Querygraph

covers querygraph (0.4.2-9ef26f3)

A Governed Semantic Lakehouse for AI Agents

Alexy Khrabrov and Slava Tykhonov

querygraph.ai

Contents

Preface ... 3
The Vision from QueryGraph.ai ... 3
- Compression, Not Noise ... 6
- Freezing Time Without Freezing the World ... 6
- The Data Infrastructure for Agentic AI ... 6
- Graph and Vector Navigation ... 6
- Digital Public Infrastructure ... 7
- Ontology-Driven Precise AI ... 7
- Local-First, OSS AI ... 7
- Source Posts ... 8

The AI Navigator Thesis ... 8
- The Alternative to Big AI ... 11
- Focused Context as a First-Class Artifact ... 11
- CPU-Bound Before GPU-Bound ... 12

The Querygraph Spine ... 12
Semantic Croissant ... 14
CDIF ... 15
DID ... 16
ODRL ... 16
TypeSec ... 19
Grust ... 20
OSI ... 21
OpenLineage ... 24
Sail ... 25
Python Interop ... 26
- Pydantic ... 27
- LangChain ... 28
- PySpark ... 28

The Demonstration Datasets ... 28
QG Lakehouse with All Components ... 33
- Step 1: Load the Lakehouse ... 33
- Step 2: Materialize Semantic Croissant ... 33
- Step 3: Project CDIF ... 34
- Step 4: Build the OSI Semantic Model ... 34
- Step 5: Load the Grust Graph ... 34
- Step 6: Identify Agents with DIDs ... 35
- Step 7: Apply ODRL Rights ... 35
- Step 8: Mint TypeSec Capabilities ... 35
- Step 9: Route to Compartmentalized Agents ... 36
- Step 10: Call Ollama Through TypeDID ... 36
- Step 11: Synthesize Without Boundary Collapse ... 36
- Step 12: Emit OpenLineage to Sail ... 37
- Step 13: Anchor the DID Attestation ... 37

LakeCat as the QueryGraph Catalog Boundary ... 39
Rust Examples ... 40
- Build a Four-Layer Semantic Bundle ... 40
- Load and Verify the Sail Warehouse ... 41

Run the Supervised Agent Story 41
Run the Live Sail, TypeDID, OpenLineage, and Ollama Path 41
Rust as the Contract 42
Python Examples 42
Build OSI from Semantic Croissant 42
Query Sail with PySpark 42
Run TypeDID Agents with Pydantic 43
Adapt a TypeDID Agent to LangChain 44
Python as the Experiment Surface 44
Operator Workflow 44
Why Querygraph Is an Alternative to Big AI 45
The Goshawk Interoperability Release 46
Real Signatures, Verified Across Languages 46
A Service Surface: the /v1 API 46
MCP: One Server, Every Framework 46
The Agent Card 47
Conformance, Proven Rather than Asserted 47
The Governed Navigator Loop 47
The Refactored QueryGraph Stack 47
Marciana Separated: Memory as a Governed Product 48
MARCIANA-ADVERSARIAL-v1 49
AGENTGYM: Authority Under Adversarial Composition 51
What This Result Does Not Prove 53
Three Language Surfaces 53
Where Querygraph Goes Next 54

Preface

Querygraph is an AI Navigator over governed enterprise data. It starts from a simple disagreement with the dominant direction of AI infrastructure: serious knowledge work should not require throwing enormous, global, poorly scoped contexts at expensive GPU systems whenever a user asks a precise question. Most enterprise and scientific questions are local, contextual, permissioned, and reproducible. They deserve an architecture built for that reality.

Querygraph treats a lakehouse as more than tables: each dataset is described by Semantic Croissant, projected through CDIF, governed by RBAC and ODRL, addressed by DIDs, and audited through OpenLineage. The result is a focused retrieval and reasoning environment where agents operate inside precise semantic contexts rather than wandering through giant prompt buffers.

The implementation in this repository is intentionally practical. It can load Dataverse and CODATA data into Sail, materialize typed tables, generate Croissant and CDIF sidecars, wrap agent requests with TypeSec TypeDID envelopes, and emit OpenLineage events back into Sail. The point of this book is to make that architecture legible as a textbook: first the motivation, then the pieces, then the full working system.

