard `getIntrospectionQuery()` document is served under both defaults.

How selections are counted

Rule	Detail
Every field counts	At every nesting level, `__typename` included. An alias is a field of its own, so `a: me { id } b: me { id }` is four selections.
Fragment spreads	A spread counts the fragment's fields every time it is spread: a fragment of 50 fields spread in three places is 150. Inline fragments and the spread itself add nothing.
Fragment definitions	Every definition is paid for. A fragment the operation never spreads still counts its own fields once, and so does a second definition of a name a later one replaces.
Other operations	Only the operation the request runs counts.
`@skip` , `@include`	Not evaluated. A skipped selection is still planned, so it still counts.

What a refusal looks like

A `400`, before your origin is called, with the numbers in the message:

```
{
  "errors": [
    {
      "message": "The operation makes 6144 selections; this service allows 4000.",
      "extensions": { "code": "DOCUMENT_SELECTION_LIMIT_EXCEEDED" }
    }
  ]
}
```

`DOCUMENT_SIZE_LIMIT_EXCEEDED` reads `The operation's document is <n> bytes long; this service allows <limit>.` It is a `400` and not a `413` because the request itself may be tiny: a hash whose registered document is too long is refused too. The log line carries the measured `bytes` or `selections` and the `limit`. A document over a limit is refused, never forwarded unparsed.

Where the limits apply

Case	Detail
Every parsed document	In the request body, in an APQ registration, or resolved from the registry for a hash. A registered document costs exactly what the same text costs from anyone.
Registered documents	Held to `persisted_query.max_document_bytes` and `persisted_query.max_selections`, which inherit the `request.*` values unless set. See `[persisted query]`.
An APQ registration over the byte ceiling	Not stored, and refused with `DOCUMENT_SIZE_LIMIT_EXCEEDED`. A registration is written only once the request has cleared every other ceiling.
A cache hit	Not checked, because a hit parses nothing. Refusals are never cached.
A persisted query push	The persisted query push checks each document against the service's deployed ceilings (the defaults when the service has not deployed yet) and marks an oversized one `REJECTED`, without failing the rest of the batch.

Query depth

Ten levels is past what ordinary product queries reach and short of what a nesting attack needs.

Rule	Detail
Two ceilings	`request.max_query_depth.client_document` for documents clients send, `registered_document` for documents from the registry. `0` disables either.
How it is counted	Selection sets from the operation's own, deepest single path; fragments flattened, type conditions not a level. See how depth is counted.

Rule	Detail
Introspection	Exempt: nothing under `__schema` is planned, so an introspection query measures one level however deeply it is written. A deeply nested one is not refused by this limit either.
A query that does not parse	Refused with `400` and `INVALID_GRAPHQL_REQUEST`, never forwarded unmeasured.
GraphPilot's own ceiling	128 levels, whatever the setting says and even with it at `0`: counted on the document's brackets before it is parsed, and on its fields with fragments expanded before it is planned. Refused with `QUERY_DEPTH_LIMIT_EXCEEDED`.
A cache hit	Not checked. See the GraphQL layer runs only on a miss.

Edge platform limits

Fastly's own ceilings, from its Compute resource limits. They apply to every request, whatever GraphPilot is configured to do, and are a property of the platform GraphPilot runs on today, not of GraphPilot itself. They are the same on every plan, and no setting raises them.

Each limit counts per execution: one run of the edge program for one request at one edge location. A request through the shield runs twice, once at the edge location that received it and once at the shield, and each run is held to the limits below on its own. The wall clock of the run at the edge location includes the time it waits for the shield.

Limit	Fastly Compute
URL length	8,192 bytes
Request header length	128 KB per header
Response header length	128 KB per header
Cached object size	100 MB
Compute time	50 ms CPU, 2 min wall clock per request
Memory	128 MB heap, 1 MB stack per request
Backend requests	32 per request

The URL length matters for `GET` requests, where the operation and its variables travel in the query string. A persisted query sends a 64-character hash instead.

Response	Limit
Larger than the cached object size	Delivered to the client, not stored.

Response	Limit
Compressed by your origin	Decoded (`gzip` , `deflate` , `br` , `zstd`). A body GraphPilot cannot decode, or one that would decode to more than it processes, is answered with `502` (`ORIGIN_RESPONSE_ENCODING_UNSUPPORTED` , `ORIGIN_RESPONSE_ENCODING_CORRUPT` , `ORIGIN_RESPONSE_DECODED_TOO_LARGE`). See compression.
A GraphQL response over 133 MiB (139,460,608 bytes), or carrying an integer larger than 18446744073709551615	Not parsed, so delivered unchanged and not stored, with `CACHE_SKIPPED_UNPARSEABLE_GRAPHQL_RESPONSE` . See refusals that depend on the response.
Streamed	Not supported. A response is delivered once it is complete, so Server-Sent Events, `multipart/mixed` and incremental delivery with `@defer` or `@stream` do not work through the edge.
A total time at the edge over 120 seconds	Refused at deploy time: every attempt the retry policy allows has to fit. See timeouts and retries.
More than 32 backend requests in one request	Every origin attempt is one backend request, and so is a fetch of an auth provider's key set (`jwks_url`) and, at the edge location that received a shielded request, the hop to the shield. An `origin.retry.max_retries` above 30 is not refused at deploy time, but its later attempts cannot be sent.

Cache limits

Limit	Value
Cached object size	100 MB (Fastly). A larger response is delivered, not stored.
Surrogate keys per response	16,000 bytes in total. See below.
`gp-surrogate-key` header	8,192 bytes by default, `1` to `128000` . See size limit.
Surrogate key length for a purge	1024 characters (Unicode code points) as sent, and 1024 bytes once folded. A longer key is still attached to a response, but a purge naming it, or an empty key, is refused with `BAD_REQUEST` .
Keys per purge call	1000, counted after deduplication, per soft and hard group, and after request ids and query hashes are expanded into keys. Over it, `BAD_REQUEST` .
Query hashes per `purgeByQueryHash`	1000 entries in `queries` . A longer list is refused with `BAD_REQUEST` before any entry is resolved.
Purges	No rate limit and no cap on calls.

How many keys fit on one response

The surrogate keys of one response are stored with the cache entry, and the edge allows them **16,000 bytes** in total. GraphPilot stores each key as a short hash, so about 1,300 to 2,000 keys fit, depending on how many a response carries. The count includes the operation and cache keys GraphPilot adds to every response.

Rule	Detail
No key is ever dropped	If the keys do not fit, the response is not cached: `CACHE_SKIPPED_SURROGATE_KEYS_TOO_LARGE`. An entry stored without some of its keys could never be reached by a purge.
Collisions	Two keys can share a hash. A collision can only over-invalidate (a purge also removes the entries of the key sharing its hash), never leave stale data.
Keys that identify nothing still count	A field that merely happens to be named `id`, `_id` or `key`, or to have a key scalar type, adds a key per object. Switch the automatic rules off there with `@surrogateKey(ignore: true)` or `surrogate_key.ignored_types` and `ignored_fields`.
Inherited keys count double	A key inherited from an interface or union adds up to two keys per object, one named after the abstract type and one after the object's own type.
A long list	A response with thousands of keys is usually a long list; paginating it keeps each page under the limit.

GraphPilot API limits

Limit	Value	Over it
Operation document size	32 KiB (32,768 bytes)	`400`, `OPERATION_LIMIT_EXCEEDED`
Operation tokens	5,000	`400`, `OPERATION_LIMIT_EXCEEDED`
Operation depth	15 levels	`400`, `OPERATION_LIMIT_EXCEEDED`
Operation selections	500	`400`, `OPERATION_LIMIT_EXCEEDED`
Operation aliases	30	`400`, `OPERATION_LIMIT_EXCEEDED`
WebSocket message	131,072 bytes	Connection closed with `1009`
`pushPersistedQueries` documents per call	100	`BAD_USER_INPUT`; nothing is applied
Persisted query document size	100,000 bytes	`BAD_USER_INPUT`; nothing is applied
`removePersistedQueries` hashes per call	1000, the edge's own limit on purge keys in one request	`BAD_USER_INPUT`; nothing is applied
`Service.clientRegisteredQueries` page	Default 50, at most 100	Clamped
Purge keys per call	1000	`BAD_REQUEST`

Limit	Value	Over it
`Service.purges` page	Default 20, at most 1000	Clamped
`Service.requests` , `operations` , `errors` , `clients` page	Default 20, at most 1000	Clamped
`Service.logs` page	Default 100, at most 1000	Clamped
Time series	At most 400 buckets. A finer granularity is coarsened until the window fits.	Coarsened
Top lists (`top`)	Default 10, at most 100 entries	Clamped
Series in a grouped time series (`topGroups`)	Default 10, at most 50; the rest is summed into one `other` series	Clamped
Time range of a telemetry query	Clipped to what your plan retains. See Insights.	Clipped
`serviceRequestsStream` , `serviceOperationsStream`	Closed after 5 minutes without a new entry	Resubscribe
`serviceLogsStream`	Closed after 30 minutes	Resubscribe
Confirmation codes requested	3 per user per 60 seconds. A code expires after 10 minutes.	`TOO_MANY_REQUESTS`
Attempts per confirmation code	5, a correct attempt included. Requesting a new code replaces the earlier one.	`INVALID_CONFIRMATION_CODE`
API key validation	Cached for up to 60 seconds, so a revoked key can work for up to a minute	
Unrecognized API keys checked per client address	10, then one more every 6 seconds. See API key validation.	`429` , `TOO_MANY_REQUESTS` , with `Retry-After`
Unrecognized API keys checked across all clients	20, then two more per second. See API key validation.	`429` , `TOO_MANY_REQUESTS` , with `Retry-After`

The purge mutations have no rate limit of their own; an unrecognized key sent to them counts against the API key validation budget below like any other request.

