QueryAPIGate

Security pack

QueryAPIGate 0.34.2: architecture, threat model, data protection, security policy, deployment, release verification and answers to a security questionnaire, for an architecture or vendor-risk review.

QueryAPIGate is self-hosted software, source-available under FSL-1.1-MIT. It runs inside your network; nothing is sent to the project.

Built from the documentation at 0.34.2. The latest documents: github.com/AnanthaRajuC/QueryAPIGate

Contents

1. Architecture overview
2. Security questionnaire answers
3. Threat model
4. Where data goes
5. Security policy
6. Production deployment
7. SBOM and release verification

QueryAPIGate architecture overview

For enterprise architects and security reviewers. Describes release **0.34.2**.

This document covers how QueryAPIGate is built and where its controls sit. The reasons behind each control, and where each one stops, are in THREAT_MODEL.md. What is kept, where and for how long is in DATA_PROTECTION.md. The hardening checklist and vulnerability reporting are in SECURITY.md. Production setup is in DEPLOYMENT.md. Answers to common questionnaire items are in QUESTIONNAIRE.md.

1. What it is, and the trust boundary

QueryAPIGate is a self-hosted gateway, source-available under FSL-1.1-MIT. It turns saved SQL queries into REST endpoints, MCP tools for AI agents, live server-sent events and exports. A saved query is versioned, and its parameters are typed and bound to the driver rather than written into the SQL. Every call goes through one governed path: authentication, limits, grants, a SQL guard, masking and a run-history record. It is a Python (Flask) application. The admin UI, the Console, is a React single-page app that ships inside the same package.

Trust boundary. QueryAPIGate runs inside the customer's network, on infrastructure the customer operates. No part of it is a hosted service. It connects outward only to systems the deployer configures:

- the databases and file locations that connections name;
- its own metadata store (SQLite or PostgreSQL) and, optionally, Redis;
- an identity provider, for Console sign-in, signed-in app users and MCP clients;
- secret providers: files, HashiCorp Vault, and AWS (experimental);
- export destinations (object-storage buckets or allowed local folders);
- a SIEM, an OpenTelemetry collector and a Prometheus scraper.

QueryAPIGate does not terminate TLS. A reverse proxy or ingress in front of it does that, and is required once any key exists. QueryAPIGate is not a network perimeter or a web application firewall: network policy, a VPN or a proxy still decides who can reach it at all.

2. Components

All components come from one image (`ghcr.io/anantharajuc/queryapigate`) and one code base. They share the metadata store.

Component	Process and port	What it serves
Main server	`queryapigate serve`, or gunicorn in the image; `:5000` (`QUERYAPIGATE_PORT`)	Runtime routes (`/q/<name>`, `/execute_sql`, `/catalog`, `/connections/<name>/schema`), `GET /events` for a few streams, the Management API, the Console, `/docs`, `/openapi.json`, `/metrics`, `/health`

Component	Process and port	What it serves
Management API	On the main server, at `/api/v1`	Versioned, resource-oriented administration: connections, saved queries and versions, keys, roles, collections, history, audit log, settings, exports. Administrators only, checked by capability per role.
Console	On the main server, at `/console`	Static React build, a client of the public HTTP APIs only. It bundles everything it runs.
MCP server	`queryapigate mcp`; `:5001` (`QUERYAPIGATE_MCP_PORT`), at `/mcp`	Saved queries as tools for AI agents. It uses the same authentication and grant checks as REST. Results are capped at `QUERYAPIGATE_MCP_MAX_ROWS`.
Events server	`queryapigate events`; `:5002` (`QUERYAPIGATE_EVENTS_PORT`)	Server-sent events for many clients, read from the store's event log, plus its own `/metrics` and `/health`
Export CLI	`queryapigate export` (local file), `queryapigate exports run` (saved exports to a destination)	Runs a saved query to a file or bucket. Scheduling comes from outside, such as host cron or a Kubernetes CronJob.
Other CLI	`queryapigate init`, `backup`, `migrate-to-postgres`, `rekey`, `audit verify`, `history forget`, `admins ...`	Operations on the store. Anyone with a shell and store access is outside the application's access model.

The image runs as a non-root user (UID 1000), with one gunicorn worker and eight threads per container. To add capacity, add containers.

3. Diagram


CALLERS
FRONT DOORS (inside the customer network)
Apps, partners
Browsers
(Console, SSE)
AI agents (MCP)
Reverse proxy / ingress
TLS is terminated here
cron / CronJob
export CLI
main
:5000
REST
/api/v1
Console
MCP
:5001
/mcp
events
:5002
SSE
steps 1-2 only,
then reads the
event log
GOVERNED CORE - the same code in each process:
1 identify caller
2 apply limits
3 check grant
4 validate parameters
5 guard the SQL
6 run it
7 mask and format
8 record it
Your databases
PostgreSQL, MySQL, SQLite, DuckDB, Click-House, ...
Files through DuckDB, only allowed_paths (disk, S3, GCS, R2, web)
Export destinations (bucket or allowed folder)
Store, used by every process: SQLite or PostgreSQL - config, key hashes, history, event log, audit log
Redis - optional; needed for several instances
OUTSIDE SYSTEMS
IdP ..... OIDC (Console), JWKS (app users), OAuth 2.1 (MCP)
Secrets ..... files, Vault; AWS (experimental)
SIEM ..... receives the audit log: syslog over TLS, or OTLP logs

```
Prometheus ..... scrapes /metrics (bearer token when set)
OTLP collector . receives traces (experimental)
```

4. The request path

Every front door applies the same governance code (authentication, rate limits, metrics). A saved-query call then passes eight steps. This is a model of what is checked, not a strict order: the server-wide rate limit applies before the key is checked, and a column masking can't trace is refused before the query runs. Codes are from the API.md error catalogue; `code` is under the compatibility promise.

1. **Identify the caller.** Admin key or scoped key (`X-API-Key`; the admin key compared in constant time, scoped keys looked up by SHA-256 hash), an administrator's token, a JWT (`Authorization: Bearer`), or on MCP an OAuth access token. Fails `401 unauthorized`; on MCP, a token without the configured scope is `403 insufficient_scope`.
2. **Apply its limits.** Server-wide per-address and per-key rate limits (`429 rate_limited`), usage quotas, checked before the response cache (`429 quota_exceeded`), `max_concurrent` (`429 too_many_concurrent`). `allowed_ips` and `expires_at` fail like a wrong key (`401 unauthorized`), so the caller learns nothing about why.
3. **Check the grant.** The key's `connections`, `collections` or `queries` grant must reach the query or connection: `403 connection_forbidden` or `403 forbidden`; `403 connection_inactive`; `404 query_not_found` or `404 not_published`. On the Management API: `403 admin_only` or `403 role_forbidden`.
4. **Validate parameters.** Each value is checked against its declared type and rules, then bound to the driver, never spliced into SQL. A `from_claim` parameter comes from the verified token, never the request. `400 param_required`, `400 param_invalid`, `400 param_from_claim`, `403 sign_in_required`, `403 claim_missing`.
5. **Guard the SQL.** A dialect-aware **regex classifier** enforces one statement (`400 multiple_statements`) and read-only unless writes are allowed, including writes hidden behind a read-only keyword (`403 read_only`, `403 write_op_not_allowed`). A **sqlglot parser** enforces `allowed_tables` (`403 table_not_allowed`; `403 table_check_failed` when the SQL can't be analysed; `403 table_check_unsupported` on H2 and JDBC). DuckDB file access outside `allowed_paths` is `403 path_not_allowed`.
6. **Run it.** A pooled connection runs the bound statement under a time limit. PostgreSQL, MySQL, ClickHouse and SQLite run read-only queries in read-only sessions; DuckDB, H2 and JDBC are opened read-write, so there the guard is the only lock. A query with `cache_ttl` may be answered from the response cache. `504 query_timeout`, `500 query_failed`, `502 connection_failed`, `503 secret_unavailable`.
7. **Mask and format.** Masking rules on the key, role or export apply to source columns; result columns are traced to them with sqlglot lineage, not matched by name. Anything untraceable is refused, never served unmasked (`403 mask_unresolvable`; `500 masking_unavailable` without sqlglot). The result is serialised in the requested format (`400 invalid_format`).
8. **Record it.** The run is written to history and the event log in one batch transaction, and metrics are updated. History keeps parameter names, never values. Configuration changes go to the hash-chained audit log with the administrator who made them. If the write queue is full, a run is dropped from both.

A scoped key calling a saved query never receives the database's own error text, which can reveal schema; the server log has it.

5. State and data flows

The full inventory is DATA_PROTECTION.md; a test fails if it misses a store table, a retention setting or an outbound flow.

The store (`queryapigate.db` on the `/data` volume, or PostgreSQL) holds connections (passwords Fernet-encrypted with `QUERYAPIGATE_SECRET_KEY`, or held as references), saved queries and every version's SQL, API keys and admin tokens as SHA-256 hashes only, roles, administrators, SSO sessions by hash, run history, the event log, usage counters, the audit log with its hash chain and SIEM delivery cursors, agents' write requests (30 days), export destinations and watermarks, the instance list, and server-generated keys such as masking's. The admin, Fernet and audit keys are not in the store.

Redis (optional; required for several instances) holds the response cache (masked rows of queries that set `cache_ttl`, for that long), rate-limit counters (the current window), quota counters (40 days) and running-query counts. If Redis fails, the cache misses and rate limits fall back to per-process counting: still enforced, with an alert.

The event log keeps each run as `/events` sends it for `QUERYAPIGATE_EVENTS_RETENTION_HOURS` (default 24). A key sees only its own events, none from before it was created; the admin key and data administrators see all.

Passes through, not kept: result rows (except in the response cache and exports), parameter values, and resolved secrets (memory only, for their lease or `QUERYAPIGATE_SECRET_CACHE_SECONDS`).

Sent outward:

To	What
Log (stdout), access log	Requests without query strings; SQL or its SHA-256; database error messages, which can quote a value
SIEM	Every audit entry with `seq` and `hash`, by syslog over TLS (mutual TLS optional) or OTLP logs
OTLP collector	Route, query name, rows, caller name and address; never parameter values, rows, statements or credentials
Prometheus	Counts and timings by connection, query and key name
Export destinations	A saved query's rows, after the export's masking
MCP clients	Rows after masking, up to the row cap
Secret providers	Requests for the secrets connections name

`queryapigate history forget` erases a person's runs, events and usage counts; the audit log records the erasure under the caller's SHA-256.

6. Deployment shapes

All three run the same image and code.

	Single instance	Several instances	Kubernetes (Helm)
Store	SQLite on a volume, or PostgreSQL	PostgreSQL	SQLite (one pod) or PostgreSQL
Redis	Optional	Required	Required for more than one pod
Live events	`GET /events` on the instance	`queryapigate events`, routed by the proxy	Events pods, routed by the ingress
Availability	Restart on failure	Load balancer skips an unhealthy instance	Rolling updates (`maxUnavailable: 0`), PodDisruptionBudget
Tested in CI	Full suite (about 1,300 tests) across Python versions; backup and restore (`test_backup_restore.py`); stores from every release since 0.7 (`test_upgrades.py`); the Compose demo	Three real servers on one store (`test_multinode.py`); previous PyPI release on a store the current code upgraded (`test_rolling_upgrade.py`); full suite on PostgreSQL with Redis 7	`helm lint` (Helm 3 and 4); rendered security contexts, routes and refusals; plain manifests compared; on kind, the PostgreSQL shape installed and upgraded under load with no failed request, and the SQLite shape's data surviving a pod restart

A further CI job runs the whole suite in Red Hat UBI 9 under OpenSSL's FIPS provider.

The chart's pods run non-root (UID 1000), read-only root filesystem, all capabilities dropped, RuntimeDefault seccomp, no service account token; the server writes only to `/data` and `/tmp`. `helm install` refuses more than one pod on SQLite or without Redis, a secret setting in `config`, and any value not in the schema. `/metrics` is token-protected by default; NetworkPolicy is off until the deployer supplies the cluster's rules.

7. Identity integrations

- **Administrators:** four fixed roles (owner, admin, developer, auditor), personal admin tokens stored as hashes with optional expiry. The shared `QUERYAPIGATE_API_KEY` remains a break-glass owner key; once a named owner exists, each use is audited and alerted.
- **Console SSO (OIDC):** authorization code flow with PKCE (S256) as a confidential client, so the browser never holds a token. State, nonce and verifier are server-side, single-use, valid ten minutes; the ID token's signature, issuer, audience, `azp`, nonce and expiry are checked. People are bound by `sub`; roles come from IdP groups at each sign-in. Session cookie `__Host-`, `HttpOnly`, `Secure`, `SameSite=Strict`, stored as a hash, paired with a CSRF token. `QUERYAPIGATE_SSO_ONLY` removes personal tokens from SSO users. Tested against Keycloak 26.
- **App users (JWT/JWKS):** RS256 against a JWKS, or HS256 with a shared secret. `exp` required; `iss` / `aud` enforced (mandatory with a JWKS); HMAC and public-key families can't be mixed; `none` never accepted. Grants come from a role, read live. A JWT can't be revoked before it expires.

- **MCP clients (OAuth 2.1):** the MCP server is a resource server. A token's `aud` must be the MCP server's URL and it must carry the configured scope; the algorithm comes from the provider's published key, never the token header; provider metadata is fetched over `https://` only and must name the configured issuer (RFC 8414). REST and MCP refuse each other's tokens. A caller's token never reaches a database.
- **Secrets:** `${VAR}`, `${file:...}`, `${vault:...}` and short-lived Vault database credentials (stable since 0.34), each reachable only through deployer allow-lists checked when saved and when used; no reference can reach the server's own settings or credentials. `${aws-sm:...}` and RDS IAM tokens are experimental. The chart supports Vault's Kubernetes auth and IRSA.

8. Security controls summary

Control	Where enforced	Reference
Authentication on every request; open-access mode only while no key, administrator or JWT is configured	Request hook in each front door (`governance.py`)	THREAT_MODEL.md
Key and token secrets stored as SHA-256 hashes; rotation with a bounded grace period	Store; `/api/v1/api-keys`	ADR 0009
Scoped grants (connections, collections, queries, writes, tables, IPs, expiry)	`apikeys.py`, before execution	THREAT_MODEL.md
Administrator roles and capabilities; every `/api/v1` operation mapped	`adminroles.py`, before any route runs	ADR 0003
Bound parameters	`sqltools.bind_parameters()`, driver paramstyle	THREAT_MODEL.md
Single-statement and read-only guard (regex, dialect-aware, fuzz-tested)	`sqltools.validate_sql`	THREAT_MODEL.md
Read-only database sessions	PostgreSQL, MySQL, ClickHouse and SQLite drivers	THREAT_MODEL.md
`allowed_tables` (sqlglot)	`tableguard.py`	THREAT_MODEL.md
Column masking by lineage, failing closed	Result path, cache key, exports	ADR 0005
DuckDB file access locked to `allowed_paths`	DuckDB connection configuration	THREAT_MODEL.md
Rate limits, quotas, concurrency, time and row limits	Each front door; Redis across instances	ADR 0011
Agent writes need a named person's approval, bound to caller, version and arguments	MCP server	ADR 0008
Passwords encrypted at rest (Fernet), rotatable	Store; `queryapigate rekey`	THREAT_MODEL.md
Secret-reference allow-lists	Checked when a reference is saved and when it is used	ADR 0015

Control	Where enforced	Reference
Tamper-evident audit log (HMAC chain), streamed to a SIEM	Store transaction; delivery cursor and lease	ADR 0012, ADR 0019
Security headers (nosniff, `no-store`, CSP, `no-referrer`, optional HSTS); SRI on `/docs`	Every response	ADR 0010
CORS off unless origins are set	`cors.py`	THREAT_MODEL.md
`/metrics` behind a bearer token	Main and events servers; on by default in the chart	DEPLOYMENT.md
FIPS-approved algorithms only (test-enforced)	Code; validated module comes from the platform	DATA_PROTECTION.md
TLS termination	Customer's reverse proxy or ingress	DEPLOYMENT.md

QueryAPIGate does not defend against these, and the deployer has to cover them:

- **Administrators' data reach.** An owner, admin or developer can run any SQL on every connection. The database account behind each connection is the real limit.
- **Gaps in the SQL guard.** The guard is defence in depth, not a substitute for least-privileged database grants.
- **Network exposure.** Network perimeter controls are the deployer's.

9. Supply chain

From 0.31, every image (both variants, multi-arch) and the Helm chart is signed keyless with cosign, by digest, by the repository's `docker.yml` release workflow on the release tag. The signature is recorded in Sigstore's public log. Each release is signed in two forms: the Sigstore bundle and the classic `.sig` tag, which admission controllers need. Releases before 0.31 are not signed.

- **Provenance and SBOM:**
  - images carry SLSA provenance, an SPDX SBOM and a GitHub artifact attestation naming the commit;
  - the chart has a GitHub attestation;
  - PyPI releases use trusted publishing, with PEP 740 attestations.
- **Pinned builds:**
  - base images are pinned by digest;
  - every GitHub Action is pinned by commit SHA, and a test refuses a `uses:` without one;
  - downloaded tools are checked against their published checksums;
  - Dependabot proposes updates as reviewed commits.
- **Verifying:** `cosign verify` with the workflow identity and the GitHub OIDC issuer, `gh attestation verify`, the SBOM, and an admission policy that admits only signed images. Sigstore policy-controller and Kyverno are covered. Guide 59 walks through each one. SECURITY.md has the summary.

A signature shows which workflow and tag built a release. It does not show that the code is correct, and builds are not reproducible byte for byte (ADR 0018).

10. Stable and experimental features at 0.34.2

Stable features are under the compatibility promise: they change only by addition, and are renamed or removed only through the deprecation process. Experimental features are outside it.

Status	Feature	Since / reason	Reference
Stable	PostgreSQL, MySQL, SQLite, DuckDB and ClickHouse connections	Tier 1	README
Stable	Per-key concurrency, time and row limits	0.24	ADR 0011
Stable	Audit log hash chain and `audit verify`	0.25	ADR 0012
Stable	Access log without query strings; `history forget`	0.29	ADR 0016
Stable	Helm chart	0.30	DEPLOYMENT.md
Stable	OAuth for the MCP server, Console SSO, MCP write tools, usage quotas, configuration as code, descriptions, agent usage	0.30	ADR 0017
Stable	Signed releases	0.31	ADR 0018
Stable	Audit log streamed to a SIEM	0.32	ADR 0019
Stable	Live events; `${VAR}`, `${file:...}`, `${vault:...}` and Vault database credentials	0.34	ADR 0021
Experimental	H2, generic JDBC and MongoDB connections	`allowed_tables` doesn't work on them; gaps per type	ADR 0017
Experimental	Tracing with OpenTelemetry	Span conventions still settling. What never goes into a span is already a promise.	ADR 0010
Experimental	Secrets and credentials from AWS (`${aws-sm:...}`, RDS IAM)	Tested against a stand-in, not AWS	ADR 0021

Vendor security questionnaire: pre-filled answers

These are answers to the questions a vendor security questionnaire (SIG Lite, CAIQ, or a customer's own) usually asks, for **QueryAPIGate 0.34.2**. QueryAPIGate is a self-hosted, source-available (FSL-1.1-MIT) gateway that turns saved SQL into governed REST, MCP, event and export endpoints.

