### Status

I’ll generate a single, exhaustive Markdown “Project Reference & Architecture Guide” for your SQL Query Explanation & Optimization Engine (QEO), following your format. It’s tailored to your current stack and includes HypoPG what-if, metrics, CI, and runbooks. You can paste it into a `docs/ARCHITECTURE.md`.

---

# SQL Query Explanation & Optimization Engine (QEO): Project Reference & Architecture Guide

Version: 0.7.0

Last Updated (UTC): 2025-08-13

Document Owner(s): TODO: Assign maintainer(s)

Quick Summary

- QEO is a local, read-only FastAPI service and CLI that explains, evaluates, and optimizes SQL queries deterministically. It integrates with PostgreSQL, leverages EXPLAIN (FORMAT JSON), optional Ollama for NL explanations, a deterministic optimizer, and an optional HypoPG “what-if” cost-based ranking to validate index candidates by estimated plan cost deltas. Observability via optional Prometheus, and a small benchmark micro-suite.

How to use this doc

- Start at Executive Summary and Requirements to understand goals and constraints.

- Use System Context & Architecture to see how components interact; follow the Runbooks in Local Development & Onboarding.

- Reference API & Integration Contracts for endpoint details and inputs/outputs; CI/CD & DevEx for pipelines.

<details>

<summary>Table of Contents</summary>

- 1. Title Page

- 2. Executive Summary

- 3. Requirements & Traceability

- 4. System Context & Architecture

- 5. Domain & Data Modeling

- 6. API & Integration Contracts

- 7. Application Design

- 8. Security & Privacy

- 9. Reliability & Performance

- 10. Observability

- 11. CI/CD & DevEx

- 12. Infrastructure & Operations

- 13. Testing Strategy & Quality

- 14. Local Development & Onboarding

- 15. Deployment & Release Management

- 16. Operations, Support & Maintenance

- 17. Decisions & History

- 18. Roadmap & Open Questions

- 19. Appendices

</details>

## 2. Executive Summary

Problem Statement

- Engineers and DBAs need local, safe tools to explain, measure, and optimize SQL queries. Existing approaches can be non-deterministic, require cloud LLMs, or risk unintended schema changes.

Objectives & Success Metrics

- Deterministic outputs for identical inputs (ordering, rounding).

- Read-only operation: no DDL/DML outside optional HypoPG hypothetical indexes.

- Explain: retrieve canonical plan JSON, warnings, basic metrics.

- Optimize: provide deterministic rewrites and index recommendations.

- What-if: optionally validate top index candidates via HypoPG with cost deltas.

- Metrics opt-in; bench micro-suite outputs reproducible JSON/CSV.

In/Out of Scope

- In: SELECT optimization, EXPLAIN analysis, NL explanations (local optional), cost-based what-if.

- Out: Executing DDL changes, non-Postgres engines, cross-db migrations.

Assumptions & Constraints

- Postgres is reachable locally (Docker port 5433).

- No PII in metrics labels; low label cardinality.

- Statement timeouts enforced.

Risks (Top 5)

- HypoPG not installed: impact limited (fallback to heuristic). Mitigation: clear flags and README.

- Ollama slow/unavailable: NL soft-fail, dummy fallback. Mitigation: gating, timeouts, retries.

- Environmental drift (ports/env): Mitigation: `.env` and README runbooks.

- Determinism regressions: Mitigation: unit tests for ordering/rounding.

- Plugin/tooling in tests slowing runs: Mitigation: `PYTEST\_DISABLE\_PLUGIN\_AUTOLOAD=1`.

## 3. Requirements & Traceability

Functional Requirements

- FR-1: Explain SQL plans with bounded timeout; return canonical `{"Plan": {...}}` plus warnings/metrics. Acceptance: 200 OK; plan shape normalized.

- FR-2: NL explanations optional; failure must not break endpoint. Acceptance: `ok=true`, message notes failure.

- FR-3: Optimizer provides deterministic rewrites and index suggestions with stable ordering/rounding. Acceptance: 5 repeated calls → identical JSON.