Operation limits

The API checks every operation document before it validates it, on HTTP and on a WebSocket alike. These limits are the API's own and are separate from the request limits your service sets for its edge.

Rule	Detail
Size	Bytes of the `query` string, UTF-8.

Rule	Detail
Depth	Fields that carry a selection set, through fragments. Introspection (`__schema` , `__type`) is not held to it.
Selections	Every field and every fragment spread, once per place it is reached: a fragment spread twenty times counts twenty times. A fragment no operation spreads counts too.
Aliases	Counted as written in the document.
Refusal	Over HTTP a `400` ; over a WebSocket the error for that operation. `extensions.code` is `OPERATION_LIMIT_EXCEEDED` , `extensions.limit` names the limit (`bytes` , `tokens` , `depth` , `selections` or `aliases`) and `extensions.max` its value.

API key validation

Each `sk_` key the API has not accepted within the last minute is looked up with the identity provider before the request runs. Those lookups are budgeted, so that a flood of invented keys cannot crowd out real ones:

- A key the API accepted within the last minute is answered from its cache and never counts against the budget, however much else is going on.
- Every other well-formed key counts: a key whose acceptance is more than a minute old (it is looked up again then, however often it is used), a new key on its first use, and every unknown, revoked or mistyped one. Within the budget it is looked up as usual; over it, the request is answered `429` with `TOO_MANY_REQUESTS` and `Retry-After` , without the key being checked.
- A client address is the one the edge saw. An IPv6 address counts by its `/64` .
- A key that was refused is remembered as refused for 10 seconds, so repeating it does not count again in that time.
- The numbers above are a floor for what one client can send, not an exact count.

A CI job that presents one key uses one lookup a minute at most. A `429` here means a client has presented many different keys in a short time, or the API is answering a flood of them; send the request again after `Retry-After` .

> **Note**
> The persisted query document size in the API is a bound on the upload, not the edge's document ceiling. What the edge accepts is `persisted_query.max_document_bytes` , which the push checks per document against the service's deployed configuration, rejecting an oversized document on its own.

REFERENCE

GraphPilot API

The GraphPilot API is the GraphQL API the portal and the `gpilot` CLI are built on. Scripts and CI jobs call it directly. This page lists the operations documented for that use; the portal reads more through it (billing, members, settings), which is not covered here.

Topic	Where
Which permission each operation needs	Permissions
Every error code	Codes
Operation limits, page sizes, batch sizes, stream lifetimes	Limits

Endpoint

Transport	Address
Queries and mutations	`POST https://api.graphpilot.io/graphql`, with a JSON body holding `query` and, where used, `variables`
Subscriptions	`wss://api.graphpilot.io/graphql`, protocol `graphql-transport-ws`
Purging without GraphQL	`POST https://api.graphpilot.io/purge`, with a JSON body; see purging

Authentication

Send an API key as a bearer credential:

```
Authorization: Bearer <api-key>
```

Property	Behavior
Scheme	Written exactly `Bearer`, followed by one space.
Where keys come from	Created in the portal under the organization's **Settings** → **API keys**.
Scope	One organization. Each operation acts in the organization the key belongs to. The one operation that takes an organization argument, `deleteOrganization`, only accepts that same organization.
Permissions	The permissions selected when the key was created. See permissions.
Validation cache	A key's validation result is cached for up to 60 seconds, so a revoked key can take up to a minute to stop working. See how long key changes take.
A missing, unknown or unusable credential	See when a credential is refused.
Subscriptions	The key travels in the `connection_init` payload, not in an HTTP header: `{ "Authorization": "Bearer <api-key>" }`. Key names are matched case-insensitively.

```
curl -sS https://api.graphpilot.io/graphql \
  -H "Authorization: Bearer $GRAPHPILOT_API_KEY" \
```

```
  -H "Content-Type: application/json" \
  -d '{"query":"{ organization { slug services { subdomain } } }"}'
```

Errors follow the GraphQL response format: an `errors` array beside `data`, in an ordinary `200` response, with a code in `extensions.code` or, for some refusals, `extensions.error_code`. Every code is on codes. There are two exceptions. A credential can be refused before any operation runs (see below). An operation over one of the API's operation limits is refused with `400` and `OPERATION_LIMIT_EXCEEDED` before it is validated.

When a credential is refused

How a bad credential is answered depends on what it looks like. A credential that was presented is never downgraded to an anonymous request: it is either accepted or refused before any operation runs. Read `extensions.code` whatever the status is.

The request carries	Status	What happens
No `Authorization` header	`200`	The request runs without an identity, and each operation refuses on its own terms: `UNAUTHENTICATED`, or `Insufficient permissions` for a read of organization data.
A value starting with `sk_` that is not a usable key: never issued, revoked, rotated or mistyped after the prefix, a key that is not scoped to an organization, or one whose every permission is one an API key may not hold	`401`	Refused by the API before any operation runs, with `WWW-Authenticate: Bearer error="invalid_token"` and a GraphQL error whose `extensions.code` is `UNAUTHENTICATED`. The message names the reason, for example "API key not accepted: revoked, rotated or mistyped." A value that cannot be a key at all (fewer than 8 characters after `sk_`, more than 512 in all, or a character a bearer credential may not contain) is refused the same way without being looked up, with "API key not accepted: it is not a well-formed key. Check that the whole key was copied."
Any other bearer value that does not verify as a sign-in token: a key missing its `sk_` prefix, a truncated or malformed value, an expired session token, or `Bearer` with nothing after it	`401`	Refused by the edge in front of the API before any operation runs, with a GraphQL error whose `extensions.code` is `UNAUTHENTICATED` and whose message is "The provided token could not be verified."
An `sk_` key that could not be checked at that moment	`503`	Nothing is said about the key. The response carries `Retry-After: 5` and a GraphQL error whose `extensions.code` is `SERVICE_UNAVAILABLE`. Send the request again after that many seconds.
An `sk_` key GraphPilot has not seen recently, sent while its client is over the validation budget for unrecognized keys	`429`	The key was not checked, so nothing is said about it. The response carries `Retry-After` with the seconds until the budget allows another check, and a GraphQL error whose `extensions.code` is `TOO_MANY_REQUESTS`. A key the API accepted within the last minute is answered from its cache and is not affected.

On a subscription, where the key travels in `connection_init`, the same outcomes close the WebSocket instead: `4401` for a refusal, `1013` (try again later) for a key that could not be checked or was not checked for want of budget.

A client that only parses `200` responses meets the `401` on exactly the case a typo in a configuration file or a revoked key produces. Handle every status.

Finding a service

Most operations start from a service. Look one up by its subdomain (the `<subdomain>` in `https://<subdomain>.<label>.graphpilot.cloud`, the value `--service` takes in the CLI):

```
query {
  services(condition: { subdomain: "my-shop" }) {
    id
    subdomain
    name
  }
}
```

`services` returns a list holding the one service with that subdomain. `id` is the service's id in this API. Every `serviceId` argument on this page accepts this `id`; the purge mutations also accept the subdomain.

Services and organizations

Operation	Kind	Does
`organization`	Query	The organization the request is scoped to, or `null`.
`organizations`	Query	Every organization the caller has access to.
`services(condition:)`	Query	The organization's services.
`createService(input: { service: { name, subdomain, originUrl, environment, color }, deploy })`	Mutation	Creates a service. The payload's `signingKey` is the service's signing key in full, the only time it is returned. `deploy: true` also starts the setup: a first deployment without a schema and without a configuration file, then GraphQL detection. Omitted or `false`, nothing is deployed.
`classifyServiceOrigin(originUrl:)`	Query	Whether the edge could reach an origin URL, decided from the URL alone without calling it: `reachableFromEdge`, and for a private or local address a `reason` (`PRIVATE_ADDRESS`, `LOOPBACK_ADDRESS`, `LINK_LOCAL_ADDRESS`, `SHARED_ADDRESS`, `UNSPECIFIED_ADDRESS`, `MULTICAST_ADDRESS`, `RESERVED_ADDRESS`, `LOCAL_HOSTNAME` or `INVALID_URL`) and a `message`. An IPv4 address carried in an IPv6 one (NAT64, translated, 6to4) is judged as that IPv4 address.
`updateServiceById(input: { id, patch })`	Mutation	Changes a service's settings and origin, and its `schemaSource`.