How to read them:

- The answers describe **the software**: what it does, what it stores, and where its controls stop. QueryAPIGate runs inside your environment, so the answers about **your deployment** (which features are on, who operates it, how the host and network are secured) are yours to give. In this document, **"Deployer"** means you, the customer running QueryAPIGate.
- Items marked **Deployer:** name a control that is your responsibility, or that depends on how you configure it.
- Each answer cites its source. Files in this pack sit side by side (`THREAT_MODEL.md`, `DATA_PROTECTION.md`, `SECURITY.md`, `DEPLOYMENT.md`, `ARCHITECTURE.md`). Other sources are linked in the project's repository.
- Where a source didn't support a claim, the claim was left out. To check the answers against your own running server, see guide 57.

---

1. Product and deployment model

Is the product delivered as a hosted service (SaaS) or as software the customer runs?

Answer: It is software the customer runs. It is not a service the project hosts. It is published as a container image (`ghcr.io/anantharajuc/queryapigate`, plus an `-h2` variant), a Helm chart and a Python package, and it runs on the Deployer's own infrastructure.

Source: DEPLOYMENT.md#1-the-image, DEPLOYMENT.md#kubernetes-helm

---

Does the product send any data to the vendor, such as telemetry, usage statistics or licence checks?

Answer: No. Every place data leaves the process is listed in the data-flow inventory, and a test fails if the code adds a path the inventory doesn't list. Every destination on that list is infrastructure the Deployer configures: their log platform, trace collector, Prometheus, SIEM, export buckets, secret providers, and the AI agents and clients they authorise. The project's trust page says this as: no telemetry, no licence server, nothing sent to the project. One exception happens in the browser, not the server: the `/docs` page loads one pinned Swagger UI release from a CDN, with Subresource Integrity hashes.

Source: DATA_PROTECTION.md#where-data-is-sent, THREAT_MODEL.md#transport, ADR 0010

Where is customer data processed and stored?

Answer: Only where the Deployer runs it. Query results pass through the process to the caller and are not kept, except in two features the Deployer turns on: the response cache and exports. Configuration, run history and the audit log are kept in a "store", which is either a SQLite file (`queryapigate.db`) on a volume or a PostgreSQL database that the Deployer provides.

Source: DATA_PROTECTION.md#what-passes-through-and-isnt-kept, DATA_PROTECTION.md#the-store

---

2. Identity and authentication

What authentication methods does the API support?

Answer: The API supports four kinds of credential:

- the shared admin key (`QUERYAPIGATE_API_KEY`);
- named administrators' admin tokens (`qagadm_...`);
- scoped API keys (`sk_...`), sent in the `X-API-Key` header;
- for applications whose users sign in, a JWT in `Authorization: Bearer`, verified against a shared secret or the identity provider's JWKS.

The MCP server accepts the same keys, or OAuth 2.1 access tokens.

Source: THREAT_MODEL.md#authentication-one-shared-secret-or-many-scoped-ones

---

Is authentication required by default?

Answer: No. If no API key is set, no scoped keys or administrators exist and JWT is off, every caller is treated as an unrestricted administrator. This "open access" mode is deliberate, for local trials, and is documented as unsafe for anything reachable beyond a machine you control. The `open_server` alert (critical) reports this state. Authentication is required once any scoped key or administrator exists, or when JWT is on.

Deployer: set `QUERYAPIGATE_API_KEY` before the server can be reached.

Source: THREAT_MODEL.md#open-access-mode-a-deliberate-default-worth-understanding, API.md, Alerts

---

How are API keys and tokens generated and stored?

Answer: The server generates scoped keys and admin tokens from the operating system's random source and shows each one once. Only its SHA-256 hash is stored, so a lost key can't be recovered; it can only be revoked and replaced. The shared admin key lives only in the process environment and is compared in constant time (`hmac.compare_digest`).

Source: THREAT_MODEL.md#authentication-one-shared-secret-or-many-scoped-ones, DATA_PROTECTION.md#cryptography

---

Can each administrator have an individual identity, rather than a shared account?

Answer: Yes. Named administrators each have a role and their own admin tokens. An administrator can hold several tokens and revoke each one separately, and deactivating an administrator stops all of their tokens at once. Once an active owner exists, the shared key becomes a break-glass key: each use is logged, audited (`break_glass_used`) and raises an alert, and owners can remove it.

Source: THREAT_MODEL.md#administration-named-administrators-and-roles, API.md, Administrators and admin roles

---

Does the product support single sign-on (SSO) for administrators?

Answer: Yes. Console sign-in supports one identity provider at a time, through OpenID Connect. SAML is not supported. The server runs the authorization code flow with PKCE (S256) as a confidential client, so the browser never holds a token, and each administrator's role comes from their groups at the provider. This feature has been stable since 0.30, after a security review.

Source: THREAT_MODEL.md#console-sign-in-through-sso-adr-0007, API.md, Console sign-in through SSO

Does the product support multi-factor authentication (MFA)?

Answer: Not directly. QueryAPIGate has no MFA of its own. When administrators sign in through SSO, the identity provider's MFA and policies apply. `QUERYAPIGATE_SSO_ONLY` removes personal tokens from people who sign in through SSO, so they can't bypass the provider's MFA. Admin tokens and API keys are bearer secrets with no second factor.

Deployer: enforce MFA at your identity provider, and set `QUERYAPIGATE_SSO_ONLY`.

Source: THREAT_MODEL.md#console-sign-in-through-sso-adr-0007, API.md, Console sign-in through SSO

How are administrator Console sessions protected?

Answer: The session cookie is `__Host-`, `HttpOnly`, `Secure` and `SameSite=Strict`, and is stored only as a hash. It authenticates a request only together with the session's CSRF token. A sign-in callback is used once, within ten minutes, and only in the browser that started it. Sessions last `QUERYAPIGATE_SSO_SESSION_HOURS` (8 by default, at most 168).

Source: THREAT_MODEL.md#console-sign-in-through-sso-adr-0007, API.md, Console sign-in through SSO

When a person is removed at the identity provider, how quickly does their access end?

Answer: A Console session that is already open lasts until it expires (`QUERYAPIGATE_SSO_SESSION_HOURS`). Switching the administrator off in QueryAPIGate ends their

sessions at once. JWTs and MCP OAuth tokens can't be revoked before they expire; disabling or narrowing the role in QueryAPIGate takes effect on the next request.

Deployer: keep access-token lifetimes short.

Source: THREAT_MODEL.md#console-sign-in-through-sso-adr-0007, THREAT_MODEL.md#oauth-for-ai-clients-adr-0006

How are tokens from an external identity provider validated (JWT/JWKS)?

Answer: The checks are:

- the signature;
- `exp`, which is required;
- `iss` and `aud`, both mandatory when a JWKS URL is used;
- the algorithm, which must be one the configuration fixes. HMAC and public-key families can't be mixed, and `none` is never accepted.

Settings that would make checking unsafe stop the server at startup, for example a JWKS URL over plain `http://` or a short secret. A parameter declared `from_claim` is bound from the verified token, never from the request.

Source: THREAT_MODEL.md#authentication-one-shared-secret-or-many-scoped-ones, API.md, Signed-in users (JWT)

How do AI agents (MCP clients) authenticate?

Answer: Agents send an API key in a request header, or use OAuth 2.1. With OAuth, the MCP server acts as a resource server and checks:

- `aud`, which must be the MCP server's own URL;
- `iss`;
- the signature, using the provider's own key algorithm (asymmetric only);
- `exp`;
- the configured scope.

A token issued for another application, including this server's own REST API, is refused. A caller's token is never passed on to databases.

Source: THREAT_MODEL.md#oauth-for-ai-clients-adr-0006, THREAT_MODEL.md#mcp-server

---

Do credentials expire?

Answer: Scoped API keys can carry an `expires_at` date, checked live on every request, and the `key_expiring` and `key_expired` alerts report them. Admin tokens expire after 90 days by default (`--expires YYYY-MM-DD|never`). Expiry is set to the day, not the hour or minute. JWT and OAuth tokens expire as their provider issues them.

Source: API.md, Key expiry, API.md, Administrators and admin roles, THREAT_MODEL.md#administration-named-administrators-and-roles

---

Can credentials be rotated without downtime?

Answer: Yes. Reissuing a scoped key's secret (`POST /api/v1/api-keys/{name}/rotate`) shows the new secret once. The old secret keeps working for a grace period the administrator chooses, at most a week, or 0 for a leaked secret. The administrator can see when the old secret was last used. The audit key and the encryption key for stored passwords can also be rotated: list the new key first and the old one after it.

Source: THREAT_MODEL.md#authentication-one-shared-secret-or-many-scoped-ones, THREAT_MODEL.md#secrets-at-rest, guide 50

---

3. Authorization and least privilege

What access-control model does the product use?

Answer: Administrators have one of four fixed roles, each holding a fixed set of capabilities:

- **owner:** everything;
- **admin:** everything except administrators;
- **developer:** saved queries and collections;
- **auditor:** read-only, with no SQL.

Data callers (scoped keys, signed-in users and agents) get grants instead. A test fails if a Management API operation has no capability assigned, so a new route can't ship unguarded.

Source: THREAT_MODEL.md#administration-named-administrators-and-roles

---

What can a scoped API key be restricted to?

Answer: A scoped key can be limited to:

- connections (ad-hoc SQL targets);
- collections and named saved queries;
- whether it may write, and which write verbs;
- `allowed_tables`;
- `allowed_ips`;
- a rate limit, concurrency, time and row limits;
- usage quotas;
- an expiry date;
- column masking.

`connections`, `collections` and `queries` add up to what the key can reach. The other fields only ever narrow that reach. A `queries` grant alone gives an external client curated access to named saved queries, with no ad-hoc SQL at all.

Source: THREAT_MODEL.md#authorization-what-a-scoped-key-can-actually-do

---

Is access read-only by default?

Answer: Yes. Writes need `QUERYAPIGATE_ALLOW_WRITES` on the server, and also `allow_writes` on the key. A key can never grant itself more than the server allows. The read-only guard rejects write statements, including writes hidden in a writable CTE, in `EXPLAIN ANALYZE` or in `SELECT ... INTO`. PostgreSQL, MySQL, ClickHouse and SQLite connections also run read-only queries in read-only database sessions.

Source: THREAT_MODEL.md#authorization-what-a-scoped-key-can-actually-do, THREAT_MODEL.md#query-execution-what-actually-stops-a-bad-statement

---

How is access to specific tables restricted?

Answer: With `allowed_tables`. It is enforced with a real SQL parser (`sqlglot`), not a regex. A key restricted to tables is refused outright on H2 and JDBC connections, rather than checked loosely. On MongoDB the list is the collections a key may `find()` in; this was fixed in 0.34.1 (see section 9).

Source: THREAT_MODEL.md#authorization-what-a-scoped-key-can-actually-do, CHANGELOG 0.34.1

Can access be restricted by source IP address?

Answer: Yes. A key's `allowed_ips` lists IPv4 or IPv6 addresses and CIDR ranges. The check fails closed if the client address can't be resolved. The shared admin key is never restricted this way. Behind a proxy, the MCP server sees the proxy's address for every caller, so `allowed_ips` can't tell MCP callers apart there.

Deployer: set `QUERYAPIGATE_TRUST_PROXY` to the real number of proxy hops. A wrong value lets a client forge its address.

Source: THREAT_MODEL.md#authorization-what-a-scoped-key-can-actually-do, THREAT_MODEL.md#rate-limiting-and-abuse-prevention, THREAT_MODEL.md#mcp-server

---

How are roles applied to keys? Do changes to a role apply retroactively?

Answer: It depends on the caller. For API keys, roles are templates: a key created from a role copies its grants once, and editing the role later doesn't change existing keys. Signed-in JWT users use their role live, so a change to the role applies on their next request.

Source: THREAT_MODEL.md#authorization-what-a-scoped-key-can-actually-do, API.md, Signed-in users (JWT)

Can an AI agent change data?

Answer: Only through saved queries an author has marked `agent_writes`. Each must be a single `INSERT`, `UPDATE` or `DELETE`, never DDL, and the caller's grant must allow the write, with `QUERYAPIGATE_ALLOW_WRITES` on. A named person approves each write before it runs, and the credential that asked can never approve its own request. Each write runs in a transaction that is rolled back if it changes more than `max_affected_rows`. Insert-only queries skip the approval. `execute_sql` over MCP never writes.

Source: THREAT_MODEL.md#agents-that-write-adr-0008, THREAT_MODEL.md#mcp-server

---

How do you protect against one caller exhausting shared resources?

Answer: Several limits apply:

- per-address and per-key rate limits;
- per-key or per-role `max_concurrent`, `query_timeout` and `max_rows`, which only ever narrow the server's own limits;
- a server-wide query time limit (`QUERYAPIGATE_QUERY_TIMEOUT`, 30 seconds by default) and `QUERYAPIGATE_MAX_PAGE_SIZE`.

Usage quotas meter calls and rows per day or month, but they aren't a hard security boundary.

Deployer: set `QUERYAPIGATE_RATE_LIMIT` (off by default) on anything reachable beyond a trusted network; it also throttles API key guessing.

Source: THREAT_MODEL.md#rate-limiting-and-abuse-prevention, SECURITY.md#hardening-checklist-for-deployments

---

Does the product enforce least privilege on the database side?

Answer: No. QueryAPIGate connects with the database account each connection is given. Its own read-only guard is defence in depth, not a substitute for database grants.

Deployer: connect with a database account that has only the privileges the API needs.

Source: SECURITY.md#hardening-checklist-for-deployments, THREAT_MODEL.md#non-goals-what-this-does-not-defend-against

---

4. Data protection and privacy

What customer data does the product store?

Answer: The store holds configuration (connections, saved queries and their versions, keys, roles and administrators), run history, usage counters, the event log, the audit log, and an agent's write requests. Query results are not stored, except by the response cache and exports when the Deployer turns them on. Parameter values are not stored either: run history records parameter names, never their values.

Source: DATA_PROTECTION.md#the-store, DATA_PROTECTION.md#what-passes-through-and-isnt-kept

---

What personal data does the product process?

Answer: The product processes:

- callers' names, which for a signed-in user is `jwt:<subject>`, often an email address;
- administrators' names and email addresses;
- IP addresses;
- an AI agent's write arguments;
- whatever the connected databases hold, which passes through.

With no SSO and no JWTs, no person is named, only API keys.

Source: DATA_PROTECTION.md#personal-data-it-handles, guide 57

Is personal data written to logs?

Answer: Requests are logged with their path, method, status, time, request id, client address and caller name; the query string is left out unless `QUERYAPIGATE_ACCESS_LOG_QUERY=1` is set. Some things can still reach the log: an ad-hoc statement's text, and a database's own error message, which can quote a value. Setting `QUERYAPIGATE_HISTORY_ADHOC_SQL=hash` reduces ad-hoc statements to their SHA-256.

Deployer: treat the log platform as holding personal data.

Source: DATA_PROTECTION.md#where-data-is-sent, THREAT_MODEL.md#what-is-kept-and-the-cryptography-it-rests-on-adr-0016

Can sensitive columns be hidden from specific consumers?

Answer: Yes. A `mask` grant on a key, role or export can hash, mask, null or drop named source columns. Result columns are traced to their sources through the SQL, not matched by name, and a query that can't be traced is refused before it runs. Masking hides values, not answers about them: a caller who writes SQL can still filter or group on a masked column. Callers with masking rules are refused on H2, JDBC and MongoDB connections.

Source: THREAT_MODEL.md#column-masking-which-values-leave, API.md, Column masking

---

How long is each kind of data retained?

Answer: Retention, by kind of data:

Data	Kept for
Run history	Bounded per query version, for ad-hoc runs, by days, or by sampling (`QUERYAPIGATE_HISTORY_*`)

Data	Kept for
Event log	24 hours by default
Usage counters	400 days
Agent write requests	30 days
Audit log in the store	The newest `QUERYAPIGATE_AUDIT_LOG_LIMIT` entries (500 by default)
SSO sign-ins in progress	10 minutes
Configuration	Until deleted

Deployer: the retention of logs, traces and SIEM copies is set by your own platforms.

Source: DATA_PROTECTION.md#the-store, DATA_PROTECTION.md#what-a-deployer-controls

---

Can an individual's data be erased on request (right to erasure)?

Answer: Yes. An owner can run `queryapigate history forget <caller>` or `DELETE /api/v1/history?caller=...`. This deletes the caller's runs, events and usage counts in the store and in Redis, and records the erasure in the audit log by the caller's SHA-256, not their name. A `--dry-run` shows the counts first. The command does not erase the audit log's entries about administrators, write requests (kept 30 days), or the Deployer's logs, traces and SIEM copies.

Source: DATA_PROTECTION.md#a-person-asks-for-their-data-to-be-erased

---

Who can read stored personal data?

Answer: Access is by administrator capability:

- run history and the audit log need `observe`;
- keys need `access.read`;
- agents' write arguments need `writes.approve`;
- erasure needs `history.forget`, which only owners have.

Callers see only what their grants allow, after masking.

Source: DATA_PROTECTION.md#the-store, guide 57

Can data be delivered to third parties, such as partners' storage buckets?

Answer: Only through exports. Only owners and admins can define an export (`exports.write`). Each export writes only under its destination's prefix, after the export's own masking, and every change to an export is audited.

Deployer: scope the destination credentials to the prefix.

Source: THREAT_MODEL.md#query-execution-what-actually-stops-a-bad-statement, DATA_PROTECTION.md#where-data-is-sent

5. Cryptography

Is data encrypted at rest?

Answer: Partly. Connection and export-destination passwords are Fernet-encrypted (AES-128-CBC with HMAC-SHA256) when `QUERYAPIGATE_SECRET_KEY` is set. This is opt-in, not the default. API keys, admin tokens and session ids are stored only as SHA-256 hashes. The rest of the store (configuration and run history) is not encrypted by QueryAPIGate.

Deployer: put the store on an encrypted volume or database.

Source: THREAT_MODEL.md#secrets-at-rest, DATA_PROTECTION.md#cryptography, guide 57

Can database credentials be kept out of the product's store entirely?

Answer: Yes. A password can be a reference: `${VAR}`, `${file:...}` or `${vault:...}`, or short-lived Vault database credentials. References to AWS Secrets Manager and RDS IAM are experimental. Resolved values are held only in memory and never written to the store, a log, a response, a trace or the audit log. Nothing is reachable through a reference until the Deployer allow-lists it.