- FR-4: Optional HypoPG what-if: compute cost deltas for top N index candidates, filter by min reduction pct, enrich suggestions. Acceptance: returns `ranking`, `whatIf`, and `estCost\*` fields.

- FR-5: CLI supports lint/explain/optimize with `--what-if` toggles and limits.

Non-Functional Requirements

- NFR-1 Performance: P50 endpoint latency < 200ms excluding DB/LLM; DB/LLM paths have timeouts.

- NFR-2 Reliability: Soft-fail on EXPLAIN/LLM/what-if; return helpful messages, 200 when safe.

- NFR-3 Security: No raw SQL/PII in metrics labels; CORS configurable.

- NFR-4 Determinism: Stable ordering, 3-decimal rounding for float outputs.

Compliance & Regulatory

- No regulated data processing. If used with PII, ensure SQL text is not emitted in metrics/log labels.

Traceability Matrix (excerpt)

- FR-1 → `src/app/core/db.py:run\_explain()`; Tests: `tests/test\_explain\_endpoint.py`.

- FR-3 → `src/app/core/optimizer.py`; Tests: `tests/test\_optimizer\_unit.py`.

- FR-4 → `src/app/core/whatif.py`; API `src/app/routers/optimize.py`; CLI `src/app/cli.py`.

## 4. System Context & Architecture

Context Diagram

```mermaid

flowchart LR

Client[Dev/DBA\nUser] -->|HTTP/JSON| API[FastAPI Service]

API -->|psycopg2| Postgres[(PostgreSQL)]

API -->|optional HTTP| Ollama[Ollama Local LLM]

API -->|/metrics| Prometheus[Prometheus (optional)]

```

C4 Container Diagram

```mermaid

graph TB

subgraph Host

API[FastAPI (QEO)]:::svc

CLI[CLI (qeo)]:::svc

end

DB[(PostgreSQL)]:::db

LLM[Ollama]:::ext

Prom[Prometheus]:::ext

API -- psycopg2 --> DB

API -- HTTP --> LLM

API -- Exposition --> Prom

CLI -- psycopg2 --> DB

classDef svc fill:#eef,stroke:#66c,stroke-width:1px

classDef db fill:#efe,stroke:#6c6,stroke-width:1px

classDef ext fill:#ffe,stroke:#cc6,stroke-width:1px

```

Key Sequence: Optimize with What-if

```mermaid

sequenceDiagram

participant C as Client

participant A as API /optimize

participant PG as Postgres

Note over A: parse → optional EXPLAIN → schema/stats → optimizer

C->>A: POST /api/v1/optimize (WHATIF\_ENABLED?)

A->>PG: EXPLAIN (FORMAT JSON) [timeout]

A->>PG: Catalog reads (schema/stats)

A->>A: Optimizer: rewrites + index suggestions (deterministic)

alt WHATIF\_ENABLED & HypoPG available

A->>PG: SELECT hypopg\_reset()

loop For each candidate (<= max trials)

A->>PG: hypopg\_create\_index(...) (hypothetical)

A->>PG: EXPLAIN (FORMAT JSON, COSTS ON, TIMING OFF)

A->>PG: SELECT hypopg\_reset()

end

A->>A: Rank by cost delta desc (then tie-breakers)

else

A->>A: Use heuristic ranking

end

A-->>C: JSON with suggestions, summary, ranking, whatIf

```

## 5. Domain & Data Modeling

Glossary

- HypoPG: PostgreSQL extension for hypothetical indexes (planner-only).

- What-if: Evaluating candidate index shapes by cost deltas without creating real indexes.

Database Schema

- App is read-only. Schema inspection via `information\_schema` and `pg\_catalog`.

- Indexing strategy: Advisor proposes composite indexes: equality → range → order/group; de-dup with existing left-prefix matches.

Data Contracts

- Plan JSON normalized to contain top-level `Plan`.

Caching Strategy

- Optional in-memory LRU-like cache for NL explanation prompts in dev.