Operation	Kind	Does
`Service.schemaSource`	Field	Where a deployment takes its schema from when it names none: `UPLOAD` (the newest upload, the default), `NONE` (no schema) or `INTROSPECT_AT_DEPLOY` (read from the origin through the service's hostname at every deployment). See schema source.
`Service.hasOwnDeployment`	Field	`true` once a deployment requested by a member or an API key completed, as opposed to the automatic first deployment of the setup.
`requestActionConfirmation(input: { action, subjectId })`	Mutation	Sends the caller a confirmation code by email. Returns `sentTo` (masked) and `expiresAt`.
`deleteService(input: { serviceId, confirmationCode })`	Mutation	Deletes a service and tears down its edge infrastructure. Returns `Boolean!`.
`legalTerms`	Query	The terms in force: `version`, `url` and `privacyPolicyUrl`. Answered without a credential.
`createOrganization(input:)`	Mutation	Creates an organization. Refused for an API key. Requires `termsAcceptance: { termsVersion }` naming `legalTerms.version`, see terms acceptance. A billing account funds at most 3 organizations; a fourth is refused with `ORGANIZATION_QUOTA_EXCEEDED`.
`updateOrganization(input: { organizationId, name, slug, domains, websiteUrl, industry })`	Mutation	Changes the caller's current organization. `organizationId` must name it; every other field is optional.
`deleteOrganization(organizationId:, confirmationCode:)`	Mutation	Deletes the caller's current organization. Returns `Boolean!`.
`deleteMyAccount(input: { confirmationCode })`	Mutation	Starts deleting the caller's own account; the deletion completes in the background. Returns `Boolean!`. See your account.
`Service.url`, `Service.primaryDomain`	Field	The hostname the service answers on: `url` as `https://` plus the hostname, `primaryDomain` as its `ServiceDomain`. The service hostname until a custom domain is made primary with `setPrimaryDomain`.
`Organization.accountLockReason`, `accountLockedAt`, `accountLockDueAt`, `accountTerminationDueAt`, `accountTerminatedAt`	Field	The state of the billing account that funds the organization, readable without billing access: why and since when it is locked (`PAYMENT_FAILED` or `MANUAL`), and for an overdue payment the day it is locked, the day the paid plan may be terminated, and when it was. `null` when they do not apply.

Terms acceptance

Creating an organization and ordering a paid plan each accept the terms in force, and the API refuses both without that acceptance. `legalTerms.version` is the version to name; any other is refused, so read it again after the terms change.

Operation	Input	Required
`createOrganization`	`termsAcceptance: { termsVersion }`	Always.
`subscribePlan`	`termsAcceptance: { termsVersion, actsAsBusiness, companyName }`	For a paid plan. `actsAsBusiness` must be `true` (the customer orders for a company or as a self-employed professional, a business in the sense of § 14 BGB). `companyName` is optional. Ignored for the free plan.

A refusal changes nothing. Without an acceptance, or with another version, the error is `TERMS_NOT_ACCEPTED`, with `extensions.reason` set to `missing` or `outdated` and `extensions.currentVersion` naming the version to accept. Without the business confirmation it is `BUSINESS_CONFIRMATION_REQUIRED`, with `extensions.reason` set to `not_confirmed`. Each accepted request is recorded with the version, the time and the user, by id and with the email address and name of the user's login at that moment; a paid order before the order is sent. If the user's details cannot be read, the request fails and nothing is created or ordered.

Invoices are addressed to the name on the billing account's billing address (`BillingAddress.companyName`). A paid order with a `companyName` that is not blank replaces that name before the order is sent, and if that fails, the request fails and nothing is ordered. Without one, or with a blank one, the billing address keeps its name.

Confirmations

Deleting a service, an organization or your account needs a code sent by email.

Rule	Detail
Actions	`DELETE_SERVICE`, `DELETE_ORGANIZATION`, `DELETE_ACCOUNT` (the `action` of `requestActionConfirmation`).
Who	An interactive user only. An API key is refused with `FORBIDDEN`, both when requesting and when spending a code.
Authorization	Requesting a code runs the same checks as the action itself: a subject the caller cannot see is `NOT_FOUND`.
Lifetime	A code expires after 10 minutes and is spent by the action it confirms, so it works once.
Attempts	5 per code. Every attempt counts, a correct one included; from the sixth on, even the right code is refused with `INVALID_CONFIRMATION_CODE`. Request a new code.
A new code	Replaces the live code for the same action and subject: the earlier one stops working.
Rate limit	At most 3 codes per user per 60 seconds, counted across all three actions (`TOO_MANY_REQUESTS`).

Rule	Detail
A wrong code	`INVALID_CONFIRMATION_CODE`, whether it was never issued, expired, out of attempts or mistyped.

Deployments

A deployment builds from the schema and configuration last uploaded for the service, unless it names other versions or asks for none.

Operation	Kind	Does
`createSchemaUploadUrl(serviceId:)`	Mutation	A signed URL. Upload the schema with `PUT` and `Content-Type: application/graphql`. The response's `x-goog-generation` header is the version.
`createConfigUploadUrl(serviceId:)`	Mutation	The same for the configuration, with `Content-Type: application/toml`.
`createConfigDownloadUrl(serviceId:)`	Mutation	A signed URL to `GET` the stored configuration. `NOT_FOUND` if none was uploaded.
`Service.latestSchema`, `Service.latestConfig`	Field	`version`, `md5`, `crc32c`, `size`, `createdAt`, and `commitHash` and `commitMessage` when the upload sent them as `x-goog-meta-commit-hash` and `x-goog-meta-commit-message`.
`createDeployment(input: { serviceId, schemaVersion, withoutSchema, configVersion, withoutConfig, forceBuild, coreVersion })`	Mutation	Starts a deployment. An omitted schema version follows the service's `schemaSource` (the latest upload by default); an omitted config version takes the latest upload. `withoutSchema: true` deploys without a schema, as a plain HTTP service with the GraphQL features off, and records a `null` schema version; it cannot be combined with `schemaVersion`. `withoutConfig: true` deploys without a configuration file, every setting at its default, and records a `null` config version; it cannot be combined with `configVersion` (`CONFIG_VERSION_NOT_ALLOWED`). `forceBuild` rebuilds even when a cached build exists. `coreVersion` is a runtime version; omitted, the service's pin is used, else the stable version. For a service that introspects at deploy, refused with `ASSET_CHANGED_CONCURRENTLY` when an upload of the same file landed while GraphPilot was writing it; nothing is written, so send it again. Returns `deployment`.
`cancelDeployment(input: { deployment })`	Mutation	Cancels a deployment in `STARTED` or `PENDING`. Otherwise `INVALID_STATUS`. Returns `result`.

Operation	Kind	Does
`destroyDeployment(input: { serviceId })`	Mutation	Tears down the service's deployed edge infrastructure, keeping the service, which can be deployed again. Runs asynchronously; returns `success` and `service`. Deleting a service does this too.
`deploymentProgress(deploymentId:)`	Query	The deployment's progress, once.
`deploymentProgress(deploymentId:)`	Subscription	The same, sent on connect and again on every change.
`Service.deployedSchema`	Field	The schema the service's most recent completed deployment was built from. `null` if it was never deployed or deployed without a schema.
`Deployment.schemaDocument`, `Deployment.configDocument`	Field	The schema and the configuration a deployment was built from. `null` when the deployment pinned none. `configDocument` needs `service:edit`, like `createConfigDownloadUrl`, because a configuration can hold a provider's shared secret: without it the field alone is refused with `FORBIDDEN` and the rest of the query answers. `schemaDocument` needs only `deployment:view`, and `configAnalysis` stays readable with secrets redacted.
`Deployment.configVersion`, `Deployment.usesDefaultConfig`	Field	The configuration generation the deployment pinned, `null` for a deployment without a configuration file, for which `usesDefaultConfig` is `true`. Its `configAnalysis` still lists every setting, each with `source: default`.
`Service.defaultConfig`	Field	The effective defaults: the configuration report of the service's newest completed deployment without a configuration file, the same shape as `configAnalysis`. `null` when there is none.
`Deployment.requestedBy`, `Deployment.requestedByPlatform`	Field	Who asked for the deployment: `requestedBy` is a person or an API key, `null` when nobody was recorded. For a deployment GraphPilot requested itself, `requestedBy` is `null` and `requestedByPlatform` names why: `SERVICE_SETUP` for the automatic first deployment of the setup. `requestedByPlatform` is `null` for every other deployment.