Source: THREAT_MODEL.md#secrets-from-secrets-managers-adr-0015, DEPLOYMENT.md#3-secrets-env

---

How are encryption keys managed?

Answer: `QUERYAPIGATE_SECRET_KEY` and `QUERYAPIGATE_AUDIT_KEY` are supplied by the Deployer and must be kept outside the store and out of version control. Without the encryption key, encrypted passwords can't be recovered. Both keys can be rotated: `queryapigate rekey` re-encrypts every stored password in one all-or-nothing transaction.

Deployer: keep the keys in your secrets manager.

Source: THREAT_MODEL.md#secrets-at-rest, API.md, A tamper-evident chain

---

Is data encrypted in transit?

Answer: Inbound traffic is not encrypted by QueryAPIGate itself: it does not terminate TLS, which is the reverse proxy's job. Outbound connections to Vault, the SIEM, the OTLP collector and Redis use TLS through Python's `ssl`. PostgreSQL and MySQL connections' TLS settings reach their drivers, and `aws-rds-iam` is refused without TLS.

Deployer: terminate TLS in front of the server. It is required once a key exists.

Source: THREAT_MODEL.md#transport, DATA_PROTECTION.md#cryptography, THREAT_MODEL.md#secrets-from-secrets-managers-adr-0015

Is the cryptography FIPS 140 validated?

Answer: QueryAPIGate uses only FIPS-approved algorithms (FIPS 140-3, SP 800-131A), and a test fails if the code gains one that isn't approved. Validation is a property of the cryptographic module, not the application: QueryAPIGate runs on a FIPS 140-validated module when the platform provides one. Two examples are RHEL 9 in FIPS mode and Ubuntu Pro FIPS, with Python and `cryptography` from the distribution. QueryAPIGate itself is not a validated module. A CI job runs the whole test suite in Red Hat UBI 9 with OpenSSL's FIPS provider forced, after first checking that FIPS is really in force.

Source: DATA_PROTECTION.md#cryptography, DATA_PROTECTION.md#on-a-fips-validated-module, guide 57

What falls outside the FIPS boundary?

Answer: The following bring their own cryptography:

- `psycopg2-binary`, which ships its own libpq and OpenSSL; build `psycopg2` from source on a FIPS platform;
- `mysql-connector-python`'s C extension;
- DuckDB's TLS for remote files;
- the Java runtime used by JDBC and H2 connections;
- the `cryptography` wheel from PyPI.

Deployer: decide which of these to use on a FIPS platform.

Source: DATA_PROTECTION.md#on-a-fips-validated-module

Which algorithms are used, and for what?

Answer:

Algorithm	Used for
Fernet	Stored passwords
SHA-256	Keys, tokens and sessions stored as hashes; PKCE
HMAC-SHA256	The audit chain; the masking `hash` action
RS256 / HS256, or configured RS*, PS*, ES*, HS*	Signed-in users' tokens
AWS SigV4	RDS IAM tokens
The OS random source (`secrets`)	Every secret the server generates

`random` is used only where nothing depends on it being unpredictable.

Source: DATA_PROTECTION.md#cryptography

---

6. Logging, audit and monitoring

Are administrative actions logged?

Answer: Yes. Every configuration change is appended to the audit log, recording:

- the actor;
- how they authenticated (`via`: their own token, break-glass key, SSO, CLI, configuration file);
- the action and target;
- a diff of what changed.

Passwords appear only as `********` or `"changed"`, and secrets and hashes never appear. Every SSO sign-in, refusal and sign-out is recorded.

Source: THREAT_MODEL.md#audit-logging-and-data-at-rest, API.md, Audit log

---

Is the audit log tamper-evident?

Answer: Yes. It is a hash chain, HMAC-signed when `QUERYAPIGATE_AUDIT_KEY` is set. Changing, removing, inserting or reordering an entry breaks the chain, which `queryapigate audit verify` detects; a check runs every 5 minutes and raises the critical `audit_chain_broken` alert. The chain alone can't show that its newest entries were deleted, or that someone holding both the store and the key rewrote it. A copy kept outside the store is what proves those.

Source: THREAT_MODEL.md#the-audit-trail

Can audit logs be sent to a SIEM?

Answer: Yes. The audit log can be sent as syslog over TLS (RFC 5425), with optional mutual TLS, or as OTLP logs. Each entry carries its `seq` and `hash`, and `audit verify --file` checks the SIEM's copy. Delivery is at least once and survives the SIEM being down. Syslog over TCP can lose a batch if the receiver dies without closing the connection; OTLP acknowledges each batch. The stream is tested on every push against rsyslog, syslog-ng and the OpenTelemetry Collector.

Source: THREAT_MODEL.md#the-audit-trail, API.md, Streaming the audit log to a SIEM

How long are audit records retained?

Answer: The store keeps the newest `QUERYAPIGATE_AUDIT_LOG_LIMIT` entries, 500 by default, and keeps more while a SIEM output is behind. `QUERYAPIGATE_AUDIT_LOG_EXPORT_FILE` appends every entry to a file that is never capped or rewritten.

Deployer: set retention in the SIEM or export file to match your obligations.

Source: API.md, Audit log, DATA_PROTECTION.md#what-a-deployer-controls

---

Is data access (query execution) logged?

Answer: Yes. Run history records each run: when it ran, which query and version, the caller, the transport, the row count, the time taken and the status. It records parameter names but never their values. Run history is a plain table and is **not** tamper-evident, unlike the audit log.

Source: DATA_PROTECTION.md#the-store, THREAT_MODEL.md#audit-logging-and-data-at-rest

---

What monitoring and alerting is available?

Answer: The server exposes Prometheus `/metrics`, and a Grafana dashboard is bundled. A built-in alerts list (`GET /api/v1/alerts`) includes, for example:

- `open_server`;
- `break_glass_used`;
- `audit_chain_broken`;
- `audit_delivery_behind`;
- `key_expiring`;
- `secret_provider_failing`;
- `agent_unusual_volume`;
- `key_unused`.

The alerts list is meant for day-to-day operators, not as a replacement for on-call alerting.

Deployer: alert on `/metrics` from Prometheus.

Source: API.md, Alerts, DEPLOYMENT.md#6-observability

Is the metrics endpoint protected?

Answer: Only when the Deployer sets `QUERYAPIGATE_METRICS_TOKEN`; until then it is open. Its labels name API keys, queries and connections. Administrators and signed-in users appear only under the labels `admin` and `jwt`. The Helm chart sets the token by default.

Deployer: set the token wherever the server can be reached from outside.

Source: THREAT_MODEL.md#non-goals-what-this-does-not-defend-against, DEPLOYMENT.md#6-observability

---

Is distributed tracing supported, and what do traces contain?

Answer: Yes. OpenTelemetry tracing is **experimental**, and it is off unless an OTLP endpoint is set. Spans carry the route, the saved query's name, the row count, the caller's name and the caller's address. They never carry parameter values, rows, statements, headers or credentials, and a test enforces this. By default a caller's `traceparent` can raise how much is sampled; `OTEL_TRACES_SAMPLER=traceidratio` prevents that.

Source: THREAT_MODEL.md#transport, ADR 0010

---

7. Availability, backup and recovery

Does the product support high availability?

Answer: Yes, with several instances on a PostgreSQL store with Redis. Everything is then shared, so instances are interchangeable behind a load balancer. This is tested in CI with three real servers. With a SQLite store, only one instance is supported; two replicas with their own SQLite volumes are two separate servers. Without Redis, rate limits and the cache are per process, and the `instances_not_shared` alert reports it. If Redis becomes unreachable, rate limits stay enforced per process, never lifted, and the `rate_limits_not_shared` alert says so.

Source: DEPLOYMENT.md#scaling-out, THREAT_MODEL.md#non-goals-what-this-does-not-defend-against

---

How is the product's data backed up and restored?

Answer: A SQLite store is backed up with `queryapigate backup`, which uses SQLite's backup API to take a consistent copy while the server runs. A PostgreSQL store is backed up with `pg_dump`, or the database's own backups. CI backs up a store, loses it, restores it, and checks the result on every run.

Deployer: schedule backups, ship them off the host, and test restores.

Source: DEPLOYMENT.md#5-persistent-data-backups-and-restores

What does a backup not contain?

Answer: A backup doesn't contain:

- `QUERYAPIGATE_SECRET_KEY`, without which encrypted passwords are unreadable;
- the admin key;
- environment variables that connections reference;
- the connected databases;
- the Redis cache, which refills on its own.

Scoped keys are in the store as hashes and keep working after a restore.

Deployer: back up those secrets separately, and back up your databases in their own right. Encrypt backups and restrict access to them: the store holds connection details, saved SQL, key hashes, run history and the audit log.

Source: DEPLOYMENT.md#what-a-backup-of-the-store-does-not-contain

---

How are upgrades performed, and can they be rolled back?

Answer: The store's schema migrates automatically on first start. A store the version can't safely run on stops startup before anything changes. Several instances can be upgraded one at a time without downtime, one minor version at a time. Going back to an older release in place is not supported: restore the backup taken before the upgrade.

Source: DEPLOYMENT.md#upgrading, DEPLOYMENT.md#rolling-upgrades-several-instances-on-one-store

---

8. Secure development and supply chain

Is security testing part of the development process?

Answer: Yes. The test suite runs in CI. It includes:

- a property-based fuzz suite (Hypothesis) for the SQL guard and parameter binder, which always runs;
- tests that fail if a Management API route has no capability, if a span carries a value, or if the code gains a non-approved algorithm;
- a full run under OpenSSL's FIPS provider.

Source: SECURITY.md#hardening-checklist-for-deployments, THREAT_MODEL.md#administration-named-administrators-and-roles, DATA_PROTECTION.md#cryptography, INSTALLATION_AND_SETUP.md

---

Are features security-reviewed before release?

Answer: Features that guard access or data come under the stability promise only after a security review. The threat model records each review's scope and findings: for example, reviews for 0.30, 0.32 and 0.34, which found and fixed issues before release.

Source: CHANGELOG, Versioning and compatibility, THREAT_MODEL.md#console-sign-in-through-sso-adr-0007, THREAT_MODEL.md#live-events-adr-0020

Are releases signed?

Answer: Yes, from 0.31. Every image (including the `-h2` variant) and the Helm chart is signed keyless with cosign, by digest, by the repository's release workflow, and the signature is recorded in Sigstore's public log. A valid signature names the identity `.../.github/workflows/docker.yml@refs/tags/v<X.Y.Z>`. Releases before 0.31 are not signed. Guide 59 shows Sigstore's policy-controller and Kyverno admitting a signed image and refusing an unsigned one.

Deployer: run only signed releases, by digest, and enforce this with an admission policy.

Source: SECURITY.md#verify-a-release, ADR 0018, guide 59

Do you provide an SBOM and build provenance?

Answer: Yes. Each image carries an SPDX SBOM, SLSA provenance naming the commit, and a GitHub artifact attestation. The chart has a GitHub attestation. PyPI packages carry PEP 740 attestations.

Source: ADR 0018, guide 59

Are build inputs pinned?

Answer: Yes. Base images are pinned by digest and every GitHub Action by commit SHA, and a test fails on an unpinned `uses:`. Dependabot proposes updates as reviewed commits. Builds are not reproducible byte for byte.

Source: ADR 0018, THREAT_MODEL.md#on-kubernetes-the-helm-chart-adr-0013

What does a release signature not guarantee?

Answer: It guarantees only that this repository's workflow built the artifact on that tag. It doesn't guarantee that the code is correct. Someone who can push a tag can still sign a release, and the public log then records it. It also doesn't vouch for the upstream base images' own supply chain.

Source: THREAT_MODEL.md#on-kubernetes-the-helm-chart-adr-0013, ADR 0018

Is configuration managed through change control?

Answer: It can be. With configuration as code, connections, grants and keys can live in one file in git. Applying a file is an owner's capability, all or nothing, and audited; a file can't hold a literal password. Anyone who can edit the file has an owner's reach at the next start.

Deployer: protect the branch and require review.

Source: THREAT_MODEL.md#configuration-as-code-adr-0014

---

9. Vulnerability management and incident response

How are security vulnerabilities reported?

Answer: Privately, not in a public issue: through GitHub's "Report a vulnerability" form on the repository, or by email to arcswdev@gmail.com.

Source: SECURITY.md#reporting-a-vulnerability

What are your response times for reported vulnerabilities?

Answer: SECURITY.md states only that a reporter can expect an acknowledgement within a few days. The project commits to no other response or remediation time.

Source: SECURITY.md#reporting-a-vulnerability

Which versions receive security fixes?

Answer: Security fixes are made against the latest released version. Older versions are not patched.

Deployer: keep up with releases. A patch release never breaks a covered surface.

Source: SECURITY.md#supported-versions, CHANGELOG, Versioning and compatibility

---

How are security fixes communicated to customers?

Answer: In the **Security** section of the changelog entry for each release that contains a fix. Recent entries give the affected version range, what could happen, and how to check whether it happened in your deployment. A security fix that can't be made compatibly may remove a feature sooner than the deprecation process allows, and says why.

Deployer: watch the changelog's Security entries.

Source: CHANGELOG, CHANGELOG, Versioning and compatibility

Can you give recent examples of security fixes?

Answer: Yes:

- **0.34.1:** fixed `allowed_tables` not being checked on MongoDB connections (affected 0.11.0 to 0.34.0).
- **0.34.2:** fixed two MongoDB issues (affected 0.9.0 to 0.34.1). A parameter value could carry query operators. Server-side JavaScript was not refused in a projection or sort.

Each entry explains how to check run history for signs of use.

Source: CHANGELOG 0.34.1, CHANGELOG 0.34.2

What tools help a customer investigate a suspected incident?

Answer: These tools are available:

- the audit log and `queryapigate audit verify`, including `--file` for the SIEM's copy;
- run history (`GET /api/v1/history`), filterable by key;
- request ids in logs;
- the `break_glass_used` alert;
- per-key `last_used_at`.

Key secrets can be revoked at once (`active: false`), or rotated with no grace period.

Deployer: incident response for your deployment is yours.

Source: THREAT_MODEL.md#the-audit-trail, API.md, Authentication and permissions, THREAT_MODEL.md#authentication-one-shared-secret-or-many-scoped-ones

---

10. Network and hardening

Does the product set HTTP security headers?

Answer: Yes. Every response sets `X-Content-Type-Options: nosniff` and `Cache-Control: no-store`; a cached saved query is `private`. API responses also carry a CSP that allows nothing and

`Referrer-Policy: no-referrer`. The server version banner is removed. HSTS is opt-in (`QUERYAPIGATE_HSTS_MAX_AGE`) and sent only over HTTPS.

Source: THREAT_MODEL.md#transport, ADR 0010

How is TLS provided?

Answer: By a reverse proxy the Deployer runs (Caddy, nginx or Traefik). QueryAPIGate does not terminate TLS itself.

Deployer: terminate TLS in front of the server, and set `QUERYAPIGATE_TRUST_PROXY` to the number of proxy hops. Leave it at `0` when clients connect directly.

Source: THREAT_MODEL.md#transport, DEPLOYMENT.md#2-compose-file, SECURITY.md#hardening-checklist-for-deployments

---

Is cross-origin (CORS) access restricted?

Answer: Yes. CORS is off unless `QUERYAPIGATE_CORS_ORIGINS` is set, and cookies are not used for API credentials. CORS controls browsers only; it is not authentication.

Deployer: list only sites you control, and never use `*` without an API key.

Source: THREAT_MODEL.md#cors, API.md, Rate limiting and CORS

Does the container run as a non-root user?

Answer: Yes. The image runs as a non-root user (UID 1000) and ships a `HEALTHCHECK`.

Source: DEPLOYMENT.md#1-the-image

What pod security does the Helm chart apply?

Answer: The chart's pods run as UID 1000, with a read-only root filesystem, every capability dropped, the RuntimeDefault seccomp profile and no service account token. The server writes only to `/data` and `/tmp`. The chart refuses secret settings in plain `config`.

Source: THREAT_MODEL.md#on-kubernetes-the-helm-chart-adr-0013, DEPLOYMENT.md#kubernetes-helm

---

Is network segmentation enforced?

Answer: The Helm chart's NetworkPolicy is **off by default**. Without it, any pod in the cluster can reach the API and the MCP server. When enabled, it limits inbound connections and restricts outbound traffic to listed destinations.

Deployer: enable it, and list your databases, Redis and SIEM. Network perimeter controls (proxy, VPN, WAF) are yours.

Source: THREAT_MODEL.md#on-kubernetes-the-helm-chart-adr-0013, THREAT_MODEL.md#non-goals-what-this-does-not-defend-against

---

How are the product's own secrets protected in Kubernetes?

Answer: Secrets come from a Kubernetes Secret: `existingSecret`, or one the chart creates. Anyone who can `get secrets` in the namespace, or read etcd unencrypted, can read them.

Deployer: keep RBAC narrow, encrypt Secrets at rest, and prefer `existingSecret`, which keeps values out of values files and Helm's release record.

Source: THREAT_MODEL.md#on-kubernetes-the-helm-chart-adr-0013

Is there protection against SQL injection?

Answer: Yes. Bound `:name` parameters are real parameterization: values are passed to the driver separately and never become part of the SQL text. Older `{name}` text placeholders are string substitution, restricted to a narrow set of safe characters.

Deployer: prefer `:name` parameters.

Source: THREAT_MODEL.md#query-execution-what-actually-stops-a-bad-statement

---

Can an administrator use the product to reach files or the product's own secrets?

Answer: Several guards prevent it. A connection can't point at the product's own store. DuckDB reads only its `allowed_paths`, which are empty by default. Local export folders are off unless the Deployer lists them. A secret reference can't name the server's own settings or cloud credentials. `${file:...}` can't read the server's secret files, service account tokens or `/proc`.

Deployer: set `QUERYAPIGATE_SECRET_ENV_VARS`.

Source: THREAT_MODEL.md#query-execution-what-actually-stops-a-bad-statement, THREAT_MODEL.md#secrets-from-secrets-managers-adr-0015

---

11. Known limits and non-goals

What does the product explicitly not defend against?

Answer: QueryAPIGate does not defend against:

- network attacks: it is not a network perimeter or a web application firewall;
- a malicious administrator: an owner, admin or developer can read every row on every connection;
- a missed statement: the read-only guard doesn't replace least-privilege database credentials;
- planted instructions: an AI agent may be steered by what it reads.

The threat model lists the rest. Run history isn't tamper-evident, connection metadata is visible to every administrator, query names can be probed, and `/metrics` is open until a token is set.

Source: THREAT_MODEL.md#non-goals-what-this-does-not-defend-against

---

What are the limits of the read-only SQL guard?