## 6. API & Integration Contracts

Endpoints

- GET `/health`

- POST `/api/v1/lint` → Lints SQL (static)

- POST `/api/v1/explain`

- Request: `{ sql: string, analyze?: bool, timeout\_ms?: int, nl?: bool, audience?: enum, style?: enum, length?: enum, plan?: object }`

- Response: `{ ok, plan, warnings, metrics, explanation?, explain\_provider?, message }`

- GET `/api/v1/schema`

- POST `/api/v1/optimize`

- Request: `{ sql, analyze?: bool, timeout\_ms?: int, advisors?: ["rewrite","index"], top\_k?: int }`

- Response adds:

- `ranking: "cost\_based"|"heuristic"`

- `whatIf: { enabled, available, trials, filteredByPct }`

- Each index suggestion may include `estCostBefore`, `estCostAfter`, `estCostDelta` (3 decimals).

- Existing fields preserved: `suggestions`, `summary`, `plan\_warnings`, `plan\_metrics`, `advisorsRan`, `dataSources`, `actualTopK`, `message`.

- GET `/metrics` (only if `METRICS\_ENABLED=true`)

Error Handling

- Soft-fail strategy returns 200 OK with helpful message when EXPLAIN/LLM/what-if fails.

## 7. Application Design

Module Boundaries

- `app/core/db.py`: connections, EXPLAIN helpers, schema/stats, now `run\_explain\_costs`.

- `app/core/sql\_analyzer.py`: parsing and lint rule extraction.

- `app/core/plan\_heuristics.py`: warnings/metrics from plan tree.

- `app/core/optimizer.py`: deterministic rewrites and index advisor.

- `app/core/whatif.py`: HypoPG evaluator (cost-based ranking).

- `app/core/llm\_adapter.py` + providers: LLM abstraction and selection/fallback.

- `app/core/metrics.py`: Prometheus collectors (optional).

- Routers: `explain.py`, `optimize.py`, `schema.py`, `lint.py`.

- `app/cli.py`: lint/explain/optimize commands.

Patterns & Rationale

- Pure functions, deterministic outputs (sorting, rounding).

- Soft-fail and bounded timeouts for safety.

- Optional integrations (Ollama, Prometheus, HypoPG) gated via env.

Configuration System

- Source: env + `.env` via `python-dotenv`. Precedence: OS env > .env defaults.

Feature Flags

- `METRICS\_ENABLED`, `LLM\_PROVIDER`, `WHATIF\_ENABLED`, `RUN\_DB\_TESTS`, `RUN\_OLLAMA\_TESTS`.

## 8. Security & Privacy

Threat Model (STRIDE summary)

- Spoofing/ Tampering: connections over local Docker; no credentials stored in code.

- Information disclosure: no SQL/PII in metrics labels.

- Denial of service: statement timeouts; retry limits.

Secrets Management

- DB creds via env; no secrets in repo.

AuthN/AuthZ

- Local development tool; no auth on endpoints by default. TODO for multi-user deployments.

Audit Logging

- Structured request logs; no raw SQL in labels.

## 9. Reliability & Performance

SLOs/SLIs

- Examples: /optimize P95 800ms (excluding what-if); what-if trials each bounded by timeout.

Resiliency

- Timeouts on all DB operations; LLM retries; what-if reset between trials.

Disaster Recovery

- Not applicable for local tool; users store test DB locally.

## 10. Observability

Logging

- Structured via middleware with request id and duration.

Metrics (Prometheus)

- Requests: `qeo\_requests\_total{route,method,status}`

- Latency: `qeo\_request\_latency\_seconds\_bucket{route,method,status}`

- DB: `qeo\_db\_explain\_seconds`, `qeo\_db\_errors\_total`

- LLM: `qeo\_llm\_latency\_seconds`

- What-if: `qeo\_whatif\_trials\_total`, `qeo\_whatif\_trial\_seconds`, `qeo\_whatif\_filtered\_total`

Dashboards

- HTTP rate/latency by route; DB explain time; what-if trial rates.

Alerts

- High error rate; timeouts spike; what-if trial errors (optional).

## 11. CI/CD & DevEx

Branching

- main with PRs; unit tests required; integration tests gated by flags.