Operation	Kind	Does
`Deployment.schemaSource`, `Deployment.schemaIntrospection`	Field	Where the deployment's schema came from: `UPLOAD`, `INTROSPECTION` or `NONE` (`null` for deployments from before this was recorded). For a service that introspects at deploy, `schemaIntrospection` has `succeeded`, `path`, `error` and `fallbackSchemaFrom`, the upload time of the schema used instead when introspection failed (`null` when there was none, and the deployment then has no schema).
`Deployment.schemaChange`	Field	What changed against the schema that was live when the deployment was requested: `previousSchemaVersion`, `addedTypes`, `removedTypes`, `addedFields` and `removedFields` (as `Type.field`), and a `summary` such as `+3 types, -1 field` or `No changes`. `null` when either side has no schema.
`Deployment.errorMessage`	Field	Why the deployment ended without going live. For `FAILED`, the failure (see deployment failures). For `CANCELLED`, the reason when GraphPilot canceled the run: `Replaced by a newer deployment of this service before it went live.` when a newer deployment of the service replaced it, `Cancelled before it went live.` otherwise; `null` when the cancellation was requested, as with `cancelDeployment`. `null` for every other status.
`Service.deployments`	Field	The service's deployments, newest first, with `status` (`STARTED`, `PENDING`, `COMPLETED`, `FAILED` or `CANCELLED`), `diagnostics` and `configAnalysis`.

`DeploymentProgress` field	Meaning
`deploymentId`	The deployment id.
`status`	`RUNNING`, `COMPLETED`, `FAILED` or `CANCELLED`.
`steps`	Each step's `id`, `name`, `status` (`PENDING`, `RUNNING`, `COMPLETED`, `FAILED`, `SKIPPED`), `startedAt`, `completedAt` and `error`.
`progress`	`total`, `completed` and `percentage` of steps.
`timestamp`	Unix time in milliseconds.

```
query {
  services(condition: { subdomain: "my-shop" }) {
    deployments(first: 1) {
      id
      status
```

```
      createdAt
      diagnostics { origin source severity message key line column }
      configAnalysis {
        unknown { path line }
        inert { path line }
      }
    }
  }
}
```

Deployment.diagnostics

What the build found wrong in your schema and configuration without refusing the deployment.

Field	Type	Meaning
`origin`	`String!`	Which document the finding is about: `schema` or `config`.
`source`	`String!`	Which check produced it, such as `cache_control`, `surrogate_key` or `auth`.
`severity`	`SchemaDiagnosticSeverity!`	`ERROR` (what you wrote has no effect), `WARNING` (it works as written, probably not as meant) or `INFO` (a notice, nothing to fix: for example that a deployment has no schema). None of them fails the deployment.
`message`	`String!`	The finding, in words.
`key`	`String`	For a configuration finding, the key it is about, such as `client.profile[0].when`. `null` for a schema finding.
`line`	`Int`	One-based line in the schema. `null` where the finding has no position, and always `null` for a configuration finding.
`column`	`Int`	One-based column in the schema, on the same terms as `line`.

`null` for the whole list means nothing was recorded: the deployment predates stored diagnostics or never got as far as building. An empty list means the build found nothing. `origin` and `key` read `schema` and `null` on records written before those fields existed. What each `source` reports is under deployments.

Deployment.configAnalysis

Field	Type	Meaning
`unknown`	`[ConfigKeyReference!]!`	Keys your configuration writes that nothing reads, usually a misspelling.
`inert`	`[ConfigKeyReference!]!`	Keys that are valid and read by nothing, reported only where you wrote them.

Field	Type	Meaning
`settings`	`[ConfigSetting!]!`	Every setting the configuration resolved to, sorted by path, defaults included.
`config`	`String!`	The file name the configuration was read from.
`version`	`Int!`	The shape of the stored report.

Each `ConfigSetting` has a `path` (such as `client.profile[0].max_age`), a `value` (JSON, `null` when `redacted`), a `source` (`explicit` when your configuration names the path, `default` when it does not) and `redacted` (`true` when the value is withheld). `Service.configAnalysis` returns the same report for the deployment the service runs now.

Each `ConfigKeyReference` has a `path` (the key) and a `line` (one-based, in the configuration file the deployment was built from). `configAnalysis` is `null` for a build with no configuration report, such as one older than 90 days.

Service setup

The setup a service created with `deploy: true` goes through, the same one the portal shows as a card on the service's dashboard and `gpilot service create` walks through. Poll `Service.setup` about every three seconds while `state` is `DEPLOYING`, `WAITING_FOR_CERTIFICATE`, `DETECTING` or `APPLYING`.

Operation	Kind	Does
`Service.setup`	Field	The setup, or `null` for a service created without it.
`retryServiceSetupDetection(input: { serviceId })`	Mutation	Probes the origin again (state `DETECTING`). Needs `deployment:edit`. Starts a setup for a service that has none, once a deployment of it completed (otherwise `SERVICE_NOT_LIVE`).
`applyServiceSetupSuggestion(input: { serviceId })`	Mutation	Uploads the suggested schema and, when it is not the default, the suggested configuration, and starts a deployment with them, requested by the caller (state `APPLYING`). Only in `SUGGESTION_READY` and with a schema (`SERVICE_SETUP_SUGGESTION_INCOMPLETE` otherwise). Never replaces files of your own: refused with `SETUP_WOULD_OVERWRITE_CONFIG` or `SETUP_WOULD_OVERWRITE_SCHEMA` when the service's newest configuration or schema upload was not written by the setup; `canApplySuggestion` does not reflect this. Refused with `ASSET_CHANGED_CONCURRENTLY` when an upload of the service's configuration or schema landed while GraphPilot was writing it; nothing is written and the setup stays in `SUGGESTION_READY`, so apply again. Needs `deployment:edit`. Returns `setup` and `deployment`.
`keepServiceAsPlainHttp(input: { serviceId })`	Mutation	Closes the setup and leaves the service as it is (state `DONE`, outcome `PLAIN_HTTP`). Not while something is running. Needs `service:edit`.

A mutation the current state does not allow is refused with `SERVICE_SETUP_STATE_CONFLICT`; the setup's `canRetryDetection`, `canApplySuggestion` and `canKeepAsPlainHttp` say which are allowed. A setup does not stay in `APPLYING` or `DETECTING`: a step that cannot finish ends in `FAILED` with `SETUP_INTERRUPTED`.

`ServiceSetup` field	Meaning
`state`	`CREATED`, `DEPLOYING`, `WAITING_FOR_CERTIFICATE`, `DETECTING`, `SUGGESTION_READY` (GraphQL found), `NO_GRAPHQL_FOUND`, `ORIGIN_UNREACHABLE` (the edge could not reach the origin), `SKIPPED_PRIVATE_ORIGIN` (a private or local origin, not deployed), `APPLYING`, `DONE` or `FAILED`.
`outcome`	For `DONE`: `GRAPHQL` (the suggestion went live), `PLAIN_HTTP` (kept as plain HTTP) or `OWN_DEPLOYMENT` (a deployment of the caller's own replaced the setup).
`message`	A sentence for the current state.
`errorCode`	For `FAILED`, `ORIGIN_UNREACHABLE`, `NO_GRAPHQL_FOUND` and `SKIPPED_PRIVATE_ORIGIN`: `NOT_ALLOWED_TO_DEPLOY`, `PRIVATE_ORIGIN`, `DEPLOYMENT_FAILED`, `CERTIFICATE_FAILED`, `CERTIFICATE_TIMEOUT` (the certificate was still pending after 60 minutes), `ORIGIN_UNREACHABLE`, `NO_GRAPHQL_FOUND`, `DETECTION_FAILED`, `SETUP_INTERRUPTED` (a step stopped before it could finish, for example an apply that got no deployment within 10 minutes; retry the detection or keep plain HTTP), `ACCOUNT_LOCKED` or `BEING_DELETED` (the first deployment was not requested because the billing account is locked or the organization or account is being deleted).
`initialDeployment`, `appliedDeployment`	The automatic first deployment, and the one `applyServiceSetupSuggestion` started.
`detection`	`graphqlPath` (the path that answered `{ __typename }`), `originUnreachable` (no probe got an answer from the origin), `introspectionAvailable`, `introspectionError`, `attempts`, and the `probes`, each with `path`, `status`, `result` and `durationMs`. `result` is `GRAPHQL`, `NOT_GRAPHQL` (the origin answered, but not as GraphQL, including its own `5xx`), `UNREACHABLE` (the edge's own `502` with `BAD_GATEWAY`, or no answer), `REFUSED` (the edge refused GraphPilot's probe; the setup is then `FAILED` with `DETECTION_FAILED`) or `NOT_TRIED`.
`suggestion`	`graphqlPath`, `configDocument` (the `gpilot.toml` it deploys, `null` for the defaults), `schemaDocument` (the schema read by introspection, `null` without one) and `typeCount`.
`createdAt`, `stateChangedAt`, `history`	When the setup started and last changed state, and every state it passed through with its time and message.

A service created with `deploy: true` by a caller who may not create deployments is created, and its setup is `FAILED` with `NOT_ALLOWED_TO_DEPLOY`. Creating the service never fails because the setup could not be started; the service then has `setup: null`. The service hostname is `Service.url`.

Runtime versions

The catalog of runtime versions a service can be built against. The two queries are answered without a credential.