Answer: It is a regex-based classifier, not a SQL parser. It is dialect-aware and fuzz-tested, but it can never be a full parser for every server mode. On DuckDB, H2 and JDBC connections, which are opened read-write, the guard is the only lock against writes. The project assumes it could have another gap like the one already found and fixed.

Source: THREAT_MODEL.md#query-execution-what-actually-stops-a-bad-statement, SECURITY.md#hardening-checklist-for-deployments

---

Are query time limits enforced on every connection type?

Answer: No. On generic JDBC connections, `QUERYAPIGATE_QUERY_TIMEOUT` is not enforced, because JDBC has no portable way to cancel a statement; a test query ran past the limit to the end.

Deployer: set a statement timeout in the database, or in the JDBC URL if your driver supports one. Setting the timeout to `0` removes the limit everywhere.

Source: guide 3, SECURITY.md#hardening-checklist-for-deployments

---

Which features are experimental?

Answer: H2, JDBC and MongoDB connections; tracing with OpenTelemetry; and secrets and credentials from AWS (`${aws-sm:...}`, `aws-rds-iam`). These sit outside the compatibility promise and may change in any minor release. Each is marked in its documentation, in `/openapi.json` and in Settings, and logs a warning at startup.

Source: CHANGELOG, Versioning and compatibility

Are usage quotas a security boundary?

Answer: No. Quotas meter a contract; they don't enforce a security boundary. A caller can overshoot by the calls it has in flight, and with several instances but no Redis, by up to one quota per instance in a short burst. `max_concurrent` and the rate limit are what bound a burst.

Source: THREAT_MODEL.md#rate-limiting-and-abuse-prevention

---

12. Certifications and audits

Do you hold a SOC 2 report or ISO 27001 certification?

Answer: No. SOC 2 and ISO 27001 assess an organisation running a service. QueryAPIGate is software that runs inside the Deployer's environment, under the Deployer's controls, so there is no hosted service to certify. What it provides is the evidence your own assessment needs: this pack, the threat model, the data inventory and checks you can run on your own server.

Deployer: your own SOC 2 or ISO 27001 scope covers how you run QueryAPIGate: hosts, network, secrets, backups, access reviews and log retention.

Source: guide 57

Has the product undergone an independent third-party penetration test or audit?

Answer: The project publishes no third-party penetration test or audit report. Its own security reviews are recorded in the threat model, feature by feature, with what each looked at and what it found.

Source: THREAT_MODEL.md#console-sign-in-through-sso-adr-0007 (an example review record), ADR 0017

Threat model and security architecture

SECURITY.md tells you what to *do* - a checklist to run through before a deployment goes live. This page explains *why*: what QueryAPIGate is actually defending against, how each defense works under the hood, and - just as importantly - where the defenses stop and become the operator's job instead. Read it before deciding QueryAPIGate is (or isn't) an appropriate way to expose a database to something you don't fully trust.

What's actually at risk

QueryAPIGate sits between a caller and one or more real databases. Four things are worth protecting, in roughly descending order of damage if lost:

1. **The ability to execute arbitrary SQL/Mongo operations** against a connected database - the whole point of the service, and therefore its biggest liability if it reaches the wrong caller.
2. **Database credentials** - held in the store (`queryapigate.db`, or a PostgreSQL schema), needed to reach the databases at all.
3. **Query results and SQL text** - a saved query's SQL, and the rows it returns, can themselves be sensitive (customer data, schema details, business logic) independent of write access.
4. **API key secrets and the audit trail** - if either is compromised, every other control built on top of them (scoping, accountability) stops meaning anything.

Who might attack it, and from where

- **An unauthenticated network attacker** - anyone who can reach the HTTP port at all. The only thing standing between them and everything above is whether `QUERYAPIGATE_API_KEY` is set (see Open-access mode below) and whether TLS is actually terminated somewhere in front of the service.
- **A caller holding a scoped (non-admin) API key**, acting maliciously or - more realistically - a legitimate integration whose key has leaked. The threat here isn't "can they authenticate," it's "how far can that one key's specific grants actually reach."
- **An administrator** - the holder of the shared `QUERYAPIGATE_API_KEY` or of a named administrator's admin token - or anyone who steals one. What they can *change* depends on the role (below); but every role except auditor can run any SQL on every connection, so for the data each is equivalent to a database superuser scoped to whatever connections exist. This tier is not defended against by QueryAPIGate itself - see Non-goals.
- **Someone with filesystem or volume access** to `QUERYAPIGATE_HOME` - a compromised host, a misconfigured backup, a leaked volume snapshot. `queryapigate.db` is a single SQLite file holding connections, saved-query SQL, API key *hashes*, roles, execution history and the audit log - see Data at rest.

Security architecture, by control

Authentication: one shared secret, or many scoped ones

Every request carries (or omits) an `X-API-Key` header, checked once per request in `app.py`'s `before_request` hook, before anything else runs. Two kinds of key exist:

- **The admin key** (`QUERYAPIGATE_API_KEY`) is a single value living in the server's environment, compared with `hmac.compare_digest` (constant-time, so response timing can't be used to guess it character by character). It is never stored anywhere on disk - only in the process environment - and grants everything: every connection, every saved query, admin-only routes (managing connections, keys, roles).
- **Admin tokens** (`qagadm_...`) belong to named administrators (`admins.py`, ADR 0003): generated by the server, shown once, stored only as a SHA-256 hash and looked up by it, like scoped keys - but in their own table, and they authorize management by role rather than data access by grant. Each can carry an expiry; deactivating an administrator stops all of theirs at once.
- **Scoped keys** (`POST /api/v1/api-keys`, owners and admins to create) are generated by the server and shown exactly once - their secret is never stored, only a SHA-256 hash, in `queryapigate.db`'s `api_keys` table. A lost scoped key cannot be recovered, only revoked and replaced, the same property a password reset flow gives you and a plaintext-stored secret never could. A key's secret can be **reissued** (ADR 0009, `POST /api/v1/api-keys/{name}/rotate`, `access.write`): the new one is shown once, and the old one's hash keeps authenticating for a grace period the administrator chooses - at most a week, 0 for a leaked secret - under the same `active`, expiry, `allowed_ips` and rate limit. Only one old secret at a time: a second rotation during the grace is refused (`key_rotation_pending`), and `DELETE .../previous` ends the grace early. When the old secret was last used is shown, so it isn't ended while still in use - or left running after it should have been.
- **Signed-in users** (`Authorization: Bearer`, optional - `jwtauth.py`) carry a JWT from the deployer's own login or identity provider instead of a key. The signature is verified against a shared secret or the provider's JWKS; `exp` is required; `iss`/`aud` are enforced (mandatory with a JWKS URL, so a token minted for another application by the same provider is refused); the accepted algorithms are fixed by configuration and can't mix HMAC with public-key families (algorithm confusion), and `none` is never accepted. Grants come from a role, read live. A saved-query parameter declared `from_claim` is bound from the verified token, never from the request, which closes the "change `user_id` in the URL" class of bug (insecure direct object reference) for queries written that way. Residual risk: a JWT can't be revoked before it expires.

Open-access mode: a deliberate default worth understanding

If `QUERYAPIGATE_API_KEY` is never set, no scoped keys or administrators exist and JWT is off, QueryAPIGate does not refuse requests - every caller authenticates as an implicit, unrestricted admin. This is intentional (the code calls it "open by design"), aimed at a local trial or a fully trusted internal network where standing up a key first would just be friction. It is **not** a safe default for

anything reachable beyond a machine you control. SECURITY.md's first checklist item exists because of exactly this: the moment you deploy this somewhere another party can reach, setting `QUERYAPIGATE_API_KEY` stops being optional.

Authorization: what a scoped key can actually do

Three grant fields say what a key reaches, and they **add up** - a saved query is reachable if `connections`, `collections` or `queries` reaches it. The rest **narrow** what that reach may do, and none of them widens what another restricts:

Grant	Narrows
`connections`	Which database connections ad-hoc SQL can target (`"*"` = all).
`collections`	Which saved queries are reachable, by current collection membership - re-evaluated live, no wildcard.
`queries`	Specific saved-query names reachable regardless of connection grants - the only way to give an external client curated access with *no* ad-hoc SQL capability at all.
`allow_writes`	Whether write statements are permitted at all, on top of the server-wide `QUERYAPIGATE_ALLOW_WRITES` ceiling (a key can never grant itself more than the server allows).
`allowed_write_ops`	Which specific write verbs (insert/update/delete/...) if writes are allowed.
`allowed_tables`	Which tables a statement may touch - enforced via a real SQL parser (`sqlglot`), not a regex; unsupported for H2/JDBC connections, which are refused outright for a table-restricted key rather than silently under-checked.
`rate_limit`	A per-key request budget, additive with the server-wide IP-based limit, never a way around it.
`expires_at`	A day after which the key stops authenticating at all, checked live on every request.
`allowed_ips`	Which source IPs/CIDRs the key may be used from at all - fails closed if the client IP can't be resolved.

Roles (`POST /api/v1/roles`) are templates, not live policy: creating a key `from` a role copies that role's grants onto the key once, at creation time. Editing or deleting a role afterward has no retroactive effect on keys already created from it - a role is a way to stamp out several similarly-scoped keys quickly, not a group membership the system re-checks later.

Administration: named administrators and roles

The Management API (`/api/v1`) is for administrators only. Every operation needs one capability, and each of four fixed roles holds a fixed set (`adminroles.py`), checked before any route runs; a test fails if an operation has no entry, so a new route can't ship unguarded. A scoped key or signed-in app user gets `403 admin_only`; an administrator whose role lacks the capability, `403 role_forbidden`.

Role	Can change	Data plane
owner	everything, including administrators	any SQL, every connection
admin	connections, API keys and roles, queries, cache - not administrators	any SQL, every connection
developer	saved queries and collections	any SQL, every connection
auditor	nothing - reads configuration, the audit log and history	none

What this buys, and what it doesn't:

- **A stolen developer token can't mint keys, change connections or create administrators** - but it can read every row on every connection. Roles bound *management* blast radius; for data, the database account behind each connection is still the backstop. Give someone a scoped API key instead when they need less data reach.
- **Who can call what is an owner's or admin's decision.** Moving a query into a collection, or merging collections, in a way that gives an API key or role new reach needs `access.write`, even for a role that may otherwise organise queries. A developer still decides what saved queries *return* - a new version of a query a partner calls, a new query in a partner's collection - which the audit log and drafts make reviewable, not preventable.
- **The shared key stays an owner** (break-glass). Once an active owner exists, each use is logged, audited (`break_glass_used`) and alerted, so a leaked shared key in use shows up; owners can then remove it.
- **Lock-out and reopen-by-accident are refused**: while any administrator exists, authentication stays required even with no shared key, and the last active owner can't be demoted, deactivated or removed without the shared key set. `queryapigate admins token NAME`, run on the server, always recovers - anyone with shell access to the store is already beyond this model.
- **Self-service is narrow**: an administrator may issue and revoke only their own tokens; anyone else's need an owner. An administrator and an API key can't share a name, so actors in logs and history are unambiguous.
- **Residual risks**: tokens are bearer secrets with day-granularity expiry; there is no MFA or SSO yet (BACKLOG #84 Phase 2), so a token is only as safe as where it's kept.

Console sign-in through SSO (ADR 0007)

With `QUERYAPIGATE_SSO_*` set, administrators sign in through the organisation's identity provider; the server runs the authorization code flow with PKCE as a confidential client, so the browser never holds a token.

- **The callback can't be forged or replayed.** A sign-in is stored server-side with its state, nonce and PKCE verifier, used once (a racing duplicate is refused), within ten minutes, and only in the browser that started it - so someone can't send you a link that signs you in as them. The ID token's signature (the key's own algorithm), issuer, audience (`azp` too, with several audiences), nonce and expiry are all checked. After sign-in the browser only ever goes to a Console page - never an address from the request.
- **The session can't be used from elsewhere.** Its cookie is `__Host-`, `HttpOnly`, `Secure`, `SameSite=Strict`, stored only as a hash; and it authenticates a request only together with the

session's CSRF token, which only a page on this origin can read. A link, form or image on another site can't act as an administrator, whatever the method. `/auth/*` never answers cross-origin. Signing in again ends the session the browser had.
- **People are bound by the provider's** `sub`, never by an email. Linking an administrator who had tokens matches a *verified* email, once, and their role becomes their groups' - a pending owner account gives no one owner rights. Started sign-ins are capped, so starting them can't grow the store without bound.
- **What remains with the provider:** who is in the groups mapped to roles - **whoever controls the owners' group controls QueryAPIGate** - and whether `email_verified` can be trusted for linking. Someone removed there keeps a session already open until it expires (`QUERYAPIGATE_SSO_SESSION_HOURS`, default 8); switching them off in QueryAPIGate ends it at once. `QUERYAPIGATE_SSO_ONLY` takes personal tokens from people who sign in through SSO, so they can't route around the provider's MFA; the break-glass key stays, audited and alerted, for when the provider is down.

Reviewed for 0.30 (ADR 0017). Looked at: PKCE with S256, the state bound to the browser and taken once, the nonce, `azp`, the ID token's algorithm from the provider's key and never from the token; `return_to` only ever a Console page; linking only to an administrator an owner invited for that verified email; the role from the provider's groups every sign-in; the session cookie (`__Host-`, `SameSite=Strict`) and its CSRF token, reads included; signing in again ending the session the browser had. **Found:** a refused sign-in sent the Console its reason in the address - `?sso_error=...&sso_message=...` - and the Console showed `sso_message` as it was written, so anyone could send an administrator a Console link that said what they liked in the Console's own error banner ("call this number"). Now only the code travels, and the Console says what each one means in its own words; the reason, with whatever the provider said, is in the audit log.

Column masking: which values leave

A `mask` grant on a key, role or export (ADR 0005) hashes, masks, nulls or drops the values of named source columns in everything that caller receives. The design choice that matters: **result columns are traced to their sources through the SQL** (sqlglot lineage, by position, with the connection's schema for `SELECT *`), not matched by name - so `SELECT email AS e`, `lower(email)`, a CTE or a `UNION` is still masked - and **anything that can't be traced is refused before it runs**, never served unmasked. Probing during the build found, and the tests now hold against, the ways tracing could be led astray: DuckDB's `COLUMNS('regex')` (columns picked by pattern), a table alias that renames columns (`customer AS c(a, e)`), `SELECT *` over a `NATURAL`/`USING` join (fewer columns than parsed, which would shift every mask), whole-row references (`row_to_json(c)`), and a `VALUES` row holding a subquery. The database's error text, which can quote a value, is never returned to a caller with rules. The cache is keyed per set of rules, and run history records what was masked.

What remains:

- **Inference.** A caller that writes SQL can filter and group on a masked column (`WHERE email LIKE 'a%'`, `GROUP BY email`) - masking hides values in results, not answers about them. For callers near sensitive data, especially AI agents, grant saved queries only, or `allowed_tables` on a view that leaves the column out.
- **Hashes show equality.** `hash` is keyed (HMAC; the key never leaves the server), so a guessed value can't be hashed and compared, but equal values give equal codes - by

design, for grouping and joining.
- **Tracing is only as good as the parser.** A construct sqlglot reads wrongly, rather than not at all, could trace a column to the wrong source. The refusal rules are deliberately broad, and the feature is experimental for that reason; database-level column privileges remain the backstop for a column that must never leave.

Query execution: what actually stops a bad statement

This is the part worth being the most precise about, because it's also the part with the sharpest edges.

- **Bound `:name` parameters are real parameterization.** `sqltools.bind_parameters()` rewrites `:name` into the target driver's native paramstyle and hands values to the driver separately from the SQL text - the same mechanism that makes parameterized queries safe in any language. Values never become part of the SQL string, so this path has no SQL-injection surface by construction.
- **The read-only / single-statement guard is a regex-based classifier, not a SQL parser.** It strips string literals and comments (dialect-aware - MySQL/ClickHouse honor backslash escapes inside string literals by default, PostgreSQL/SQLite/H2 do not, and the guard accounts for the difference), checks for a bare `;` outside what's left to block multi-statement smuggling, and classifies the leading keyword against an allow-list when writes aren't permitted - then looks behind a read-only one for writes it can hide: a writable CTE (`WITH d AS (DELETE ... RETURNING *) SELECT ...`), the statement `EXPLAIN ANALYZE` runs, and `SELECT ... INTO`. The same rule covers a key's `allowed_write_ops`. PostgreSQL, MySQL, ClickHouse and SQLite connections also run read-only queries in read-only sessions, so the database refuses a write even if the guard missed it; **DuckDB, H2 and JDBC connections are opened read-write, so there the guard is the only lock** - the case where 0.16 closed a real gap (`EXPLAIN ANALYZE DELETE` ran on DuckDB with writes off). **This is explicitly documented as defence in depth, not a substitute for least-privileged database grants** - a regex can approximate a SQL parser's boundaries but can never fully replace one, and the project has already found and fixed a real gap of exactly this kind (see below).
- `allowed_tables` **uses a real parser** (`sqlglot`) specifically because table extraction has to be correct, not just approximately right - a key restricted to certain tables is refused entirely rather than let through on a best-effort guess when the dialect can't be parsed reliably (H2/JDBC).
- **DuckDB file access is enforced by DuckDB itself.** A DuckDB connection reads only its `allowed_paths` - none by default - because each connection's configuration is locked as it opens (`enable_external_access = false`, the allowed paths, `lock_configuration`): no SQL can reach another file, URL or extension, or change the setting back. Object-storage prefixes are a second lock behind the credentials' own permissions, since an S3-compatible server could resolve `..` in a key; web addresses must name files for the same reason.
- **A local-folder destination is off unless the operator turns it on** (`QUERYAPIGATE_EXPORT_DIRS`), can only be a folder inside one listed there, and is never the server's home - so an administrator can't write a file over the store, or anywhere else the process can reach, through an export. Object-storage destinations need no such list: the credentials bound them. Checked when the destination is saved and again before each write.