CI (GitHub Actions)

- Lint, unit tests always; DB/Ollama/bench workflows via env gates.

- Release workflow builds Docker and publishes package; manual bench job uploads artifacts.

Pre-commit Hooks

- TODO: add `black`, `isort`, `ruff` as optional.

## 12. Infrastructure & Operations

Target Platforms

- Local dev: Docker Compose for Postgres; API run on host or Dockerized.

Networking

- Postgres mapped host 5433 → container 5432 to avoid conflicts.

Runbooks

- DB up: `docker compose up -d db`

- API: `python -m uvicorn app.main:app --reload`

- Enable metrics: `METRICS\_ENABLED=true`

- HypoPG quick install (container): `apt-get install postgresql-16-hypopg` then `CREATE EXTENSION hypopg;`

## 13. Testing Strategy & Quality

Unit Tests

- Determinism: 5x identical outputs.

- Optimizer de-dup; small-table suppression; top-K; rounding.

- LLM dummy provider and explain NL mode.

Integration Tests (gated by RUN\_DB\_TESTS=1)

- /explain, /schema, /optimize end-to-end.

- Bench micro-suite report existence.

Benchmark

- `scripts/bench/run\_bench.py` creates ephemeral schema and records timings to `bench/report/`.

## 14. Local Development & Onboarding

Prereqs

- Python 3.11+, Docker Desktop, PowerShell (Windows) or bash.

Setup

```bash

python -m venv .venv

.\.venv\Scripts\Activate.ps1 # Windows

pip install -r requirements.txt

docker compose up -d db

$env:PYTHONPATH="src"; python -m uvicorn app.main:app --reload

```

Env Vars (excerpt)

- DB\_URL, HOST\_PG\_PORT=5433

- LLM\_PROVIDER=dummy|ollama

- WHATIF\_ENABLED, WHATIF\_MAX\_TRIALS, WHATIF\_MIN\_COST\_REDUCTION\_PCT

- METRICS\_ENABLED, METRICS\_NAMESPACE

Troubleshooting

- 404 on endpoints → router prefix; check `/docs`.

- DB auth error → ensure port 5433; correct `.env` encoding (UTF-8).

- Ollama timeouts → use dummy; simplify prompt; increase timeout.

- HypoPG missing → install package and `CREATE EXTENSION`.

## 15. Deployment & Release Management

Versioning

- `src/app/\_\_init\_\_.py` → `\_\_version\_\_ = "0.7.0"`

Change Log

- `CHANGELOG.md` maintained; v0.7.0 adds HypoPG what-if.

Rollback

- Disable what-if with `WHATIF\_ENABLED=false` (no code changes).

## 16. Operations, Support & Maintenance

Ops Tasks

- None persistent; local-only tool. Keep dependencies updated; monitor CI.

Incident Response

- For local failures: check logs, verify DB connectivity, disable optional integrations.

Known Issues

- Ollama performance varies by model; prompt simplified for speed.

- Windows CRLF warnings (harmless).

## 17. Decisions & History

ADRs (summarized)

- ADR-001: Deterministic outputs are mandatory; round floats to 3 decimals.

- ADR-002: Read-only by default; no DDL executed. HypoPG allowed for hypothetical ops only.

- ADR-003: Metrics opt-in; no SQL/PII in labels.

- ADR-004: LLM provider fallback to dummy; Ollama gated by `RUN\_OLLAMA\_TESTS`.

Milestones

- v0.6.0: Observability, OpenAPI examples, bench suite.

- v0.7.0: HypoPG what-if evaluator, API/CLI wiring, metrics.

## 18. Roadmap & Open Questions

Short-term

- Add text-mode compact table output for CLI cost deltas.

- Expand integration tests for what-if (skip if hypopg missing).

Mid-term

- Additional rewrite heuristics; more robust schema-aware projections.

Long-term

- Optional tracing; richer advisor plugins.

Open Questions / TODOs

- Document Owner(s) assignment.

- Auth for multi-user deployments.

- Pre-commit hooks adoption and coding standards.