Operation	Kind	Does
`runtimeVersions(includeYanked: Boolean = false)`	Query	Every version, newest first. Withdrawn versions only with `includeYanked: true`.
`runtimeChangelog(from:, to:)`	Query	The versions released after `from`, up to and including `to` (default: the stable version), newest first, withdrawn ones included.
`Service.runtimePin`	Field	The version the service is pinned to. `null` follows the stable version.
`Service.runtimeVersion`	Field	The version the active deployment was built with. `null` for a service never deployed.
`Service.runtimeTarget`	Field	The version the next deployment is built with: the pin, else the stable version. `null` while the pin names a withdrawn version.
`setServiceRuntimePin(input: { serviceId, version, deployNow })`	Mutation	Pins the service to `version`, or with `version: null` removes the pin. `deployNow` (default `true`) rebuilds the service with the version the pin resolves to; `false` saves the pin alone, and the service's next deployment uses it. Even with `deployNow: true`, no deployment starts when the live deployment already runs that version and no deployment is in progress. Returns `service`, `resolvedVersion` (the pinned version, or the stable version for no pin), `outcome` and `deployment`. `outcome` is `DEPLOYMENT_STARTED`, `ALREADY_RUNNING`, `DEPLOYMENT_DEFERRED` (`deployNow: false`) or `NEVER_DEPLOYED` (the service's first deployment uses the pin); `deployment` is the new deployment for `DEPLOYMENT_STARTED` and `null` otherwise. Removing the pin while no stable version exists is refused with `PRECONDITION_FAILED` (`No runtime version is on channel stable yet`).

`RuntimeVersion` field	Meaning
`version`	The name to pin.
`status`	`ACTIVE` (offered), `DEPRECATED` (still served, no longer offered) or `YANKED` (withdrawn).
`releasedAt`	When the version was made available.
`yankedAt`, `yankReason`	When and why it was withdrawn. `null` unless `YANKED`.
`notes`	Release notes, in Markdown. Empty when there are none.
`channels`	The channels that point at it now, such as `stable`.

Purging

Four mutations purge a service's cache. All take a `serviceId` (the id or the subdomain) and return the same `PurgePayload`. Every failure is a GraphQL error rather than a payload with `success: false`.

Mutation	Addresses
`purgeBySurrogateKey`	One or more surrogate keys, or a cache key written with its `ck:` prefix
`purgeByQueryHash`	A query, optionally narrowed to one variable set
`purgeByRequest`	The cache entry one telemetry request read or wrote
`purgeAll`	Everything the service has cached

```
mutation {
  purgeBySurrogateKey(input: {
    serviceId: "my-shop"
    keys: ["Product:id:42"]
  }) {
    purgedKeyCount
    purgedKeys
  }
}
```

`purgedKeyCount` counts keys, not cache objects. Inputs, soft purges and failure modes are in purging through the API.

`purgeBySurrogateKey` and `purgeAll` can also be sent without GraphQL, to `POST /purge` on the same host. It runs the mutation with the same credential and permission, answers `200` with the `PurgePayload` as the body, and answers a refusal with a matching HTTP status and the same `errors` list. The body forms and the status for each code are under over REST.

```
curl -sS https://api.graphpilot.io/purge \
  -H "Authorization: Bearer $GRAPHPILOT_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"serviceId":"my-shop","keys":["Product:id:42"]}'
```

`Service.purges`

Every invalidation of a service's cache, automatic and manual, newest first. See verifying a purge. The history is request telemetry, stored and kept under the rules on what Insights needs.

```
query {
  services(condition: { subdomain: "my-shop" }) {
    purges(
      first: 20
      timeRange: { from: "2026-09-22T00:00:00Z", to: "2026-09-23T00:00:00Z" }
      filter: { trigger: [MANUAL] }
    ) {
      nodes { id timestamp trigger soft keys keyCount error actor { email } }
```

```
      pageInfo { hasNextPage endCursor }
    }
  }
}
```

Argument	Type	Meaning
`timeRange`	`TimeRangeInput!`	Required. `from` and `to` as ISO 8601 timestamps, both inclusive. Clipped to what your plan retains; a window entirely older returns an empty page.
`first`	`PositiveInt`	Page size. Default 20, at most 1000.
`after`	`String`	The `endCursor` of the previous page.
`filter`	`PurgesFilterInput`	Optional. `trigger` (`AUTO`, `MANUAL`), `scope` (`EVERYTHING`, `KEYS`), `soft`, `outcome` (`SUCCEEDED`, `FAILED`), `key` (at least one key containing this text, matched literally) and `actorUserIds`. Fields combine with AND.

`PurgeEntry` field	Type	Meaning
`id`	`ID!`	The entry's id.
`timestamp`	`DateTime!`	When the purge ran.
`trigger`	`PurgeTrigger!`	`AUTO` for a purge caused by a mutation, `MANUAL` for one sent through the API, the CLI or the portal.
`soft`	`Boolean!`	`true` when entries were marked stale rather than removed.
`keys`	`[String!]!`	The surrogate keys addressed. Empty for a purge of everything.
`keyCount`	`Int!`	How many keys were addressed, not cache objects. `0` for a purge of everything.
`error`	`String`	`null` when no error was recorded. Otherwise the cache is known to be wrong, and for a keyed purge an unknown part of the keys may already have been purged. The value is a fixed message, not the edge's own failure text.
`actor`	`User`	The member who sent a manual purge. `null` for an automatic purge, for a purge sent with an API key (which is not recorded under a user id), and for a user who no longer exists.
`triggeredByRequest`	`RequestEntry`	For an automatic purge, the mutation request that caused it, while it is within telemetry retention.
`targetRequest`	`RequestEntry`	For a purge addressed by request id, the request named, while it is within telemetry retention.

Persisted queries

See persisted queries for how the registry is used at the edge.

Operation	Kind	Does
`pushPersistedQueries(input: { serviceId, documents: [{ hash, document }] })`	Mutation	Registers manifest documents. Additive and idempotent: nothing is removed, and re-sending an entry is a no-op. Each `hash` is the hex SHA-256 of its `document` and is verified.
`removePersistedQueries(input: { serviceId, hashes })`	Mutation	Revokes operations: deletes the registry entries and purges everything cached for them. Clients still sending these hashes break.
`Service.persistedQueries`	Field	The manifest: every pushed operation, with `usage(startTime, endTime)`.
`Service.persistedQueryCount`	Field	How many operations the manifest declares, revoked ones excluded.
`Service.clientRegisteredQueries(first, after)`	Field	Operations clients registered over APQ that the manifest does not declare, one page of hashes at a time.
`Service.registryEntry(hash:)`	Field	One registry entry by hash, or `null`, with `origin` (`APQ` or `PUSHED`).

Result	Fields
`pushPersistedQueries`	`results`, one per document in the order sent: `hash`, `status` (`ADDED`, `ALREADY_PRESENT`, `ADOPTED`, `REJECTED`) and `reason` for a rejection; and `warning`, non-null when the service's deployed configuration is not known to turn persisted queries on. A `REJECTED` entry does not fail the batch.
`removePersistedQueries`	`results`, one per hash: `removed` (the registry held it and it was deleted), `purged` (everything cached for it was cleared; `false` means repeat the call for that hash) and `recorded` (the withdrawal was written to your manifest).

Telemetry

The request telemetry behind Insights. Every list takes a required `timeRange`, clipped to what your plan retains, and a `filter` (`RequestsFilterInput`); a filter or a `sort` your plan does not include is refused with `FORBIDDEN`.

Operation	Kind	Does
`Service.requests(first, after, timeRange, filter, sort)`	Field	Requests, newest first. The connection's `analytics` carries `timeSeries(groupBy, granularity, topGroups)`, `metrics(granularity)`, `summary` and `top(limit)`.

Operation	Kind	Does
`Service.request(id:)`	Field	One request by id.
`Service.operations(...)`, `Service.operation(id:)`	Field	The same for GraphQL operations.
`Service.errors(...)`	Field	Failures grouped by `groupBy`.
`Service.clients(...)`	Field	Traffic by client.
`Service.suggestions(field, search, limit)`	Field	Values seen for a filter field within the plan's telemetry retention, for autocomplete. `limit` defaults to `20` and is clamped to `100`.
`Service.logs(first, after, startTime, endTime, requestId, level, target)`	Field	The service's log lines.
`serviceRequestsStream(serviceId, after, filter)`	Subscription	New requests as they are recorded. Without `after`, only entries arriving after the subscription starts.
`serviceOperationsStream(serviceId, after, filter)`	Subscription	The same for GraphQL operations.
`serviceLogsStream(serviceId, after, level, target, requestId)`	Subscription	New log lines.

A request carries two ids. `RequestEntry.id` is its identity in GraphPilot, the one `Service.request(id:)`, `gpilot requests` and the portal's **Entry ID** use. `RequestEntry.edgeRequestId` is the edge platform's own id for the request (Fastly's request id), for correlating it with the edge platform's logs, for example in a support case. It is visible on every plan, and `null` on a request recorded before GraphPilot stored it.

A request stream polls for new entries and closes itself after 5 minutes without one; a log stream closes after 30 minutes. Resubscribe with the last cursor as `after` to continue. Subscriptions authenticate in `connection_init` (see authentication).