- **A connection can't be pointed at the server's own metadata store** (`store_not_allowed`): neither a SQLite/DuckDB/H2 `database` of `queryapigate.db` nor a DuckDB `allowed_paths` folder containing it - which would otherwise hand an administrator the key and token hashes. A PostgreSQL store is kept apart by its own credentials.
- **A `${VAR}` reference can't name one of the server's own `QUERYAPIGATE_*` settings** in a connection or destination - otherwise an administrator could carry the admin key or secret key out in a password sent to a host of theirs. **Nor a credential the server itself may run with** - `AWS_*`, `AZURE_*`, `ARM_*`, `GOOGLE_*`, `GCP_*`, `VAULT_*`, `KUBERNETES_*` - which would reach beyond every allow-list (found in the review for 0.30: `${AWS_SECRET_ACCESS_KEY}` was accepted, and with it the server's whole AWS identity, not just the secret prefixes `QUERYAPIGATE_AWS_SECRET_PREFIXES` opens). Any other variable is reachable until `QUERYAPIGATE_SECRET_ENV_VARS` lists the ones that may be: **set it**, or a credential under a name no rule knows (`DB_ADMIN_PW`) is an administrator's to send. The server warns at startup while references are in use without it.
- **Exports write only under their destination's prefix** (ADR 0004). A destination is a bucket prefix or folder and write credentials, stored like a connection password. Each run writes through a fresh DuckDB connection locked the same way - `allowed_paths` is exactly the prefix plus the one temporary file the rows were staged in - and object paths are checked before DuckDB sees them: a parameter value in a path may only be a plain name, `..` and absolute paths are refused. So a path template, parameter or file name can't reach another partner's prefix. The residual risks: credentials that can write more than the prefix (scope them, as for reading), and *what* is delivered - defining an export decides who receives which data, so only owners and admins may (`exports.write`), and every change to one is audited; a developer may only run one already defined.
- **Older `{name}` text placeholders are string substitution**, not parameterization - restricted to a narrow safe-character pattern (digits and plain text only) specifically because they're substituted directly into the SQL text. Prefer `:name` bound parameters; `{name}` exists for cases (identifiers, table/column names) that can't be a bind parameter in any SQL dialect.

A real bypass was found and fixed, and is worth citing as the honest version of "defence in depth": the fuzz test suite (`tests/test_sql_guard_fuzz.py`) documents that MySQL/ClickHouse's backslash-escape handling inside string literals let a crafted literal hide a `;`-separated second statement from an earlier, dialect-naive version of the guard. It happened not to be exploitable in practice at the time (no driver call ever asked for a smuggled statement's results, and read-only sessions blocked it anyway), but the project's own account of it calls it "a real failure of an explicitly documented guarantee" that "directly undermined auditability" - which is why the guard is now dialect-aware and exercised by a fuzz test on every change, not just trusted to keep working.

Secrets at rest

Connection passwords in the store can be a literal string, an `"${ENV_VAR}"` reference resolved from the process environment at load time (never written back to disk), or - with `QUERYAPIGATE_SECRET_KEY` set - Fernet-encrypted at rest. Encryption is **opt-in, not the default**; without a secret key, a connection password is only as protected as the file/volume it sits in. The key itself must be kept outside `QUERYAPIGATE_HOME` and out of version control - there is no recovery path for an encrypted password without it, by design, the same property that makes it real

encryption rather than obfuscation. API responses always mask stored passwords regardless of how they're stored. The key can be rotated without taking a connection down (ADR 0009): `QUERYAPIGATE_SECRET_KEY` lists the new key first and the old one after it, `queryapigate rekey` re-encrypts every stored password under the new one in a single transaction (all or nothing), and the old key is then removed. A key suspected leaked is replaced the same way; backups taken before the rekey still open with the old key, so they need retiring with it.

The audit trail

The audit log is a hash chain (ADR 0012): each entry's `hash` covers the previous entry's, so changing, removing, inserting or reordering one breaks every hash after it, and `queryapigate audit verify` (and a check every 5 minutes) names the first. Against someone who can write to the store, a plain hash is only consistency - they can recompute it - so with `QUERYAPIGATE_AUDIT_KEY` set the chain is HMAC-signed with a key kept outside the store, and a key once set means every newer entry must carry it. What the chain can't show alone: the newest entries deleted (its end can't know what came after it), and a rewrite by someone holding both the store and the key. Both are why a copy outside - the SIEM stream, or the export file shipped elsewhere - matters. The stream sends each entry with its `seq` and `hash` over TLS - the server's certificate verified, the client's optional - so a later rewrite of the store contradicts what the SIEM already has; its delivery is at least once, and syslog over TCP can lose the batch sent to a SIEM that dies without closing the connection (OTLP, acknowledged per batch, doesn't). `queryapigate audit verify --file` checks the SIEM's copy on its own - repeats from at-least-once delivery dropped only when exact.

Reviewed for 0.32 (ADR 0019). Looked at, against rsyslog, syslog-ng and the OpenTelemetry Collector rather than a receiver of our own: the server's certificate verified against the configured CA; a missing client certificate, and one from another CA, refused by the receiver and noticed before anything is counted delivered; a receiver stopped with a connection open, noticed rather than written into; the cursor and lease surviving it; nothing in a message but the entry as the store holds it - no parameter values, which entries never carry. **Found:** a SIEM's copy of an untouched log failed `audit verify --file` as soon as at-least-once delivery had repeated one entry; fixed - exact repeats are dropped and counted, a repeat that differs is `conflicting_copies`.

The key's guarantee starts at the `audit_key_set` entry the server writes when it first starts with a key. Someone with write access to the store, while the server is stopped, could delete the keyed history and leave an unkeyed one: at the next start the server sees an unkeyed chain, as when a key is first set, and writes `audit_key_set` again. It records how many entries came before it unkeyed (`entries_before_unkeyed`) - a second `audit_key_set`, or one counting entries you know were keyed, is the sign - but only a copy outside the store proves which history is real: the SIEM's already holds the keyed hashes.

Transport

QueryAPIGate does not terminate TLS itself - that is explicitly the reverse proxy's job (Caddy, nginx, Traefik; see DEPLOYMENT.md). An API key travels in a request header, so serving over plain HTTP means the key is recoverable by anyone who can observe the traffic - TLS termination in front of QueryAPIGate is not optional once a key exists.

Response headers don't depend on that proxy (ADR 0010): every response sets `nosniff`, `Cache-Control: no-store` (a cached saved query `private`, never shareable: callers authenticate with `X-API-Key`, which HTTP's shared-cache rule doesn't cover - the 0.22.1 fix) and, on API responses, a CSP that runs nothing and a `no-referrer` policy. HSTS is the deployer's choice (`QUERYAPIGATE_HSTS_MAX_AGE`), sent only over HTTPS. `/docs`, where people type an API key, loads one exact Swagger UI release with Subresource Integrity hashes and a CSP allowing only those files: a compromised CDN, or a bad release of the library, can't run script there. The Console bundles everything it runs.

Tracing (ADR 0010, experimental, off unless an OTLP endpoint is set) sends spans to a collector the deployer runs. Its readers are treated like the audit log's: no data access. So spans carry the route, the saved query's name and the row count - never a parameter value, row, statement, header or credential, and not exception messages, which can carry values; a test enforces it on every front door. A caller's `traceparent` is accepted, which also lets it ask for its trace to be sampled: with the SDK's default sampler, a caller can raise how much is exported. Deployers who care set `OTEL_TRACES_SAMPLER=traceidratio`. The OTLP endpoint's headers (`OTEL_EXPORTER_OTLP_HEADERS`, often a token) are never shown in Settings. Spans do carry two things that can be personal data, as run history does: the caller's address (`client.address`) and who called (`queryapigate.caller` - for a signed-in user `jwt:<subject>`, which may be an email address) - keep the collector's retention to what that allows. An MCP client's tool name is recorded only if it has the shape a saved query's name can have; anything else is `(invalid)`.

On Kubernetes (the Helm chart, ADR 0013)

The chart's pods run as UID 1000, with a read-only root filesystem, every capability dropped, the RuntimeDefault seccomp profile, and no service account token: QueryAPIGate never calls the Kubernetes API, so a compromised pod has no credential for it. The server writes only to `/data` (the store's volume, or an empty directory on PostgreSQL) and `/tmp`. What stays the deployer's:

- **Secrets in a Kubernetes Secret are only as safe as the cluster.** Anyone who can `get secrets` in the namespace, or read etcd unencrypted, reads the API key, the Fernet key and the database URL. Keep that RBAC narrow, and encrypt Secrets at rest. With `secret.create`, the values are also in your values file and in Helm's release record (itself a Secret in the namespace). `existingSecret` keeps them out of both.
- **The chart refuses a secret setting in** `config`: the keys, the database and Redis URLs, the SSO client secret and OTLP headers. `config` is plain text in the Deployment, where `get deployments` reads it.
- **NetworkPolicy is off by default**, because the right rules depend on the cluster. Without it, any pod in the cluster can reach the API and the MCP server. `networkPolicy.enabled` limits who can connect in, and narrows what a pod can reach out to, to `egress` - so a stolen admin key can't point a new connection at an internal address it doesn't list.
- **The chart and image are signed by this repository's release workflow** (from 0.31, ADR 0018): keyless, by digest, the certificate naming `docker.yml` at the release's tag, the signature in Sigstore's public log - with SLSA provenance, an SBOM and a GitHub attestation naming the commit. So a mirror, a moved tag or a look-alike repository can't pass for a release: verify before you run it, and have the cluster refuse anything else (guide 59). What a signature doesn't say: that the code is right. Someone who can push a tag here can still sign

a release - the log then records it. What it's built on is pinned too: base images by digest, every action by commit. Releases before 0.31 aren't signed.

Reviewed for 0.30 (ADR 0017). Looked at: every pod's security context, the service account token left unmounted, `vaultToken` a projected token for Vault alone, the secret settings refused in `config`, the routes, what the NetworkPolicy allows. **Found:** the ingress routes every path to the API server, `/metrics` included - and `/metrics` was open, with API keys', queries' and connections' names in its labels, and each administrator's own name (often their username, for one signed in through SSO). So an install with the ingress on published them. Now `metricsAuth` (on by default) gives `/metrics` a token from a Secret of the chart's own, kept across upgrades, and the ServiceMonitor sends it; and administrators are counted under one `admin` label, as signed-in users are under `jwt`.

Configuration as code (ADR 0014)

A configuration file can change anything an owner can - connections, grants, keys - so applying one is an owner's capability (`config.apply`), and so is reading what applying one would do. What keeps a file safe to put in git, and what's still the deployer's:

- **A file never holds a secret.** A password must be a `${VAR}` reference - a literal is refused - and a reference to the server's own settings or credentials, or to a variable `QUERYAPIGATE_SECRET_ENV_VARS` doesn't list, is refused, as through the API (a connection pointed at a host of the author's would otherwise send it there). Export turns stored values into references. Not checked: a secret inside another field - a JDBC URL with `password=` in it - which is in the file as it is in the store.
- **A key is in a file by its secret's hash.** That's safe to publish only because the secret is 32 random bytes: `config new-key` makes one. A key someone gave a guessable secret could be found from its hash, offline. Two keys can't share a secret.
- **Who can change the file can change the server.** With `QUERYAPIGATE_CONFIG_FILE` (the chart's `configAsCode`), anyone who can edit the file - the repository's main branch, the ConfigMap - has an owner's reach at the next start. Protect the branch and require review; that is the control this feature exists to give.
- **Applying is all or nothing, and audited** - each change `via: config`, under one `config_applied` entry with the file's hash and who applied it. A rolled-back apply leaves no audit entry, in the store or the export file.
- `QUERYAPIGATE_CONFIG_LOCKED` keeps what a file manages to the file for the API and the Console. It doesn't bind someone with a shell on the server: the CLI works on the store directly, as it always has.
- **Pruning deletes only what a file manages**, and what the API deletes is no longer managed - so a later object of the same name, made in the Console, isn't pruned as the file's.
- **A file is capped at a million values**, YAML aliases counted as they expand, so a small file can't be a huge one.

Reviewed for 0.30 (ADR 0017). Looked at: who may diff and apply (`config.apply`, owners); every object checked by the function the API uses for it, so a file can't do what the API refuses; the `--server` CLI, now tested against a real `queryapigate serve` - its key only from `QAG_API_KEY`, no store opened locally; startup apply under the store's write lock, once per file; pruning only what a file manages; the lock's reach. **Found:** a `${VAR}` reference could name the server's own cloud

credentials - `${AWS_SECRET_ACCESS_KEY}` with a host of the author's - through a file as through the API. Fixed in both (Query execution, above): those credentials are always refused, and `QUERYAPIGATE_SECRET_ENV_VARS` limits the rest to what the deployer lists.

Secrets from secrets managers (ADR 0015)

A reference - `${file:...}`, `${vault:...}`, `${aws-sm:...}` - or a connection's `credentials` lets an administrator make the server fetch a secret and use it as a password. The server's identity (its files, Vault token, IAM role) reaches far more than any one connection should, and an administrator could point a connection at a host of theirs to be sent whatever it fetches. So:

- **Nothing is reachable until the deployer says so**: `QUERYAPIGATE_SECRET_FILE_DIRS`, `QUERYAPIGATE_VAULT_PATHS`, `QUERYAPIGATE_VAULT_DATABASE_ROLES`, `QUERYAPIGATE_AWS_SECRET_PREFIXES` and `QUERYAPIGATE_AWS_RDS_HOSTS`, checked when a reference is saved and again when it's used. The store's own folder is never readable, and the server's own `QUERYAPIGATE_*` / `OTEL_*` settings never reachable, by any reference - nor, through `${file:...}`, a file one of them or the server's own credentials name, the Kubernetes and EKS service account tokens, or `/proc`. A file is opened at the real path that was checked, so a symbolic link changed in between can't point it elsewhere.
- **A Vault database role can be bound to the host its credentials may go to** (`reporting-ro@warehouse.internal`), and an RDS IAM token is made only for the listed hosts. **Bind every role**: an unbound one hands its credentials to whatever host a connection names - including through `POST /api/v1/connections/test`, without saving anything.
- **What an allowed reference names, an administrator can still send elsewhere** - a `${vault:...}` password to a host of theirs - exactly as with `${VAR}`: an administrator is trusted with the credentials of the connections they manage (as above). The allow-lists bound which; short-lived credentials bound for how long; a bound role, where.
- **Resolved values are held in memory only**, for at most their lease or `QUERYAPIGATE_SECRET_CACHE_SECONDS`, and never written to the store, a log, a response, a trace or the audit log; errors name the provider and its reason, never a value. A provider that fails is forgiven only within what it gave before; past that, `secret_unavailable`.
- **IAM tokens and Vault credentials are passwords**: `aws-rds-iam` is refused without TLS, and a PostgreSQL or MySQL connection's TLS settings now reach its driver (before 0.28 they were dropped).
- **The Vault token, AppRole secret ID and the chart's projected token** are the server's to protect like `QUERYAPIGATE_SECRET_KEY`. Vault's policy for that identity is the outer limit of what any reference can reach - keep it to the paths and roles connections need. The same goes for the IAM role.
- **Leases are revoked when a process stops** (on SIGTERM too), so a stopped instance's database users don't linger; one that crashes leaves them to their TTL.
- **Why a secret couldn't be had is the operator's, not the caller's**: `secret_unavailable` says only that, and the reference and the provider's error go to the log and to `detail`, which a scoped key calling a saved query doesn't get - they name where the server's secrets live, and its Vault's address.

Reviewed for 0.34 (ADR 0021) - `${VAR}`, `${file:...}`, `${vault:...}` and `vault-database`, against Vault 1.21. Looked at: each allow-list checked when saved and again when used, and Vault paths

and roles matched whole (a `reporting` prefix doesn't reach `reporting-admin`); a file opened at the path that was checked; the cache failing closed, and a failing provider asked at most every 10 seconds; the Vault token's sign-in and renewal, a forbidden answer signing in again only once; what an error tells whom. **Found, and fixed:** a `${file:...}` could read the server's own secrets wherever they were files inside an allowed folder - `QUERYAPIGATE_SECRET_KEY_FILE=/run/secrets/fernet` beside `QUERYAPIGATE_SECRET_FILE_DIRS=/run/secrets`, as the docs suggest - and so have the server's key sent to an administrator's host as a password; and, with a broad folder, `/proc/self/environ` or the service account's token. And `secret_unavailable` told any caller the reference and Vault's own answer. AWS (`${aws-sm:...}`, `aws-rds-iam`) stays experimental, tested only against a stand-in.

What is kept, and the cryptography it rests on (ADR 0016)

Where data goes is the inventory; what bears on threats:

- **Values stay out of what's kept by default.** Run history keeps parameter names, never values; traces never hold one; the access log has paths without query strings and no referrer (`QUERYAPIGATE_ACCESS_LOG_QUERY` puts the query string back - then the log platform holds whatever callers put in URLs). What can still carry a value: an ad-hoc statement's text in history and the log (`QUERYAPIGATE_HISTORY_ADHOC_SQL=hash` keeps both to its SHA-256), a database's own error message in the log, an agent's write arguments for 30 days, an export's watermark, and the response cache's rows. Treat the log platform and the store as holding personal data.
- **Erasure is an owner's, and it isn't silent.** `history forget` deletes a caller's runs and usage counts - any caller, a key's too, so an owner could clear the record of what a key ran. The audit log records each erasure, its counts and the caller's SHA-256 (key names are few: the hash names the key to anyone who has the list), so it can be seen, not undone. The audit log itself can't be erased from without breaking its chain (ADR 0012).
- **The cryptography is FIPS-approved algorithms only**, enforced by a test over the code; whether it runs in a validated module is the platform's (a FIPS-mode OpenSSL, the distribution's `cryptography`). The page names what is outside that boundary - the PostgreSQL and MySQL drivers' binary wheels, DuckDB's TLS, a JDBC or H2 connection's Java runtime - and CI runs the whole suite under OpenSSL's FIPS provider.

Live events (ADR 0020)

`GET /events` - on the main server, and `queryapigate events` for many clients - streams each run as it happens: which query, on which connection, by which key, how it went, and an ad-hoc run's SQL as history keeps it. So:

- **Never public.** A real key or a signed-in user's token, in a header - never the URL, where it would reach access logs - unless the server runs with no keys at all. A stream is checked again every minute, so a revoked key or an expired token loses it within about a minute and a quarter.
- **The admin key and data administrators see every run; anyone else only their own** - the runs made with that key, or by that user, since the key was created. Filters only narrow that, and their values are bound, never SQL. Key names can't hold the `:` of `jwt:<subject>`, and an administrator's name can't be a key's.

- **Resource use is bounded**: `QUERYAPIGATE_EVENTS_MAX_CONNECTIONS` streams a process, `QUERYAPIGATE_EVENTS_MAX_STREAMS` on the main server (each a request thread), `QUERYAPIGATE_RATE_LIMIT` before a key is even checked, 10 seconds and 100 headers for a request, and a client that stops reading cut off once 1,000 events wait for it. Not bounded: how many of those streams one key holds.
- **Kept for a while, and forgettable**: the event log keeps `QUERYAPIGATE_EVENTS_RETENTION_HOURS` (24 by default), and `history forget` erases a caller's events with their runs.

Reviewed for 0.34 (ADR 0021). Looked at: authentication on both endpoints and the recheck; who sees what, live and on a resume, for keys, signed-in users and each administrator role; the filters' SQL; the request parser's limits; CORS and the security headers; `/metrics` behind its token. **Found, and fixed:** a deleted key's name can be given to a new key, and the event log kept the old key's runs under it - the new key could resume from 0 and read its predecessor's runs of the last day. A key now sees no event from before it was created.