## 19. Appendices

Code Snippets

- What-if evaluator entry: `src/app/core/whatif.py`

- Cost EXPLAIN: `run\_explain\_costs()` in `src/app/core/db.py`

cURL Examples

```bash

curl -s -X POST http://127.0.0.1:8000/api/v1/optimize \

-H "Content-Type: application/json" \

-d '{"sql":"SELECT \* FROM orders WHERE user\_id=42 ORDER BY created\_at DESC LIMIT 50","analyze":false,"timeout\_ms":8000}'

```

Example k6 Script

```js

// TODO: Add k6 script for /optimize with different payload sizes

```

License & Attributions

- License: MIT (see `LICENSE`)

- Third-party: FastAPI, psycopg2, sqlglot, prometheus\_client, requests, uvicorn, HypoPG.

---

Summary

- Delivered a comprehensive, navigable guide covering goals, architecture, APIs, optimizer and what-if design, observability, CI, testing, and runbooks, tailored to your current repository and recent HypoPG integration.

- What-if (HypoPG) path is live end-to-end.

- HypoPG installed and enabled in the DB.

- API started with WHATIF flags; DB pointed to port 5433.

- Optimizer produced index candidates; HypoPG trials executed.

- API/CLI output now shows cost-based ranking.

- ranking: cost\_based

- whatIf: enabled=true, available=true, trials=1, filteredByPct=0

- Suggestions enriched with estCostBefore/After/Delta (3 decimals).

- Deterministic fields (ordering, rounding, advisorsRan, dataSources, actualTopK) preserved.

- Stability fixes applied.

- Indentation bug in db connection cleanup fixed.

- Lowered OPT\_MIN\_ROWS\_FOR\_INDEX to 1000 (env) so index suggestions appear and trials run.

- Verified via smoke test.

- POST /api/v1/optimize returned cost-based ordering and metrics as designed.

Status update:

Added a durable HypoPG setup and seed data path, updated config defaults, bumped API version, and refreshed README with env, what-if, metrics, and seeding instructions. Built and started the DB with HypoPG, verified extension, and seeded 50k rows into orders.

What I did:

docker-compose: switched DB service to build docker/db.Dockerfile, added infra/init volume.

Added docker/db.Dockerfile to install postgresql-16-hypopg.

Added infra/init/10-enable-hypopg.sql to auto-create HypoPG.

Added infra/seed\_orders.sql and seeded 50k rows (matched existing schema).

src/app/core/config.py: default DB\_URL now port 5433.

src/app/main.py: bumped FastAPI version/meta to 0.7.0.

README.md: added final env defaults, HypoPG what-if section, metrics runbook, and seeding commands.

Built and started DB, verified HypoPG via psql.

Committed changes.

Next actions for you:

Create .env from the new example (the tool couldn’t write .env directly): copy .env.example contents below into your .env or set via environment:

DB\_URL=postgresql+psycopg2://postgres:password@localhost:5433/queryexpnopt

LLM\_PROVIDER=dummy

OPT\_MIN\_ROWS\_FOR\_INDEX=10000

WHATIF\_ENABLED=true

WHATIF\_MAX\_TRIALS=8

WHATIF\_MIN\_COST\_REDUCTION\_PCT=5

METRICS\_ENABLED=false

Restart API and try CLI with what-if:

PowerShell:

$env:PYTHONPATH="src"

python -m app.cli optimize --what-if --sql "SELECT \* FROM orders WHERE user\_id=42 ORDER BY created\_at DESC LIMIT 50"

Optional metrics check:

Set METRICS\_ENABLED=true; restart API; GET http://127.0.0.1:8000/metrics; grep for qeo\_whatif.

Summary

Persisted final config defaults (README and config), baked HypoPG into the DB image, added auto-init and seeding, bumped API version to 0.7.0, and documented CLI/metrics. You only need to place the .env file locally (tool could not create it due to ignore) and restart the API.  
  