Signing key

Operation	Kind	Does
`Service.signingKey`, `Service.pendingSigningKey`	Field	The active and the pending key, redacted: `gpsk_`, 24 bullets and the key's last four characters. `null` without `service:edit` (and always for an API key), or when there is none.
`createService` payload `signingKey`	Field	The key a new service is issued, in full. The only time it is returned.

Operation	Kind	Does
`rotateServiceSigningKey(input: { service })`	Mutation	Generates a pending key. The payload's `pendingSigningKey` is the key in full, the only time it is returned.
`cancelServiceSigningKeyRotation(input: { service })`	Mutation	Discards the pending key.
`activateServiceSigningKey(input: { service })`	Mutation	Makes the pending key the active one.

`service` is the service's id. How rotation works is on request signing.

Custom domains

Operation	Kind	Does
`addServiceDomain(input: { serviceId, fqdn })`	Mutation	Attaches a hostname and starts its verification. Needs a paid plan and a domain your organization has verified. Returns the `ServiceDomain`.
`ServiceDomain.dnsChallenges`	Field	The DNS records to create for verification.
`ServiceDomain.certificateStatus`	Field	For a service platform hostname that gets a certificate of its own: `PENDING` while it is issued, `ISSUED` once the hostname serves TLS, `FAILED` when issuance gave up. `null` for every other hostname.
`retryDomainVerification(id:)`	Mutation	Retries verification of a failed domain. Returns the `ServiceDomain`.
`setPrimaryDomain(id:)`	Mutation	Makes an active domain the service's primary hostname. Returns the `Service`.
`removeServiceDomain(id:)`	Mutation	Detaches a domain. Starts a deployment that takes the hostname off the edge; the service keeps answering on it until that deployment finishes. Returns the `Service`.
`enablePlatformDomain(serviceId:)`, `disablePlatformDomain(serviceId:)`	Mutation	Turns the `<subdomain>.<label>.graphpilot.cloud` hostname on or off. It can be disabled, not removed, and only while the service has another active domain. Disabling also turns off an older `<subdomain>.graphpilot.cloud` kept beside it while a service is moved onto its label; enabling turns on only the current one. Returns the `Service`.

See custom domains for plans and verification.

`Service.ipRanges`

The IP ranges Fastly's edge network reaches your origin from, for an origin allowlist, which also has the example query and why to re-read them.

Field	Type	Meaning
`ipv4`	`[String!]!`	IPv4 ranges in CIDR notation.
`ipv6`	`[String!]!`	IPv6 ranges in CIDR notation.

`Query.clientIp`

`clientIp: String` is your public IP address as GraphPilot's edge saw the request, IPv4 or IPv6. It answers only a signed-in caller (a person or an API key) and is `null` for an anonymous one, and `null` too when the edge reported no address. It carries no cache lifetime, so no response holding it is stored. `gpilot requests --ip` reads it. It is not affected by `telemetry.client ip`, which changes what Insights stores: on a service that sets it to `"truncated"` or `"none"`, `RequestClient.ip` is the truncated address or `null`, and the geo and network fields that setting drops are `null` (`utcOffset` reads `0`).

Related

- API keys for creating keys and choosing their permissions.
- Purging through the API for the purge mutations in full.
- Deployments for what the diagnostics mean.
- Origin allowlist for keeping a firewall rule current.

REFERENCE

Performance and benchmarks

This page explains how the latency figures on the GraphPilot website are measured, what exactly they include, and how you can measure your own service the same way. What the cache does to earn those numbers is on how a request flows; what it can and cannot cache is on when a response is cached. It is published so every figure shown can be checked rather than taken on trust.

What is measured: p95 HTTP response time at the load generator

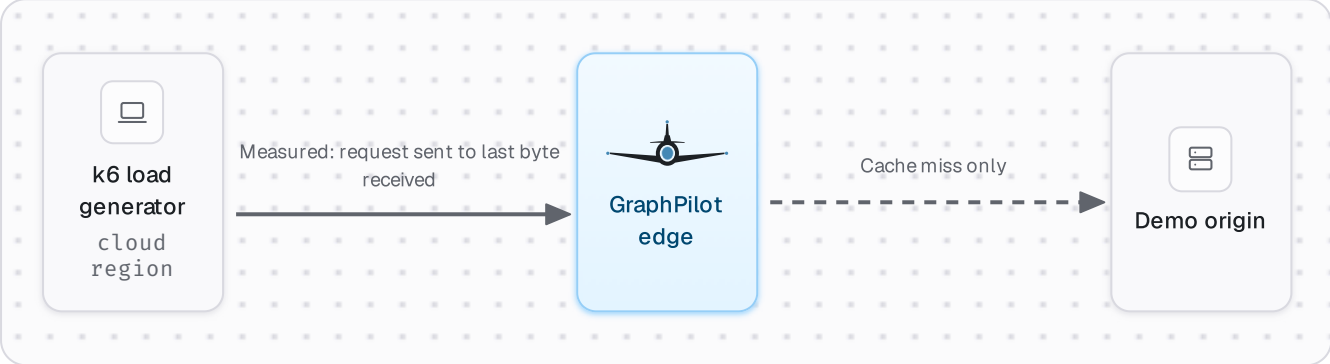
The latency figures on the GraphPilot website are the **p95 HTTP response time of cache hits**, measured by a k6 load test from three cloud regions (Europe, US East, Asia). The metric is k6's built-in `http_req_duration`.

`http_req_duration` is measured **at the load generator**, not at the edge. It is the sum of three k6 timings:

- `http_req_sending`: sending the request to the edge.

- `http_req_waiting`: waiting for the first response byte (time to first byte, TTFB).
- `http_req_receiving`: downloading the response body.

It therefore includes the network round trip between the load generator and the nearest edge location, the edge's own work, and the full body download. It excludes the DNS lookup and the TCP and TLS connection setup, which k6 reports separately; k6 keeps connections open between requests, so a VU pays that setup once rather than on every request.



The measurement starts and ends at the load generator.

Because the clock runs at the load generator, the figure is neither an edge-internal timing nor time to first byte. It is closer to what a client in a well-connected data center sees: a client on home Wi-Fi or a mobile network adds its own last mile on top.

How the benchmark runs

The benchmark runs against a dedicated GraphPilot service on the production edge, in front of a demo origin in a single European region. The demo origin answers a fixed set of deterministic, cacheable queries, so the response bytes are the same on every run.

Each size class (see Payload size classes) gets its own cache-hit run:

1. **Regions.** The load is split across three cloud regions: 40% Europe, 30% US East and 30% Asia. Each request is served by the edge location nearest its region, and the percentiles are formed across all three regions combined.
2. **Warm-up.** 5 virtual users (VUs) send the query for 15 seconds. The first request at each edge location fills the cache there.
3. **Measurement.** 20 VUs send the same query for 60 seconds, in a closed loop with no think time: each VU sends its next request as soon as the previous one has returned.
4. **Hit check.** Every request in the measurement phase is checked for status 200 and for the `gp-cache: HIT` response header, so a cache miss shows up as a failed check instead of passing as a hit.

The whole run lasts 75 seconds, well inside the 300-second `max_age` of the benchmark queries, so no entry expires mid-run.

`http_req_duration` is recorded by k6 for **every** request of the run, the warm-up included. The warm-up's first request per edge location is a cache miss; a handful of misses among thousands of requests sits above the p95 and barely moves it. Alongside it, each run records time to first byte and full response time for the measurement phase alone, which leave the warm-up out.

The same setup also runs cache-miss tests (10 VUs for 60 seconds after a 10-second warm-up, every request a unique operation so it reaches the origin) and a stress ramp (rising from 20 to 2,000 requests per second over two minutes, then one minute at the peak, 95% cache hits). Those runs exist to compare against and to check that cache-hit latency holds as the request rate rises; the website figures come from the cache-hit runs only. A cache miss reflects the distance to the origin and the origin's own speed, not the cache, and its numbers would say little about your origin.

Payload size classes

The benchmark uses five queries. Their response sizes are fixed by the demo origin and checked by a test, so a size class means the same bytes on every run. Sizes in KB use 1 KB = 1,024 bytes.

Class	Query selects	Response size	Represents
small	one item: `id name price rating`	101 B	a single-entity or detail query
medium	300 items: `id name price rating`	22,230 B (~22 KB)	a list query
typical	700 items: `id name price rating`	51,831 B (~50 KB)	the size a GraphQL p95 list or collection response commonly lands at
large	1,500 items: `id name price rating description tags`	390,029 B (~381 KB)	a heavy, data-rich response (the tail)
nested	4 stores, 16 departments, 80 products, 240 variants, five object levels deep	26,990 B (~26 KB)	query shape rather than size

The four flat classes measure size. `nested` measures shape: three of its levels omit `id`, so on a cache miss the edge adds the key field to the origin request and strips it from the response. On a cache hit the edge serves the stored bytes, so the shape costs nothing there.

On a cache hit, `http_req_duration` grows with the response size because the body download grows; the edge does no backend work for a larger response. That growth is bandwidth between the edge and the load generator, not cache lookup time.