Rate limiting and abuse prevention

The rate limiter throttles both by source IP (server-wide) and, additively, by API key (`rate_limit` grant). Without `QUERYAPIGATE_REDIS_URL` its state is in each process's memory, so running more than one process silently multiplies the effective limit. With it, every instance and process counts against one budget in Redis (an atomic token bucket, on Redis's own clock); if Redis becomes unreachable, each process falls back to its own counting - limits stay enforced, per process, never lifted - and the `rate_limits_not_shared` alert and a metric say so. `QUERYAPIGATE_TRUST_PROXY` tells the server how many reverse-proxy hops to trust when resolving the real client IP from `X-Forwarded-For` - left at its default of `0` with a direct-connecting client, or set to the actual hop count behind a proxy; getting this wrong in either direction lets a client forge its apparent IP and evade both the rate limiter and `allowed_ips` checks.

A rate limit counts calls, not what they cost, so a key under its rate can still hold every pooled connection with slow queries and starve everyone sharing them. A key's or role's own `max_concurrent`, `query_timeout` and `max_rows` (ADR 0011) bound that: queries running at once (a stream counts until its last row), how long each may run, how many rows a call returns - each only ever narrowing the server's limits. Counted in Redis across instances, with leases so a crashed process can't hold a caller's slots forever; without Redis, per process, as rate limits are. Usage quotas cap a caller's calls and rows per day or month. They're a contract's meter, not a hard security boundary: a caller overshoots by its calls in flight, and with several instances but no Redis by up to a quota per instance in a short burst. `max_concurrent` and the rate limit are what bound a burst.

Reviewed for 0.30 (ADR 0017). Looked at: who sets a quota (`access.write`, never the caller; a configuration file's role through the same validation); that it's checked before the response cache, so a cached answer counts and is refused when used up; that MCP tool calls and REST calls share one counter per caller, and per user for a role's signed-in callers; the Redis keys (`qag:quota:<caller>:<period>`), where a period always ends in its date, so two callers' keys can't meet whatever their names; that `forget` matches Redis keys with the caller's glob characters escaped; who reads every caller's counts (`GET /api/v1/usage/quotas`, `observe` - callers' names, personal data for a signed-in person, as in run history). Found nothing to fix; the documentation now says

what doesn't count (exports, live events, MCP resources, the Management API) and that an agent's write request does.

CORS

Cross-origin access is off unless `QUERYAPIGATE_CORS_ORIGINS` is explicitly set, and CORS is a browser-only control, not authentication - it stops a browser tab from making credentialed requests on a site's behalf, it does nothing against a direct HTTP client. Setting it to `*` without also requiring an API key means any site's script can call the API directly from a visitor's browser.

Audit logging and data at rest

Every configuration change (`record_audit(actor, action, target, changes)`) is appended to an `audit_log` table - the actor (the administrator's name, or `admin` for the shared key), how they authenticated (`via`: their own token, the break-glass key, the CLI), the action, the target, and a JSON diff of what changed. Only administrators can read it, and nothing in the API can alter it; a scoped key cannot see it. A separate `execution_history` table records individual query runs (caller, row count, timing), independent of the audit log's configuration-change trail. **The audit log is tamper-evident** - a hash chain, signed with `QUERYAPIGATE_AUDIT_KEY` (The audit trail); keep a copy outside the store too (the SIEM stream, or `QUERYAPIGATE_AUDIT_LOG_EXPORT_FILE` shipped elsewhere), since only that catches its newest entries deleted. **Run history is not**: a plain table, which someone with direct database access (not API access - the API never exposes a way to alter history) could edit without detection.

Data at rest: what one file actually holds

Everything lives in a single SQLite file, `queryapigate.db`, inside `QUERYAPIGATE_HOME` - connections (with passwords plaintext, `${VAR}`-referenced, or Fernet-encrypted per the setting above), saved queries and every version's full SQL text, execution history (including row counts), API key *hashes* (never secrets) and grants, roles, administrators and their admin tokens' *hashes*, and the audit log. Protecting this one file - filesystem permissions, backup encryption, who can reach the volume - is entirely the operator's responsibility; `QUERYAPIGATE_SECRET_KEY` only ever covers connection passwords, not the rest of what the database holds.

MCP server

`queryapigate mcp` (a separate process from `queryapigate serve`) reuses the identical authentication and authorization path - the same `X-API-Key` header, the same `apikeys.authenticate()`, the same per-key `connections`/`queries`/`collections` scoping `GET /catalog` already enforces, the same open-access fallback when no server key is configured. A saved query that writes is a tool only when its author marked it for agents and the caller's grant allows the write - and then each write is approved by a person first, unless it only inserts (below); otherwise it isn't listed, and isn't run when called by name or by version (fixed in 0.20.1). `execute_sql` never writes. One transport-specific note: an MCP client has to be configured to send the key as a request header (Streamable HTTP clients can) - never as a URL query parameter, which would leak it into access logs. The MCP server doesn't read `QUERYAPIGATE_TRUST_PROXY`: behind a proxy it sees the proxy's address for every caller, so `allowed_ips` and the server-wide rate limit can't tell MCP callers apart there.

Every request to `/mcp` is authenticated before the MCP session sees it, and refused at the HTTP level (`401`, `403`, `429`) - nothing of the protocol is reachable without a credential once one is required.

OAuth for AI clients (ADR 0006)

With `QUERYAPIGATE_MCP_RESOURCE_URL` and `QUERYAPIGATE_MCP_AUTH_SERVER` set, the MCP server is an OAuth 2.1 resource server: it checks tokens; the organisation's identity provider issues them. What that protects, and what it doesn't:

- **Tokens are bound to the MCP server.** A token's `aud` must be the MCP server's own URL, its `iss` the configured provider, its signature one of the provider's published keys with that key's own algorithm (never `none`, never an HMAC "signed" with a public key), its `exp` in the future, and it must carry `QUERYAPIGATE_MCP_SCOPE`. A token issued for any other application of the same provider - including this server's own REST API - is refused, and REST refuses a token naming the MCP server; startup refuses a REST audience that includes the MCP URL. A leaked token works against one service only, until it expires.
- **No token is passed on.** Databases are reached with each connection's own credentials; a caller's token never leaves the MCP server.
- **The provider's keys are found through its metadata**, fetched over `https://` only (a redirect to plain HTTP is refused), from documents that must name the configured issuer as their own (RFC 8414) - a metadata document claiming to be another issuer isn't used. A provider that can't be reached means every token is refused, never accepted unchecked.
- **One person, one name.** An agent's runs are under `jwt:<sub>` - the name that person has signed in to REST - when REST trusts the same provider; under `mcp:<sub>` when REST trusts another, so that two people whose `sub`s happen to match never share a name, rate limit, history or event stream.
- **What remains with the provider:** who may get a token for the MCP server, with which scope and which role claim, is the provider's configuration - QueryAPIGate trusts what a valid token says. Review that the role claim can't be set by users themselves (a self-editable profile attribute mapped into the token would let anyone pick a role), and how clients register: anonymous Dynamic Client Registration lets anyone register a client in your realm.
- **Revocation** is the provider's too: a token is valid until it expires. Keep access tokens short-lived; disabling or narrowing the role in QueryAPIGate takes effect on the next request.

Reviewed for 0.30 (ADR 0017). Looked at: the algorithm taken from the provider's published key, never the token's header, and only an asymmetric one; `iss`, `aud` and `exp` required; tokens for REST and for the MCP server kept apart in both directions; the metadata's issuer and `https://` endpoints, and no redirect off `https://`; one name per person across two issuers (`mcp:<sub>`); the role claim naming only roles that exist. The signing keys come through PyJWT's own client, which follows no redirect at all - from 2.14, the floor CI tests; a test now holds it to that. Found nothing to fix.

Agents that write (ADR 0008)

- **Only what an author marked, only data changes.** A write tool is a saved query marked `agent_writes` - one `INSERT`, `UPDATE` or `DELETE`, never DDL, checked when saved - for a caller whose per-query grant allows the write, with `QUERYAPIGATE_ALLOW_WRITES` on. The grant, the

mark and the server's setting are all checked again on every call, so taking any of them back stops the next one, approved or not.
- **A named person approves each write, and that write only.** The approval is bound to the caller, the query's version and the exact arguments (by hash), runs once, and expires. A model can't satisfy it by itself: the credential that asked can never approve (`own_write`), so an MCP client holding an approver's token or the shared key can't approve its own writes - give agents their own keys.
- **A ceiling on what any write changes.** It runs in a transaction, rolled back when it changed more than its `max_affected_rows` - counted when it runs, not estimated when approved. On MySQL that needs transactional (InnoDB) tables.
- **Insert-only queries skip the approval**, and can add rows but never change or remove one - choose that mode only where an added row can do no harm on its own.
- **What remains:** an approver judges what they're shown - the SQL and the arguments, not which rows; and a description, a row or a document the agent read can still talk it into *asking* for a write. The approval is where that stops.
- **What's recorded:** every request 30 days, with who decided and how they signed in; the run in history with its approver; the audit log with the arguments' names and hash, never their values, since auditors read it.

Reviewed for 0.30 (ADR 0017). Looked at: an approval bound to the caller, the query's version and the arguments' hash; approved to running in one conditional update, so it runs once however many calls race; deciding guarded the same way; the credential that asked never the one that approves; the grant, the mark and the server's setting checked again on the call that runs it, before the approval is claimed; at most 100 pending per caller; insert-only's idempotency key scoped to the caller and the tool. Found nothing to fix.

What agents are told, and what they did

- **Descriptions are instructions-shaped data.** A table's or column's description, and a connection's notes, are read by agents as context (descriptions). Whoever can write them can steer what an agent does with the data it can reach - not what it can reach. Writing them takes `connections.write` (owners and admins), and every change is audited. **Importing from the database's comments** brings in text written by whoever has `COMMENT` rights in the database: review an import (it says `source: "database"` on each) before relying on it.
- **Descriptions reach no further than the schema.** A caller sees those of the tables it can reach, and a caller restricted to some tables never gets the connection's notes, which may name any table.
- **Agent usage** (`GET /api/v1/usage/agents`) shows administrators each agent's ad-hoc SQL, as run history does - it is readable by every administrator role, auditors included. With `QUERYAPIGATE_HISTORY_ADHOC_SQL` set to `hash` or `none`, it shows hashes or nothing.

Non-goals: what this does not defend against

Being explicit about these is as important as everything above:

- **QueryAPIGate is not a network perimeter or a web application firewall.** It assumes it is reachable only by callers who should be attempting requests at all - a reverse proxy, network

policy, or VPN is still the operator's job for anything beyond that.

- **An owner, admin or developer is equivalent to database-superuser-scoped-to-your-connections** for data. Roles limit what each can *change* in QueryAPIGate, not which rows they can read; database-level least-privilege grants on the connection account itself are the actual backstop here, not a QueryAPIGate setting.
- **The regex-based read-only/single-statement guard is defence in depth, never a substitute for least-privileged database credentials.** Connect with an account that only has the grants the API actually needs; assume the guard could someday have another gap like the one already found and fixed.
- **Several instances are one service only on a shared store and Redis.** With a PostgreSQL store and `QUERYAPIGATE_REDIS_URL`, every instance shares keys, connections, the audit trail and rate limits; without Redis, each counts limits on its own (the `instances_not_shared` alert says so), and two replicas each with their *own* SQLite volume are two separate servers with their own keys, connections and audit trail. `/metrics` is always per process - scrape each one.
- **Run history is not tamper-evident, and the audit log's chain has limits.** The chain shows an edit, but not its newest entries deleted, nor a rewrite by someone holding both the store and `QUERYAPIGATE_AUDIT_KEY` - a copy outside the store (the SIEM stream) is what proves those (The audit trail).
- **Connection metadata (hosts, ports, usernames, database names) is visible to every administrator role** - `GET /api/v1/connections` is administrators-only, and passwords are always masked, but the rest is not treated as secret.
- **Names can be probed.** A saved query a key can't reach answers `403` naming its connection; a name that doesn't exist answers `404`. Don't put anything sensitive in query or connection names.
- **An agent may be steered by what it reads.** Data in results, descriptions and notes all become part of an AI agent's context. QueryAPIGate limits what an agent can reach and hides what masking covers; it can't make an agent ignore instructions planted in data it is allowed to read.
- `/metrics` **needs no API key** - a scraper shouldn't hold one - **and is open until** `QUERYAPIGATE_METRICS_TOKEN` **is set**: then only a request with that bearer token, or an administrator who may observe, reads it. Its labels name API keys, queries and connections (administrators and signed-in users only as `admin` and `jwt`): set the token wherever the server can be reached from outside. The Helm chart does.

Where to go from here

- Where data goes - what is kept, where, for how long and who reads it; the record of processing this page's controls protect.
- SECURITY.md - the operational hardening checklist this page explains the reasoning behind, and where to report a real vulnerability.
- DEPLOYMENT.md - TLS termination, secrets in an env file, backups, and the single-worker rationale in a real production Docker setup.
- API.md - the full, field-by-field reference for every grant on a scoped API key.

Where data goes

What QueryAPIGate keeps, where, for how long, who can read it - and what it never keeps. For a record of processing (GDPR Article 30), a data-classification review, or a security questionnaire's data section (ADR 0016). Every table of the store, every setting that bounds what is kept, and every place data leaves the process is on this page: a test fails if one isn't. Walkthrough, answering a questionnaire from a running server: Answer a security questionnaire about data and cryptography.

What passes through, and isn't kept

QueryAPIGate sits between callers and databases. **The rows a query returns pass through and aren't kept**: they're streamed or paged to the caller, after column masking, and written nowhere - with two exceptions you turn on: the response cache and exports.

Parameter values aren't kept either. A saved query's parameters are bound, not written into the SQL; run history records which query ran and with which parameter *names*, never their values; traces never hold a value (ADR 0010); the access log leaves the query string out (`QUERYAPIGATE_ACCESS_LOG_QUERY`, below).

Personal data it handles

Category	Where it comes from	Why it's kept
Callers' names	API key names; `jwt:<subject>` for a signed-in user - often an email address	To say who ran what, apply their limits and quotas
Administrators' names and email addresses	Created by an owner, or from SSO	To say who changed what
IP addresses	Every request	Rate limits, IP allow-lists, the access log, traces
Whatever the connected databases hold	Your databases	Passes through - kept only where listed below
An agent's write arguments	An AI agent asking to write (ADR 0008)	So a person can approve exactly what runs

The store

`queryapigate.db`, or the PostgreSQL store. Read through the Management API by administrators, each to what their role's capabilities allow (Administrators and admin roles).

Table	Holds	Personal data	Kept for	Read by
`connections`	Each connection's settings; its password encrypted (`QUERYAPIGATE_SECRET_KEY`) or a reference	-	Until deleted	`connections.read` (passwords masked)

Table	Holds	Personal data	Kept for	Read by
`saved_queries`, `saved_query_versions`	Saved queries and every version: SQL, parameters, descriptions, the author's name	Author	Until deleted	`queries.read`
`descriptions`	Descriptions of tables and columns, and who wrote each	Administrator's name	Until changed	`connections.read`; agents
`execution_history`	Each run: when, which query and version, the caller, the transport, rows, time, status, QueryAPIGate's error message; an ad-hoc run's parameter *names* and its SQL as `QUERYAPIGATE_HISTORY_ADHOC_SQL` says (`text`, `hash`, `none`)	Caller	The newest `QUERYAPIGATE_HISTORY_LIMIT` runs per version and `QUERYAPIGATE_HISTORY_ADHOC_LIMIT` ad-hoc runs - or `QUERYAPIGATE_HISTORY_RETENTION_DAYS` days; `QUERYAPIGATE_HISTORY_SAMPLE_RATE` keeps only a share	`observe`
`events`	The event log (ADR 0020): each run as `/events` sends it - the same fields as its `execution_history` entry, every run whatever the sample rate	Caller	`QUERYAPIGATE_EVENTS_RETENTION_HOURS` (24 by default)	Each key its own events, through `/events`; the admin key every one
`usage_counters`	Calls and rows per caller per day and month (quotas, ADR 0011)	Caller	400 days	`observe`
`audit_log`, `audit_chain`, `audit_delivery`	Every administrative change: who, how they signed in, what changed (passwords only as "changed"); its hash chain; how far each SIEM output has got	Administrator's name	The newest `QUERYAPIGATE_AUDIT_LOG_LIMIT` entries - more while a SIEM output is behind	`observe`
`api_keys`, `roles`	Keys' names, grants and the SHA-256 of their secrets; when each was last used	Key names	Until deleted	`access.read` (never a secret)
`administrators`, `admin_tokens`	Administrators' names, roles, email addresses, SSO subjects, when last seen; their tokens' hashes	Name, email	Until deleted	`admins.read`

Table	Holds	Personal data	Kept for	Read by
`admin_sessions`	Console sessions through SSO, by the hash of their id	Administrator's name	`QUERYAPIGATE_SSO_SESSION_HOURS`	-
`sso_logins`	A sign-in under way: its state and PKCE verifier	-	10 minutes	-
`write_requests`	An agent's write: its arguments, who asked, who approved or rejected it, the result	Caller, approver; the arguments may be personal	30 days	`writes.approve`
`destinations`, `exports`	Where exports go, with credentials encrypted or as references; each export's watermark - the last value of its incremental column it delivered	The watermark may be a value	Until deleted	`exports.read` (secrets masked)
`instances`	Each running process: host name, process id, version	-	1 day after it stops	`observe`
`config_managed`, `config_applied`	What a configuration file manages, and each file applied, by its hash (ADR 0014)	-	Until deleted	`observe`
`server_secrets`	Keys the server makes for itself - masking's `hash` key, unless `QUERYAPIGATE_MASK_KEY` is set	-	For the store's life	-
`schema_version`	The store's schema version	-	-	-

The response cache and Redis

Where	Holds	Kept for
The response cache (in the process, or Redis with `QUERYAPIGATE_REDIS_URL`)	Responses of saved queries that set `cache_ttl` - **rows**, after masking - by query, version, parameter values, format and page; never by caller	The query's `cache_ttl`
Redis's rate-limit counters	Calls per key and per client address in the current window	The window
Redis's quota counters	Calls and rows per caller per period	40 days
Redis's limits	Queries running now, per caller (ADR 0011)	While they run

Where data is sent

To	What	Bounded by
The log (stdout)	One line per request: method, path, status, time, the caller's name, the request id - **not the query string**; each statement's SQL, or with `QUERYAPIGATE_HISTORY_ADHOC_SQL` `hash` or `none`, its SHA-256; warnings and errors - a database's own error message among them, which can quote a value. `QUERYAPIGATE_JSON_LOGS` for JSON	Your log platform
The access log (the image's gunicorn, `queryapigate serve`)	Client address, time, method, **path without the query string**, status, size, time taken, request id, user agent - never the referrer. `QUERYAPIGATE_ACCESS_LOG_QUERY=1` keeps the query string	Your log platform
Traces (`OTEL_EXPORTER_OTLP_ENDPOINT`)	Spans: the route, the saved query's name, rows, the caller's name and address - never a parameter value, row, statement or credential (Tracing)	Your collector
`/metrics`	Counts and timings by connection, query and API key name; every signed-in user under one `jwt` label, every administrator under one `admin`	Your Prometheus - and whoever can reach `/metrics`, unless `QUERYAPIGATE_METRICS_TOKEN` is set (the Helm chart sets it)
The audit log's copies (`QUERYAPIGATE_AUDIT_LOG_EXPORT_FILE`, `QUERYAPIGATE_AUDIT_SYSLOG_URL`, `QUERYAPIGATE_AUDIT_OTLP`)	Every audit entry, with its place in the chain	Your file and SIEM
Live events (`/events`)	Each run as it happens: the same fields as run history	Not kept
AI agents (MCP)	Rows, after masking, up to `QUERYAPIGATE_MCP_MAX_ROWS` a call; descriptions	What the agent keeps
Exports (destinations)	A saved query's rows, after the export's masking, to the bucket or folder it names	Your storage
Secret providers (Vault, AWS)	Requests for the secrets connections name (ADR 0015)	-

What a deployer controls

- **Run history:** how much is kept (`QUERYAPIGATE_HISTORY_LIMIT`, `QUERYAPIGATE_HISTORY_ADHOC_LIMIT`, `QUERYAPIGATE_HISTORY_RETENTION_DAYS`), whether every

run is (`QUERYAPIGATE_HISTORY_SAMPLE_RATE`), and what of an ad-hoc statement (`QUERYAPIGATE_HISTORY_ADHOC_SQL`).
- **The audit log:** how many entries the store keeps (`QUERYAPIGATE_AUDIT_LOG_LIMIT`); its copies elsewhere are yours to keep for as long as your obligations say.
- **What leaves:** column masking, by key, role and export; the response cache, per query; the access log's query strings; tracing, when an endpoint is set.
- **A person's data, on request:** below.