  
**Where the project stands**

* End-to-end is working locally:
* Postgres 16 with HypoPG installed and enabled.
* Seeded orders (~100k rows in your run).
* API up with working routes: GET /health, POST /api/v1/optimize, POST /api/v1/explain, POST /api/v1/lint, GET /metrics (when metrics enabled).
* Cost-based what‑if is active: ranking="cost\_based", whatIf.enabled=true, available=true, trials>0, estCost\* fields present.
* Metrics are wired and incrementing: qeo\_requests\_\*, qeo\_request\_latency\_seconds, qeo\_whatif\_\* confirmed; LLM metrics move on nl=true.

**What we’ve accomplished**

* Infra:
* Durable HypoPG via docker/db.Dockerfile + infra/init/10-enable-hypopg.sql.
* Local DB on host port 5433; seeding via infra/seed\_orders.sql.
* App:
* Deterministic optimizer path with index suggestions + rewrite advisor.
* What‑if evaluator (src/app/core/whatif.py) with HypoPG cost deltas and ranking.
* Metrics plumbing (src/app/core/metrics.py) with low‑cardinality labels.
* CLI (app.cli) supports lint/explain/optimize with --what-if.
* Fixed db.py indentation bug; config defaults updated; version set to 0.7.0.
* README updated with env defaults and runbooks.
* Validation:
* Health OK, optimize OK with cost-based ranking, metrics OK and incrementing.

**Use cases we currently address**

* Explain plans deterministically with warnings/metrics; soft-fail on DB issues.
* Deterministic optimization: rewrites + index suggestions with stable ordering.
* Cost-based validation of index candidates via HypoPG, including filtering by minimum reduction percentage and re-ranking.
* CLI-based analysis without running the API.
* Basic observability with Prometheus-compatible metrics.

**How close to production**

* Good for local/dev and internal demos. Not production-ready yet. Key gaps:
* No auth/z on API; CORS is permissive.
* API not containerized in compose (DB only is containerized); no liveness/readiness probes for the API.
* No rate limiting, no circuit-breakers, no request timeouts at ASGI layer.
* Limited integration tests for what‑if; no load tests or soak tests.
* Config/secrets not managed for production (env only; no secrets manager).
* Monitoring/dashboards/alerts not defined; metrics exist but no dashboards.
* No structured error taxonomy or SLOs; logs unshipped.
* Security review not done; no SBOM/dependency scanning.
* No CD pipeline; CI coverage not shown here.

**What progressed in your latest run**

* Verified cost-based what‑if works against real seeded data (trials and latencies recorded).
* Metrics verified across optimize and explain paths.
* Stress runs confirmed stability under repeated optimize calls.
* Startup/port conflicts resolved; metrics exposed correctly when METRICS\_ENABLED=true.

**Recommended next steps (production track, in order)**

1) Containerize API and compose it with DB

* Add api service to docker-compose.yml with envs and dependency on db.
* Add healthcheck; expose 8000; run uvicorn with --proxy-headers.

2) Basic security hardening

* Add API key or token auth (header) and disable \* CORS by default.
* Hide /docs behind auth in non-dev.

3) Operational readiness

* Add readiness/liveness endpoints or reuse health appropriately.
* Create Prometheus/Grafana dashboard JSONs for qeo\_\* metrics.
* Define simple alerts (high error rate, slow latencies).

4) CI improvements

* Ensure pytest -q green in CI; add flags to skip DB/LLM unless enabled.
* Add a container build job and image scanning.

5) Resilience

* Configure uvicorn workers, set ASGI timeouts and max connections.
* Add rate limiting and request size limits.

6) Documentation

* Add docs/ARCHITECTURE.md (use the outline you pasted) and env.example (don’t commit .env).

7) Expanded testing

* Add integration tests for what‑if (skipped when HypoPG unavailable).
* Add performance test script (k6 or Locust) for /optimize.

If you want, I can turn your pasted guide into docs/ARCHITECTURE.md and add env.example.

* You’re functionally complete for dev: explain, optimize, what‑if, metrics, CLI.
* To reach a production-ready pilot, focus on API containerization, auth, basic ops, CI, and guardrails.

23%

queryexpnopt