The figures on the website

The homepage shows the p95 cache-hit response time per size class, from the benchmark run of 2026-10-02. Each measured value is rounded up to the next millisecond:

Class	Response size	Measured p95 (cache hit)	Shown
small	101 B	8.2 ms	9 ms
medium	~22 KB	10.0 ms	10 ms
typical	~50 KB	12.0 ms	12 ms
large	~381 KB	17.8 ms	18 ms

The `nested` class is measured but not shown on the homepage.

What each run records

Every run is recorded with:

- its kind (cache hit, cache miss or stress) and size class, start time and duration,
- the regions and their split, and the number of requests sent from each region,
- the number of VUs over time,
- k6's built-in HTTP timings for every request (`http_req_duration` and its parts, including `http_req_waiting`), and its error rate,
- time to first byte and full response time for the measurement phase alone,
- the outcome of the checks: status 200, and `gp-cache: HIT` for a cache-hit run or `gp-cache: MISS` for a cache-miss run.

The response size is not part of the k6 run. It is fixed per class by the demo origin, as listed in Payload size classes. The query text of each class is fixed as well.

Percentiles, not averages

The figures are **percentiles** rather than a single average, because averages hide the slow tail that users actually feel.

- **p50** (median): half of requests are this fast or faster.
- **p95**: 95% of requests are this fast or faster; only the slowest 5% are above it.
- **p99**: 99% of requests are this fast or faster.

A **p95** figure of 12 ms means 95% of the requests in that run had an HTTP response time of 12 ms or less. p95 is a common industry choice because it captures real-world consistency without being dominated by rare outliers.

"Reproducible demo benchmark", not an SLA

The latency figures on the GraphPilot homepage are a **demo benchmark run under controlled conditions**: a demo origin, fixed queries, and load generators in cloud data centers. They illustrate the effect of edge caching; they are **not a guaranteed service level**. Actual latency depends on geography, cache hit rate, query shape, response size, and the client's own network.

GraphPilot does not publish a fixed number of edge locations and does not guarantee a specific latency.

How to measure your own service

You can measure your own service the same way. The key is to **measure from stable cloud vantage points, not from a laptop over Wi-Fi**, to warm the cache before you measure, and to keep cache hits and cache misses in separate runs.

1. Break a single request into its phases

`curl` reports each phase, so you can see how much is connection setup, how much is waiting for the edge, and how much is the body:

```
curl -w "dns:%{time_namelookup} connect:%{time_connect} tls:%{time_appconnect} ttfb:%{time_starttransfer} total:%{time_total}\n" \
  -o /dev/null -s https://<your-service-hostname>/graphql
```

`ttfb - tls` (`time_starttransfer - time_appconnect`) approximates the round trip plus the edge's time to start answering; `total - ttfb` is the body transfer. Run it once over Wi-Fi and once over a wired connection to see your own last-mile contribution.

2. Choose the metric that answers your question

k6 reports both timings for every request:

- `http_req_duration` is the full response time, including the body download. It is the metric behind the GraphPilot website figures, and the one closest to what your clients wait for.
- `http_req_waiting` is time to first byte. It leaves out the body download, so it stays roughly flat across response sizes and shows how fast the edge starts answering. Use it when you want to compare cache performance across very different payload sizes without the bandwidth of the link in the number.

Both are measured at the load generator, so both include the network round trip to the nearest edge location. When you publish or compare a figure, name the metric it is.

3. Warm up, measure, then confirm each request was a hit

```
// k6: warm the cache, then measure cache hits only
import http from "k6/http";
import { check } from "k6";
import { Trend } from "k6/metrics";
import exec from "k6/execution";

const ttfb = new Trend("ttfb_ms", true);
const total = new Trend("total_ms", true);

export const options = {
```

```
  scenarios: {
    warmup: { executor: "constant-vus", vus: 5, duration: "15s" },
    measure: { executor: "constant-vus", vus: 20, duration: "60s", startTime: "15s" },
  },
};

export default function () {
  const res = http.post("https://<your-service-hostname>/graphql", body, params);
  if (exec.scenario.name === "measure") {
    ttfb.add(res.timings.waiting);   // time to first byte
    total.add(res.timings.duration); // full response, same as http_req_duration
    check(res, { "cache hit": (r) => r.headers["Gp-Cache"] === "HIT" });
  }
}
```

Keep the warm-up and measurement windows shorter than your query's `max_age`, so entries do not expire mid-run. Run cache misses separately: a miss reflects your origin's speed, not the cache's.

4. Measure from multiple regions and combine

A single location is not a global figure. Drive the load from **several regions** (multiple cloud regions, or a load-testing service with global load zones) and form the p50, p95 and p99 **across all of them**. Repeat across a few representative response sizes: time to first byte stays roughly flat, while the full response time grows with the payload.

Summary

- The website figure is the **p95 HTTP response time** (k6 `http_req_duration`) of **cache hits**, measured **at the load generator**.
- It includes the network round trip to the nearest edge location and the body download; it excludes DNS and connection setup.
- It is measured **from three cloud regions** (Europe, US East, Asia), after a **warm-up**, with every measured request **checked to be a cache hit**.
- It is reported per **payload size class**, rounded up to the next millisecond, and as a **percentile**, not an average.
- Homepage figures are a **reproducible demo benchmark**, not a guaranteed SLA or uptime commitment.

Related

- Locations
- How a request flows
- Cache statuses

REFERENCE

Glossary

Several of these words mean something narrower here than they do in general use. Where that is true, the entry says so.

Billing account

What pays for GraphPilot: it holds the plan, any add-ons and the usage. One billing account can fund several organizations, and plan limits such as the request quota are counted across all of them. Managed in the portal on an organization's **Billing** page. See billing.

Bucket

Which partition an entry belongs to: **public**, **private**, a **named** scope's, or **platform** (requests GraphPilot itself sent). A response that selected fields from several is keyed on all of them. The bucket is part of what a lookup matches on, so entries in different buckets never meet, and it is chosen when the response is stored. See when a response is cached.

Bypass

A policy expression that, when it matches, sends a request straight to your origin without consulting the cache and without storing the answer. Reported as `PASS` with `CACHE_PASS_BYPASS_POLICY`.

Cache key

What identifies one cache entry. It is **not** the same thing as a surrogate key. See the cache key.

- **HTTP:** the request method and the path, query string included, such as `GET:/products/42?color=blue`. A request that carries a body is keyed by a SHA-256 of the body as well, such as `POST:/search#<sha256 of the body>`. Scheme and hostname are left out unless the service sets `cache.http_key.host = "include"`, which keys by the full URL, such as `GET:https://catalog-api.k7m2qp.graphpilot.cloud/products/42`.
- **GraphQL:** the path and the query string without the GraphQL parameters, then a SHA-256 of the query text exactly as sent, plus the operation name and a hash of the variables when present, such as `/graphql?locale=de#<sha256>#GetProduct:<variables hash>`. The method is not part of it, so `POST`, `GET` and `QUERY` on one route share one entry. Telemetry and the `ck:` surrogate key carry it without the route. Up to runtime 1.4.0 the key has no route: it starts at the SHA-256.

Caller

The identity behind one request: the person, token or API key it carries. Two callers using the same client can see different data, which is why a private cache entry is partitioned per caller. Compare client.

Client

The application that sends requests to your service: your web app, a mobile app, a partner integration, the `gpilot` CLI. GraphPilot recognizes a client only if you declare it in a client profile. Compare caller. See telling your clients apart.

Customer

In examples, one of **your** customers: a tenant of the application you run behind GraphPilot, as in `x-customer-id` or “one shared entry per customer”. Not a GraphPilot account; that is an organization.

Deployment

One version of a service’s schema and configuration, built and rolled out to the edge. Created with `gpilot deploy` or from the portal, which lists them under **Deployments**. See deployments and Deployments in the portal.

Deployment build

The step at the start of a deployment that compiles your schema and configuration into the package the edge runs. It checks both, derives the surrogate key and cache rules from the schema, and reports anything it refused or ignored as diagnostics, which the portal shows under findings.

Discriminator

The value a `public` or `private` policy expression resolves to, which partitions an entry inside its bucket. It never reaches your origin or the cached response.

Edge location

One of the points of presence of Fastly’s edge network, which GraphPilot runs on. Each holds **its own cache**, so a response stored at one is not stored at another. Purges, by contrast, reach all of them. See edge locations.

Effective policy

The caching policy that applies to one response, resolved across everything the operation selected. For `@cacheControl` this is the minimum `maxAge` across the selection, and the union of every selected field’s discriminators, which is the narrowest audience all of them allow. See lifetime from the schema.

Empty-list key

A surrogate key of the form `![Typename]`, tagged on a response whose list of that type came back empty. It exists so that creating the first object of a type can invalidate the cached “there are none” answer. See automatic invalidation.

Entity key

A surrogate key naming one object, such as `Product:id:1`. Derived from key fields found **in the response body**, and one of only two kinds a mutation purges automatically.

Freshness lifetime

How long an entry may be served without going back to your origin. Comes from `s-maxage` or `max-age` on the origin's `Cache-Control`, or from `@cacheControl(maxAge:)` when the origin sent no `Cache-Control` header, or one that says nothing about storing or ageing the response (a bare `private`, `public` or `no-transform`; `must-revalidate`, `immutable` or a `stale-*` directive on its own does count). Often called the TTL. See how long an entry lives.