A person asks for their data to be erased

A person who signs in to call QueryAPIGate is named `jwt:<subject>` - their email address, often - in run history and usage counts. To erase them:

```
queryapigate history forget jwt:alice@example.com --dry-run   # how many runs, events and usage periods
queryapigate history forget jwt:alice@example.com
```

or `DELETE /api/v1/history?caller=jwt:alice@example.com` (owners only, `history.forget`). Their runs, events and usage counts - in the store and in Redis - are deleted, and the erasure is recorded in the audit log as `forget_caller` with the caller's SHA-256 and the counts, not their name. That record is pseudonymous: someone who already knows the name can confirm that it was erased - which is how you show a request was met - but it names no one to anyone who doesn't. Their quota's count for the period starts again.

With several instances, runs another instance has queued but not yet written - for at most `QUERYAPIGATE_HISTORY_FLUSH_INTERVAL`, a second by default - can land just after. Check with `--dry-run` a few seconds later, or run it again: it finds what arrived since.

What isn't erased, and what to say:

- **The audit log** names administrators acting in their role - keep it under legitimate interest, for as long as your accountability obligations require. It is hash-chained (ADR 0012): an entry can't be removed without breaking the chain. A person who isn't an administrator isn't in it.
- **Write requests** - an agent's, naming the caller - are kept 30 days, then deleted.
- **Logs, traces and SIEM copies** are in your platforms, under their retention.
- **The connected databases** are yours: QueryAPIGate doesn't hold their data.

Cryptography

Every algorithm QueryAPIGate uses is FIPS-approved (FIPS 140-3, SP 800-131A); a test fails if the code gains one that isn't - MD5, SHA-1, BLAKE2, or `random` for anything that must be unpredictable.

Protects	Algorithm	Through
Connection and destination passwords at rest (`QUERYAPIGATE_SECRET_KEY`)	Fernet: AES-128-CBC and HMAC-SHA256	`cryptography`

Protects	Algorithm	Through
API keys, admin tokens, Console sessions, sign-in state - stored as hashes	SHA-256	`hashlib`
The audit log's chain (`QUERYAPIGATE_AUDIT_KEY`)	HMAC-SHA256 (SHA-256 without a key)	`hmac`, `hashlib`
Column masking's `hash`	HMAC-SHA256	`hmac`
PKCE for Console sign-in	SHA-256 (`S256`)	`hashlib`
Secrets it makes - keys, tokens, verifiers, ids	The operating system's random source	`secrets`
Signed-in users' tokens	RS256 with a JWKS, HS256 with a shared secret - or `QUERYAPIGATE_JWT_ALGORITHMS` (keep to RS*, PS*, ES*, HS*)	PyJWT, `cryptography`
TLS: to Vault, the SIEM, the OTLP collector, Redis	As the platform's OpenSSL negotiates	Python's `ssl`
RDS IAM tokens	AWS Signature Version 4: HMAC-SHA256	botocore
Cache keys, ETags, configuration files' hashes - not secrets	SHA-256	`hashlib`

`random` is used only where nothing depends on it being unpredictable: which runs history samples, and the example database, from a fixed seed.

On a FIPS-validated module

Validation is a property of the cryptographic module, not of the application: QueryAPIGate is FIPS-ready when it runs on a validated OpenSSL in FIPS mode, and that is a choice of platform.

- **Use a distribution whose OpenSSL is validated, in FIPS mode** - Red Hat Enterprise Linux 9 (booted with `fips=1`, or `fips-mode-setup --enable`), or Ubuntu Pro FIPS. A container uses its host's mode.
- **Python and `cryptography` from the distribution, not PyPI** - PyPI's `cryptography` wheel carries its own OpenSSL, which is not a validated module. On RHEL 9:

  ```
  dnf install python3.12 python3.12-pip python3.12-cryptography
  python3.12 -m venv --system-site-packages /opt/queryapigate
  /opt/queryapigate/bin/pip install "queryapigate[server,postgres]"   # keeps the distribution's cryptography
  ```

- **Check it:** both must say so.

```
/opt/queryapigate/bin/python -c "import hashlib; hashlib.md5(b'x')"           #
ValueError: unsupported
/opt/queryapigate/bin/python -c "from cryptography.hazmat.backends.openssl import
backend; print(backend._fips_enabled)"
```

- **Database drivers bring their own TLS.** `psycopg2-binary` carries its own libpq and OpenSSL: on a FIPS platform, build `psycopg2` from source against the system's libpq (`dnf install libpq-devel gcc python3.12-devel`, then `pip install psycopg2 --no-binary psycopg2`). `mysql-connector-python`'s C extension carries its own OpenSSL too: MySQL connections are outside the validated boundary unless your connector build uses the system's. Also outside QueryAPIGate's control: DuckDB's own TLS for remote files, and a JDBC or H2 connection's Java runtime - use a FIPS-configured JRE, or don't use them there.
- **What your platform's FIPS policy refuses is refused** - a JWT algorithm in `QUERYAPIGATE_JWT_ALGORITHMS` its module doesn't offer, a TLS cipher it doesn't allow - with an error, never a weaker fallback.

Tested in CI: a job runs the whole suite in Red Hat UBI 9 with OpenSSL's FIPS provider forced (`OPENSSL_FORCE_FIPS_MODE=1`), Python 3.12 and `cryptography` from the distribution - and first checks that FIPS is really in force: MD5 refused, `cryptography` on the system's OpenSSL, in FIPS mode. RHEL 9's SQLite is 3.34, older than the 3.35 that `RETURNING` and `DROP COLUMN` need: an agent's write with `RETURNING` to a SQLite database fails there, and the two tests that need 3.35 skip.

Security Policy

QueryAPIGate executes SQL on behalf of whoever can reach it, so please treat it as a sensitive piece of infrastructure. This page is an operational checklist; for the reasoning behind each control - what's actually being defended against, how it works, and where it stops - see documentation/THREAT_MODEL.md.

Reporting a vulnerability

Please **do not open a public issue**. Report it privately through GitHub's "Report a vulnerability" form on this repository, or email arcswdev@gmail.com. You can expect an acknowledgement within a few days.

Supported versions

Security fixes are made against the latest released version.

Verify a release

From 0.31, every image (`ghcr.io/anantharajuc/queryapigate`, and its `-h2` variant) and the Helm chart (`oci://ghcr.io/anantharajuc/charts/queryapigate`) is signed keyless with cosign, by digest, by this repository's release workflow - and carries SLSA provenance, an SBOM and a GitHub attestation (ADR 0018). A signature you can trust names exactly this identity:

```
cosign verify ghcr.io/anantharajuc/queryapigate:0.31.0 \
  --certificate-identity
https://github.com/AnanthaRajuC/QueryAPIGate/.github/workflows/docker.yml@refs/tags/v0.31
.0 \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com
gh attestation verify oci://ghcr.io/anantharajuc/queryapigate:0.31.0 --owner AnanthaRajuC
```

The same for the chart, `ghcr.io/anantharajuc/charts/queryapigate:0.31.0`. Anything else - another repository, another workflow, another tag, or no signature - isn't a release of this project. To have a cluster refuse it, and to read the SBOM: guide 59. Releases before 0.31 aren't signed.

Hardening checklist for deployments

- Set `QUERYAPIGATE_API_KEY`, and put the service behind TLS (a reverse proxy) - the key is sent in a header. It is the shared, full-access admin key; give anyone or anything that only needs to run queries a scoped key instead (`POST /api/v1/api-keys`), limited to specific connections and, separately, to read-only access.
- Give the people (and pipelines) who manage the server their own admin tokens, with the narrowest role that fits - `queryapigate admins create NAME --role owner` for the first - and then keep the shared key sealed as a break-glass key, or remove it (guide 43).

- Leave `QUERYAPIGATE_ALLOW_WRITES` unset unless you really need writes, and connect with a database account that only has the privileges the API should have. The read-only guard - including the single-statement check it relies on - is defence in depth, not a substitute for grants; it is dialect-aware (MySQL/ClickHouse honour backslash escapes in string literals by default, PostgreSQL/SQLite/H2 do not) and is exercised by a fuzz test suite (`tests/test_sql_guard_fuzz.py`), but a regex-based guard can never be a full SQL parser for every server mode.
- Keep database passwords out of the store: use `"password": "${ENV_VAR}"` references, or set `QUERYAPIGATE_SECRET_KEY` to encrypt them at rest (guide 22).
- Set `QUERYAPIGATE_SECRET_ENV_VARS` to the variables your connections reference, and `QUERYAPIGATE_METRICS_TOKEN` wherever `/metrics` can be reached from outside (guide 58).
- Run only signed releases (above), by digest.
- Do not enable `QUERYAPIGATE_DEBUG` on a reachable host.
- Set `QUERYAPIGATE_RATE_LIMIT` on anything reachable beyond a trusted network; it also throttles API key guessing.
- Behind a reverse proxy, set `QUERYAPIGATE_TRUST_PROXY` to the number of proxies so limits apply per real client - and leave it at `0` when clients connect directly, otherwise they can forge `X-Forwarded-For` to evade the limit.
- Only enable `QUERYAPIGATE_CORS_ORIGINS` for sites you control, and never `*` without an API key.
- Keep the query time limit (`QUERYAPIGATE_QUERY_TIMEOUT`, 30 seconds by default) and `QUERYAPIGATE_MAX_PAGE_SIZE` so one expensive request cannot monopolise the service; setting the timeout to `0` removes that protection.
- Prefer bound `:name` parameters over `{name}` text placeholders.
- Note that `GET /api/v1/connections` reveals hosts, ports, users and database names (passwords are masked).

Production deployment (Docker)

A production-shaped setup: the published image, a real reverse proxy terminating TLS, secrets kept out of the compose file, a persistent volume with a backup job, and Prometheus/Grafana wired to the metrics QueryAPIGate already exposes. `docker-compose.yml` in the repo root is a **one-command demo** (open to localhost, a throwaway Postgres, a demo API key) - not this. Start from the compose file below instead.

If you're installing from source or pip rather than Docker, see Installation and setup instead; the gunicorn command and every environment variable there apply here too - this page is about the container and everything around it.

1. The image

```
docker pull ghcr.io/anantharajuc/queryapigate:latest
```

Tag	Contents
`X.Y.Z`, `latest`	QueryAPIGate with every optional feature: the PostgreSQL, MySQL, ClickHouse, DuckDB and MongoDB drivers (SQLite is built in), `allowed_tables`, password encryption, signed-in users (JWT), the Redis cache and `queryapigate mcp`
`X.Y.Z-h2`, `latest-h2`	The same plus Java and the H2 driver - also the variant for a generic `jdbc` connection (mount your vendor's jar)

Pin to `X.Y.Z` in production, not `latest`. An upgrade should be a deliberate, one-line version bump you control, not something that happens on the next `docker compose pull`. Bump it the same way you'd bump any other dependency - read the Changelog first, especially around a minor version bump (this project's pre-1.0 equivalent of a major one).

The image already does the things a production container should: runs as a non-root user (`uid 1000`), ships a `HEALTHCHECK` against `/health`, and starts gunicorn with `--worker-class gthread` and one worker (see Scaling out below) - none of that needs reconfiguring.

2. Compose file

```
# docker-compose.yml
services:
  queryapigate:
    image: ghcr.io/anantharajuc/queryapigate:1.2.3   # pin a real version - see above
    restart: unless-stopped
    env_file: .env                                    # secrets live here, never inline -
see below
    environment:
      QUERYAPIGATE_TRUST_PROXY: "1"                    # one hop: the reverse proxy below
    volumes:
      - queryapigate-data:/data
    networks:
      - internal
    deploy:
      resources:
        limits:
          cpus: "1.0"
          memory: 512M

  # Reverse proxy: TLS termination, the only container with a published port. Caddy shown
here for its
  # automatic HTTPS; nginx/Traefik work the same way - proxy to queryapigate:5000 with
the real client
  # address forwarded.
  proxy:
    image: caddy:2-alpine
    restart: unless-stopped
    ports:
      - "443:443"
      - "80:80"                                        # ACME HTTP-01 challenge + HTTP-
>HTTPS redirect
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile:ro
      - caddy-data:/data
    networks:
      - internal
    depends_on:
      - queryapigate

  # Optional - see "Shared response cache" below. Skip both this service and
QUERYAPIGATE_REDIS_URL if
  # cache_ttl either isn't used or the built-in per-process cache is enough.
  redis:
    image: redis:7-alpine
```

```
    restart: unless-stopped
    networks:
      - internal

networks:
  internal:

volumes:
  queryapigate-data:
  caddy-data:
```

```
# Caddyfile
api.example.com {
    reverse_proxy queryapigate:5000
}
```

That's it for TLS - Caddy requests and renews a real certificate automatically for `api.example.com` the first time it starts, as long as DNS already points at this host and ports 80/443 are reachable from the internet (for the ACME challenge). Swap in nginx or Traefik if you already run one of those; the only requirement is that whatever terminates TLS forwards to `queryapigate:5000` and QueryAPIGate is told how many proxy hops sit in front of it via `QUERYAPIGATE_TRUST_PROXY`, so rate limiting and redirects use the real client address and scheme rather than the proxy's.

3. Secrets (`.env`)

```
# .env - not committed; chmod 600
QUERYAPIGATE_API_KEY=<a long random string>
PROD_DB_PASSWORD=<your database password>
```

`QUERYAPIGATE_API_KEY` is the admin key - full access, sent as the `X-API-Key` header. Generate a real one (`openssl rand -hex 32`), not a word you'll remember; create scoped keys for anything that only needs to run queries (`POST /api/v1/api-keys`, from the admin UI or the API - see API.md). Then give each person who manages the server their own admin token and role, and keep the shared key as a break-glass key - `docker compose exec queryapigate queryapigate admins create alice --role owner` creates the first owner (API.md).

A connection's own password shouldn't be stored as plain text - give it as a reference to an environment variable instead (in the Console's connection form, or the API), resolved from the container's environment (i.e. anything in `.env` above):

```
{"name": "prod", "db": "postgres", "host": "db.internal", "user": "app",
 "password": "${PROD_DB_PASSWORD}", "database": "app", "active": true}
```

`QUERYAPIGATE_SECRET_KEY` (a Fernet key) additionally encrypts every literal connection password *at rest* in `queryapigate.db` itself, independent of the `${VAR}` convention above - worth setting if the

volume or its backups might be read by someone who shouldn't see connection credentials. Generate one and add it to `.env` (keep a copy in your secrets manager - backups of the store are useless for these passwords without it):

```
# a Fernet key is 32 random bytes, base64url-encoded
python3 -c "import base64, os; print(base64.urlsafe_b64encode(os.urandom(32)).decode())"
```

See Keep connection passwords out of the store.

Read the full hardening checklist before going live - `QUERYAPIGATE_ALLOW_WRITES`, `QUERYAPIGATE_RATE_LIMIT`, `QUERYAPIGATE_CORS_ORIGINS` and the rest all matter for a reachable production instance and aren't repeated here.

4. First run

```
docker compose run --rm queryapigate queryapigate init   # writes a starter
queryapigate.db into the volume
docker compose up -d
curl https://api.example.com/health
```

`init` is optional - it only adds templates - and works either before the first `up` (as above) or afterwards with `docker compose exec queryapigate queryapigate init`; the running server sees the templates at once. It does nothing on a store that already has connections.

`init` writes one template connection per supported database type, all inactive - activate the ones you need from the admin UI (`https://api.example.com/console`) or `PATCH /api/v1/connections/{name}`. Do **not** run `queryapigate examples load` against a production instance; it's meant for trying the product, seeds a throwaway SQLite database and five example API keys, and is documented as such in EXAMPLES.md.

5. Persistent data, backups and restores

Everything QueryAPIGate owns - connections, saved queries and their versions, run history, API keys, roles and the audit log - lives in its store: `queryapigate.db` on the `/data` volume, or a PostgreSQL database if you set one up (section 8). Back up the store, and separately the few secrets it depends on (below). Then restore one now and then: a backup nobody has restored is not a backup.

The SQLite store

`queryapigate backup` copies it while the server keeps running. It uses SQLite's own backup API, so the copy is one consistent moment and includes changes still in the write-ahead log (`queryapigate.db-wal`) - which a plain `cp` of the file can miss:

```
docker compose exec queryapigate queryapigate backup /data/backup.db --force
docker compose cp queryapigate:/data/backup.db "./backups/queryapigate-$(date +%F).db"
docker compose exec queryapigate rm /data/backup.db
```

Run that on a schedule (host cron, or a small sidecar with its own) and ship the result off the host.