Insights

The portal's view of your request telemetry: every request, its cache status, reason code, latency and origin details, with filters and charts. Part of every plan; plans differ in list columns, filters and how far back you can look. What is stored is on what Insights needs.

Lean mode

What every service a billing account funds switches to when the account is lean: on Glider (Free), once its included request units for the period are used up; on any plan, while a renewal payment is overdue or when GraphPilot pauses the account. Edge caching is withdrawn: every request goes to your origin and nothing is stored, and no `gp-cache` header is sent on responses the cache would otherwise have looked up. Authentication, request signing and purging keep running, and request telemetry is not stored. Lean mode keeps the services serving; on Glider, only fair use can lock the account for the rest of the month. See quotas.

Log line

A warning or error GraphPilot writes while handling a request, carrying a stable code such as `AUTH_KEY_MATERIAL_UNAVAILABLE`. A decision or refusal code (`CACHE_STORED`, `INGRESS_RESERVED_HEADER_DROPPED`, `GRAPHQL_INVALID_QUERY`, …) is not written as a line: it is recorded on the request itself, with the detail the line used to carry (the blocking header or cookie, for example), and listed under **Edge codes** in the **Overview** tab of a request's detail in Insights. When these docs say a code appears "in your logs", they mean either of the two. Log lines are not shown in the portal; read them through the API's `Service.logs`. Both are stored under the same rules as request telemetry (see Insights). See codes.

Organization

The workspace your services, members and API keys belong to. Members are invited to an organization and act in it with a role. One billing account can fund several organizations. See organizations and members.

Origin

The server your service forwards requests to. Its URL is a property of the service, not of the configuration file.

- **HTTP:** any API that answers over HTTP. Caching follows its `Cache-Control`, and its `Surrogate-Key` header names keys.
- **GraphQL:** a GraphQL API on the paths in `request.graphql_paths`. Its schema adds lifetimes, scopes and keys through directives.

Only the URL's scheme, host and port are allowed; without a port, GraphPilot connects to port `443` for `https://` and `80` for `http://`. See TLS to your origin.

Persisted query

An operation the service already knows, sent by the client as a hash instead of a document. Either registered by clients at runtime (APQ) or pushed from a manifest you control. See persisted queries.

Policy expression

The small language used by `cache.policy.public`, `private`, `vary[].value`, `bypass`, client profiles and conditional response filters. Reads verified claims, headers, cookies and query parameters; returns a string or nothing. See the reference.

Provider

A configured way of verifying one kind of credential, named so a policy expression can address its claims as `auth:<name>.<claim>`. Two types exist: `jwt` and `api_key`. See authentication.

Purge

Removing entries from the cache by naming a surrogate key. A **hard** purge removes the entry; a **soft** purge marks it stale so a revalidation serves rather than a miss. Purges reach every edge location.

Selection set expansion

Injecting the fields needed to derive surrogate keys into a query, then stripping them from the response before it reaches the client. It is why clients never have to select `id` for invalidation to work. See selection set expansion.

Service

One origin behind GraphPilot, with its own hostname, configuration and cache: an origin URL, a service hostname, a configuration and its deployments. An organization can have many.

A GraphQL service deploys with `gpilot.toml` and a `schema.graphql`, which the deployment build reads for directives and key fields.

A plain HTTP service deploys with `gpilot.toml` alone. Without a schema, GraphQL features are off and every path is served as plain HTTP.

Hostname label

The six characters between a service's subdomain and `graphpilot.cloud` in its service hostname, such as `k7m2qp` in `checkout-api.k7m2qp.graphpilot.cloud`. GraphPilot generates one per organization; it is the same for every service of the organization, never changes, and is never given to another organization. See the hostname label.

Service hostname

The address GraphPilot issues for a service, `https://<subdomain>.<label>.graphpilot.cloud`, where `<label>` is the organization's hostname label. A service created before labels existed keeps `https://<subdomain>.graphpilot.cloud`, or answers on both while it is moved onto its label. Your clients send their requests there instead of to your origin, and the portal shows it for each service. A custom domain can stand in front of it.

Skip

A response the cache looked for, missed, fetched, and then declined to store, with a `CACHE_SKIPPED_*` reason. Always reported as `PASS`, because nothing was stored, whether the request or the response ruled caching out or storing it failed inside GraphPilot. Distinct from a lookup-time **pass** (`CACHE_PASS_*`), where the response was neither served from nor written to the cache.

Surrogate key

A label attached to a cache entry so that entries can be removed as a group later. Several kinds exist, and the kind decides whether a mutation purges it automatically. See surrogate keys.

Vary rule

A `[[cache.policy.vary]]` entry that derives a value and writes it onto the request as a header. Every response is partitioned on it, plain HTTP included, whether or not your origin's `Vary` names the header. A rule marked `scope_only` is the exception: a response your schema decided is partitioned on it only when its scope names it in `@vary`. A request the expression does not resolve for is stored as the variant without the header. See who shares an entry.

REFERENCE

Runtime versions

Every service runs a version of the GraphPilot proxy runtime. By default it follows the stable version and moves to a new one on its next deployment. You can fix a service to one version in the portal (Service settings, Runtime) or with `gpilot service runtime pin <version>`. Changing the pin rebuilds the service with the version the pin resolves to, unless the service already runs that version and no deployment is in progress. In the portal you can

also save the pin without deploying, and the service's next deployment uses it. Either way it needs `service:edit` and `deployment:edit`.

You can also pin a single deployment in the service config:

```
[runtime]
version = "0.33.1"
```

This key pins only the deployment that `gpilot deployment create` makes with that config. The CLI checks the version, removes it from the uploaded config and sends it as the deployment's runtime version. It is not the service's pin: a redeploy from the portal, or a change of the pin, does not carry it. To keep the service itself on a version, pin it in the portal or with `gpilot service runtime pin`.

A withdrawn version can no longer be chosen or deployed. A service that runs one is flagged in the portal. What moves it off depends on its pin:

- **Not pinned, or pinned to a supported version:** redeploy it. The deployment uses the pinned version, or else the stable version, which is never a withdrawn one.
- **Pinned to a withdrawn version:** every deployment of the service that names no version is refused, including one from the portal that keeps the default **Proxy version** and a `gpilot deployment create` without a `[runtime]` `version`. The service is not moved to the stable version instead. A deployment you request is refused before it is created, with `RUNTIME_VERSION_WITHDRAWN` in the API error, so `gpilot` exits with `4`. A redeploy that GraphPilot starts on its own is created and fails before anything is built, with the same message on the deployment. Nothing is built or activated for a refused deployment, so a deployed service keeps serving the version it already runs. Pin a supported version, or remove the pin with `gpilot service runtime unpin` or in the portal; either change redeploys the service, unless you choose to apply it with the next deployment in the portal. A pin on a version that is no longer in the catalog is treated the same way, and the portal flags it on the Runtime settings page and in the deployment dialog.

A deployment that names a supported version itself, with `[runtime]` `version` or under **Proxy version** in the portal's deployment dialog, is not refused, whatever the service's pin says.

The version list is filled in from the GraphPilot API when this page is served. If you see this sentence, run `gpilot runtime versions` instead.

REFERENCE

Versioning and deprecation

GraphPilot has three parts that change on their own schedule: the edge runtime your services run on, the `gpilot` CLI, and the GraphPilot API behind the portal and the CLI.

The edge runtime

Every service runs a version of the GraphPilot proxy runtime. A deployed service keeps the runtime it was built with until its next deployment, so a new runtime version never reaches a service on its own.

- **Unpinned,** a service follows the stable version and moves to a new one on its next deployment.
- **Pinned,** it stays on the version you chose, in the portal (Service settings, Runtime) or with `gpilot service runtime pin <version>`. One deployment can also name a version in `[runtime]`.

A version is in one of three states: **active** (offered), **deprecated** (still served, no longer offered), or **withdrawn** (no longer offered or deployable). A service on a withdrawn version is flagged in the portal; what moves it off depends on its pin. See runtime versions, which also lists every version and its state.

The CLI

Before any command that talks to the GraphPilot API, `gpilot` reads a published version manifest and checks the minimum version GraphPilot supports. A new release is normally published as the new minimum. Behind the latest release, the CLI prints a notice; below the minimum, it refuses API commands and exits with `6`. Commands that never leave your machine are not checked. See staying up to date, including what to do with a pinned CLI in CI.

The GraphPilot API

The GraphPilot API has one endpoint and no version number. Most changes add types, fields and arguments, which an existing client does not notice. When a field is replaced, the old one is marked deprecated in the schema, with a `deprecationReason` naming its replacement, and keeps answering; for example, `Deployment.creatorApiKey` is deprecated in favor of `Deployment.requestedBy`. A change that removes something or changes what it means, such as a new required argument or a stricter permission, is a breaking change.

Related

- Runtime versions
- Installing the CLI
- GraphPilot API