To restore, stop the server, put the backup in place as `queryapigate.db` - removing the `-wal` and `-shm` files, which belong to the store being replaced - and start it again:

```
docker compose stop queryapigate
docker compose run --rm -v ./backups:/backups queryapigate sh -c \
  'cp /backups/queryapigate-2026-10-04.db /data/queryapigate.db && rm -f
/data/queryapigate.db-wal /data/queryapigate.db-shm'
docker compose start queryapigate
```

The PostgreSQL store

Back it up with `pg_dump`, like any other PostgreSQL data. On a managed database whose snapshots or point-in-time recovery already cover the whole database, that covers QueryAPIGate's store too. To back up just the store, dump its schema - `public`, or the one `?options=-csearch_path%3D...` names in `QUERYAPIGATE_DATABASE_URL`; `queryapigate backup` prints this command with the right schema filled in:

```
pg_dump --format=custom --schema=public --no-owner --no-privileges \
  --file="queryapigate-$(date +%F).dump" "$QUERYAPIGATE_DATABASE_URL"
```

`pg_dump` reads one snapshot, in a single transaction, so it is consistent and safe to run while QueryAPIGate is serving; runs recorded after it starts are simply not in it. Use a `pg_dump` of the same major version as the server, or newer.

To restore, stop every QueryAPIGate process using the store, restore into an empty database (or one where that schema has been dropped), and start them again:

```
pg_restore --no-owner --no-privileges --dbname="$QUERYAPIGATE_DATABASE_URL"
"queryapigate-2026-10-04.dump"
```

The schema is recreated under its original name, so the URL's `search_path` (if any) must name the same schema.

What a backup of the store does not contain

- `QUERYAPIGATE_SECRET_KEY`, if you set it: connection passwords are stored encrypted with it. A store restored without the same key has every encrypted password unreadable - each connection must be given its password again. Keep the key in your secrets manager, with the `.env` it lives in. After a key rotation (`queryapigate rekey`), a backup taken before it still needs the old key: list it after the new one to restore it, then run `rekey` again.

- `QUERYAPIGATE_API_KEY`, the admin key, and any environment variable a connection refers to (`"password": "${WAREHOUSE_PASSWORD}"`). Scoped API keys *are* in the store - as hashes, so they keep working after a restore, but a lost secret can't be recovered from a backup: issue a new key instead.
- **The databases QueryAPIGate connects to** - back those up in their own right. The Redis response cache, if used, is disposable: it refills.

A restore from an older release's store is upgraded on first start, as any upgrade is; one from a newer release is refused rather than run on. `tests/test_backup_restore.py` does all of this on every CI run: it fills a store, backs it up, loses it, restores it, and checks the admin API sees exactly what it saw before.

6. Observability

`/metrics` (Prometheus text format) and the admin UI's own **Metrics** tab need no setup - see API.md's Observability section for the full metric list and what each one means. For real history, trends and alerting, add a Prometheus scrape target and import the bundled Grafana dashboard:

```
# prometheus.yml
scrape_configs:
  - job_name: queryapigate
    authorization: {credentials_file: /etc/prometheus/queryapigate-metrics-token}   #
with QUERYAPIGATE_METRICS_TOKEN
    static_configs:
      - targets: ["queryapigate:5000"]
```

Then *Dashboards → New → Import* in Grafana, uploading `documentation/grafana-dashboard.json` from this repo - panels for request/query rate and latency (p50/p95/p99), error rate, rows returned, active queries, pool occupancy and rate-limit rejections, built on exactly the metric names `/metrics` already exposes. `/metrics` needs no API key - a scraper shouldn't hold one. It's open to whoever can reach the server, until you set `QUERYAPIGATE_METRICS_TOKEN`: then only a request with `Authorization: Bearer <the token>` - and an administrator, the Console's Metrics screen - reads it. Set it whenever the server can be reached from outside your network: the labels name your API keys, queries and connections.

With several instances, scrape each one - every process keeps its own counters, as Prometheus expects - and the bundled dashboard sums across them. The Console's Metrics screen shows only the instance that served it.

7. Shared response cache (optional)

A saved query's `cache_ttl` is served from an in-process cache by default - zero setup, but wiped on every restart. If a query is hit often enough that a warm cache actually matters, point it at the `redis` service in the compose file above instead:

```
# .env
QUERYAPIGATE_REDIS_URL=redis://redis:6379/0
```

See Shared response cache for what changes (a Redis outage degrades to a cache miss, never a failed request) and browse what's actually cached from the admin UI's **Caching** tab.

8. Shared metadata store (optional)

By default connections, saved queries, keys, roles, history and the audit log live in `queryapigate.db` on the `/data` volume - see section 5 for backing it up. To keep them in PostgreSQL instead (a managed database you already back up, or a store several instances share), add to `.env`:

```
# .env
QUERYAPIGATE_DATABASE_URL=postgresql://queryapigate:
<password>@db.internal:5432/queryapigate
```

then copy the existing store across once, before switching traffic:

```
docker compose run --rm queryapigate queryapigate migrate-to-postgres
```

The image already includes the PostgreSQL driver. See Shared metadata store for what it changes and what it doesn't.

How many PostgreSQL connections. Each process holds one connection to the store per request thread, plus one for its history writer: about `threads + 1` per instance (9 with the image's 8 threads), plus one per `queryapigate events` and `queryapigate mcp` process. Three instances and an events server need about 30 - well inside PostgreSQL's default `max_connections = 100`, which the databases QueryAPIGate *queries* don't share. For many more instances, put PgBouncer in front in **session** pooling mode: the store's transactions take advisory locks, which transaction pooling would break.

Which instances are running. `GET /api/v1/instances` (admin key) lists every process using the store - role, host, version, whether it shares rate limits through Redis - seen in the last 90 seconds; each refreshes itself as it serves requests and health checks. Two **Alerts** watch it: several instances without Redis (`instances_not_shared`), and instances on different versions (`instances_versions_differ`), expected only during a rolling upgrade.

9. Live events for many clients (optional)

The main server's own `GET /events` holds a request thread per open stream, so it serves only a few (see Live events). For apps and phones, add the asyncio events server as a second service from the same image, and route `/events` to it:

```
# docker-compose.yml - next to the queryapigate service
  events:
    image: ghcr.io/anantharajuc/queryapigate:1.2.3     # the same image and version as
queryapigate
    restart: unless-stopped
    command: ["queryapigate", "events", "--host", "0.0.0.0"]
    env_file: .env                                     # the same store:
QUERYAPIGATE_DATABASE_URL, or ...
    environment:
      QUERYAPIGATE_TRUST_PROXY: "1"
    volumes:
      - queryapigate-data:/data                        # ... the same /data volume, for
SQLite
    healthcheck:                                       # the image's own check probes
port 5000
      test: ["CMD", "python", "-c", "import urllib.request;
urllib.request.urlopen('http://127.0.0.1:5002/health', timeout=3)"]
    networks:
      - internal
```

```
# Caddyfile
api.example.com {
    reverse_proxy /events* events:5002
    reverse_proxy queryapigate:5000
}
```

Caddy streams `text/event-stream` responses without buffering. With nginx, use `proxy_buffering off;` and a `proxy_read_timeout` above the 15-second keepalive for that location. The events server sees runs from every instance sharing the store - one events service is enough for several `queryapigate` replicas.

Scaling out

Two shapes are supported:

	One instance	Several instances
Store	SQLite on a volume, or PostgreSQL	**PostgreSQL** (section 8)
Redis	optional (shared cache across restarts)	**required** - rate limits and the response cache (section 7)
Live events	`GET /events` on the instance	`queryapigate events` (section 9), routed by the proxy
Availability	restart on failure	the load balancer skips an instance that fails its health check

With a PostgreSQL store and Redis, everything an instance owns is shared - connections, saved queries and versions, keys, roles, history, the audit log, rate limits, the cache - so instances are interchangeable and a load balancer can send any request to any of them. `tests/test_multinode.py` checks this on every CI run with three real servers.

The reference deployment is `deploy/scale-out/`: three instances, PostgreSQL, Redis, the events server and Caddy load-balancing them, in one Compose file. Run end to end before it was published: requests rotated across the three instances, a key's `4/minute` limit allowed exactly four calls in total across them, live events arrived through `/events`, and with one instance stopped twelve calls in a row still succeeded - Caddy retried them on the others. Walkthrough: Run several instances.

One worker per container, more containers for more capacity. The image runs gunicorn with one worker and eight threads, and that stays right when scaling out: each process keeps its own `/metrics`, and Prometheus scraping a container with several workers would reach one of them at random. For more headroom inside one container, raise `--threads` - `gthread` keeps a slow request from blocking the others - with a `command:` override:

```
    command: ["gunicorn", "--bind", "0.0.0.0:5000", "--workers", "1", "--worker-class",
"gthread",
              "--threads", "16", "--timeout", "120", "--graceful-timeout", "30", "--
access-logfile", "-",
              "--no-control-socket", "-c", "python:queryapigate.gunicorn_conf",
"queryapigate.app:create_app()"]
```

What stays per instance: `/metrics` (scrape each instance; the bundled dashboard sums them), the Console's Metrics screen, and the main server's own `GET /events` (use `queryapigate events`). `GET /api/v1/instances` lists the instances; an `instances_not_shared` alert means some of them run without Redis.

Several instances without a PostgreSQL store - each with its own SQLite volume - are separate servers with their own keys and connections, not one service. For simple redundancy without PostgreSQL, run one active instance and a standby restored from backups.

Kubernetes (Helm)

> **Stable** since 0.30, under the compatibility promise (what that means). Values may still be added; none is renamed or removed except through the deprecation process. Limits: chart X.Y.Z runs QueryAPIGate X.Y.Z; it makes no PostgreSQL or Redis of its own; and its NetworkPolicy is off until you give it your cluster's rules.

The chart is published with every release, at the server's version: chart 0.26.0 installs QueryAPIGate 0.26.0.

```
helm install queryapigate oci://ghcr.io/anantharajuc/charts/queryapigate --version 0.26.0
\
  -n queryapigate --set existingSecret=queryapigate
```

It has the two shapes above. `store.type: sqlite`, the default, runs one pod with `queryapigate.db` on a PersistentVolumeClaim. The claim is kept on `helm uninstall`. Upgrades replace that pod rather than roll it: the old pod stops before the new one starts, so expect a few seconds of downtime. `store.type: postgres` runs `replicas` pods on the PostgreSQL store with Redis. They roll one at a time, and a new pod must be ready before an old one stops (`maxUnavailable: 0`). A PodDisruptionBudget keeps all but one up during node drains.

Concern	How the chart handles it
Secrets	`existingSecret`, a Secret you manage, becomes the pods' environment: `QUERYAPIGATE_API_KEY`, `_SECRET_KEY`, `_DATABASE_URL`, `_REDIS_URL`, `_AUDIT_KEY`, ... Or `secret.create: true` with `secret.values`, which then sit in your values file.
Other settings	`config`: any `QUERYAPIGATE_*` or `OTEL_*` variable from Installation and setup.
Refusals	`helm install` fails, before anything changes, on more than one pod on SQLite, more than one without Redis, PostgreSQL without a database URL, a secret setting in `config`, or a value not in `values.schema.json`.
Routing	`ingress.*` sends `/events` to the events server (`events.enabled`) and the rest to the API pods, and sets `QUERYAPIGATE_TRUST_PROXY=1`. `mcp.enabled` runs the MCP server; `mcp.ingress.host` gives it its own host. The default annotations are ingress-nginx's no-buffering for `/events`; replace them for another controller.
Pod security	Non-root (UID 1000), a read-only root filesystem, every capability dropped, the RuntimeDefault seccomp profile, no service account token. The server writes only to `/data` and `/tmp`.
Observability	`serviceMonitor.enabled` for the Prometheus Operator (section 6), sending the token `/metrics` needs: `metricsAuth` makes one in a Secret of the chart's own, kept across upgrades - or name yours, `metricsAuth.existingSecret`, where the chart is rendered without the cluster (`helm template`, Argo CD). Tracing through `config` (`OTEL_EXPORTER_OTLP_ENDPOINT`).
Network	`networkPolicy.enabled`: in only from `networkPolicy.ingressFrom`, out only to `networkPolicy.egress` and DNS. List your databases, Redis and the SIEM there.
Scale	`autoscaling.enabled`, CPU-based, PostgreSQL shape only.
Secrets managers	`${vault:...}`, `${aws-sm:...}` and short-lived database credentials (Secrets from files and secrets managers; AWS's experimental): `vaultToken.enabled` for Vault's Kubernetes auth, `serviceAccount.annotations` for IRSA, and the `QUERYAPIGATE_VAULT_*` / `QUERYAPIGATE_AWS_*` settings through `config`.
Configuration as code	`configAsCode.enabled` with `configAsCode.file` (or `existingConfigMap`): connections, queries, roles, keys and the rest as one file in git, applied by the first new API server pod as it starts, once per file (Configuration as code, experimental). A change to it rolls the pods; a file that can't be applied stops the rollout at its first pod. `locked: true` keeps what it manages to the file.

Two things to know:

- **Changing a Secret you manage doesn't restart the pods**, since the chart can't see inside it. Run `kubectl rollout restart` on each Deployment. Changes to `config`, and to a Secret the chart makes, roll the pods on `helm upgrade`.
- **Backups** are the same as above. Run `kubectl exec deploy/queryapigate -- queryapigate backup /tmp/backup.db`, then `kubectl cp` the file out. For PostgreSQL, use your database's own backups.

Every value: the chart's README. Walkthrough, run on kind: Install on Kubernetes with Helm. Plain manifests of the PostgreSQL shape, kept in step with the chart by a test: `deploy/kubernetes/`.

Upgrading

1. Read the Changelog entry for the target version, especially anything under a "Breaking" or "Upgrading" heading.
2. Back up first - always, even for a patch version.
3. Bump the pinned tag in `docker-compose.yml`, then `docker compose pull && docker compose up -d`.
4. Check `docker compose logs queryapigate` and `curl .../health`.

The `queryapigate.db` schema is migrated automatically on the container's first start against it - nothing to run by hand. Every release since 0.7 is tested this way: a store each one built through its own API is started under the current code, which must read back its connections, queries and their versions, run history, keys and roles (`tests/test_upgrades.py`). A store this version can't safely run on - one with a column missing, or one a *newer* release has already upgraded - stops startup with a message naming the problem and what to do, before anything is changed; it never starts and then fails on first use. Going back to an older release is not supported in place: restore the backup from step 2.

Rolling upgrades (several instances on one store)

Replace instances one at a time - no downtime - because of this rule: **within 1.x, a release changes the store only additively** (new tables, new nullable columns, new indexes; never a rename, a drop or a changed meaning). So while some instances run the new release and have upgraded the store, the ones still on the previous release keep working against it. `tests/test_rolling_upgrade.py` checks exactly that on every CI run: the previous release from PyPI keeps serving, writing and recording runs on a store the current code has just upgraded, and each sees the other's changes.

1. Back up the store.
2. Start one instance on the new version - it upgrades the store on its first start - and check its `/health`.
3. Replace the others one by one. `GET /api/v1/instances` shows who is on which version meanwhile.
4. Replace `queryapigate events`, if you run it, with the instances. From 0.33 it streams the event log (ADR 0020), and an instance still on 0.32 writes its runs to history only - so they aren't events until it's upgraded too.

Two limits:

- **An instance can't *start* on the previous release once the store is upgraded** - it refuses, naming the newer release. If one restarts mid-upgrade, bring it back on the new version.
- **Upgrade one minor version at a time** (0.15 → 0.16, not 0.15 → 0.17) when rolling: the guarantee is about the release just before. A release that can't keep it says so under **Breaking** in its changelog entry, and then all instances must be stopped before the first one starts on it. If you're upgrading from the project's old name (SQL2API), see Upgrading from SQL2API first; environment variables renamed and the server refuses to start until they're renamed too.

Software bill of materials and release verification

This pack describes **QueryAPIGate 0.34.2**. The SBOM is not copied into the pack: it is attached to each released image, signed, and you should read it from the exact image you run, by digest. This page says where it is and how to check that it, and the image, came from the project's release workflow.

What is published with every release (since 0.31)

Artefact	Form
Container image, `linux/amd64` and `linux/arm64`	Signed keyless with cosign, by digest, in both the Sigstore bundle and the classic `.sig` form
SBOM, per platform	SPDX 2.3, attached to the image as an in-toto attestation
Build provenance, per platform	SLSA provenance v1, attached to the image
GitHub attestation	Names the repository, workflow, commit and run
Helm chart	Signed the same way, published at the server's version

Releases before 0.31 are not signed. Background: ADR 0018: Signed releases.

0.34.2 at a glance

	Reference
Image	`ghcr.io/anantharajuc/queryapigate:0.34.2`
Image index digest	`sha256:25511ebc7b89e1dfdbe6a55e4a9ef3e10d9049e82aefba8cba72b4c0884ff1b2`
`linux/amd64` image	`sha256:8237acc69bc9ebe7f2346cb72837d8383a9ffb374ab83e9582b17a30b7d4754a`
`linux/arm64` image	`sha256:2c6d5a0288cde1ffbebae0983b46c6288c2f4509e8146318f183b67199c42c8b`
Helm chart	`oci://ghcr.io/anantharajuc/charts/queryapigate` version `0.34.2`, digest `sha256:fd233bbf9d36a9973828c0d9515c10d05f31c91208b3a1f6c34b0545a8d29385`
Signing identity	`https://github.com/AnanthaRajuC/QueryAPIGate/.github/workflows/docker.yml@refs/tags/v0.34.2`
OIDC issuer	`https://token.actions.githubusercontent.com`

The digests were read from the registry when this pack was built. Pull by digest, and verify before you run.

Read the SBOM

```
docker buildx imagetools inspect ghcr.io/anantharajuc/queryapigate:0.34.2 --format
'{{json .SBOM}}'
```

One SPDX 2.3 document per platform. Feed it to your scanner (Grype, Trivy) to see which advisories apply, without pulling the image.

Verify the signature

```
cosign verify
ghcr.io/anantharajuc/queryapigate@sha256:25511ebc7b89e1dfdbe6a55e4a9ef3e10d9049e82aefba8c
ba72b4c0884ff1b2 \
  --certificate-identity
https://github.com/AnanthaRajuC/QueryAPIGate/.github/workflows/docker.yml@refs/tags/v0.34
.2 \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com
```

Keyless means there is no key to trust or rotate: the signing certificate names the workflow and the tag it ran on, and the signature is recorded in Sigstore's public transparency log.

Verify how it was built

```
gh attestation verify oci://ghcr.io/anantharajuc/queryapigate:0.34.2 --owner AnanthaRajuC
```

or, with cosign, `cosign verify-attestation --new-bundle-format --type https://slsa.dev/provenance/v1` with the same identity and issuer as above. The provenance names the repository, the workflow, the commit and the run; the image's own build provenance records the base images by digest.

Admit only signed releases to a cluster

Sigstore's policy-controller and Kyverno can refuse any image this workflow did not sign on a release tag. The policies, and the outputs they give for signed, unsigned and wrongly signed images, are in Verify a release before you run it.

How builds are pinned

Every GitHub Action in the release workflow is pinned by commit SHA and the base images by digest; a test fails on anything unpinned (0.31.0).