The system rests on three coordinated, named open-source releases that this book tracks throughout: Grust 0.12.1 “Lobster” for the graph and query substrate, TypeSec 0.13.1 “Torcello” for the typed security fabric, and LakeCat 0.3.0 “Ocelot” for the catalog boundary. Where a chapter leans on a specific capability, it names the release that brought it — many foundations arrived with the previous line (Grust “Crab”, TypeSec “Burano”, LakeCat “Lynx”), and the chapters keep that history.

The Vision from QueryGraph.ai

The public QueryGraph.ai posts describe a larger ambition than a metadata library. They describe an AI Navigator: a system that lets agents move through data with the same precision that a navigation system gives to physical travel. The navigator does not merely retrieve documents. It finds the right semantic object, checks whether the agent is allowed to use it, records the provenance of every step, and makes the resulting answer reproducible.

The motivation begins with a critique of contemporary LLM systems. They are fluent but unstable. The same prompt can drift across time, model version, provider, and context window. For casual chat this may be charming. For science, policy, medicine, finance, infrastructure, and enterprise operations, it is a control failure. A reliable AI system needs context, provenance, and governance before it needs more tokens.

The deeper critique is computational. Big AI often treats ambiguity by scaling up: bigger models, larger context windows, more embeddings, more GPU cycles, more remote inference, more global search. Querygraph takes the opposite bet: the best way to make AI useful is to reduce the problem before inference. Resolve the ontology. Find the exact dataset. Select the permitted fields. Reuse cached semantic descriptions. Retrieve the smallest responsible context. Then call a local or governed model only when the model is actually needed.

The posts make several claims that become design principles in Querygraph:

- Data must be understandable by both humans and agents.
- Ontologies must carry domain meaning, units, hierarchies, and controlled vocabularies.
- AI answers should be reproducible enough to inspect and compare over time.
- Prompts, skills, variables, datasets, and agents need identities.
- Governance must be machine-actionable and attached to the data itself.

- Graphs and vectors should work together: the graph provides identity, provenance, and relationships; vector search provides similarity and discovery.
- A responsible AI platform starts at the data layer, not at the last safety prompt before inference.

In this vision, an AI Navigator is not a chatbot. It is the semantic operating layer for agentic AI.

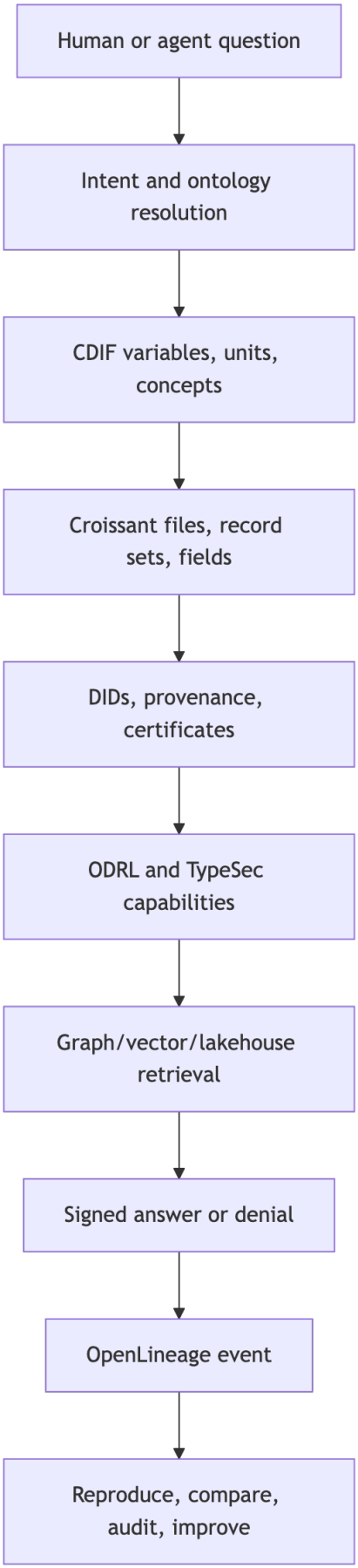

Figure 1: Diagram 1

Compression, Not Noise

One QueryGraph.ai theme is compression: a system understands something only when it can reduce noise to a portable, structured signal. Querygraph applies that idea to enterprise data. A raw table with 800 columns is not yet knowledge. A table described by Croissant, grounded in CDIF variables, linked to ontology terms, governed by ODRL, and indexed in a graph has been compressed into a form an agent can safely use.

Example: an energy survey may contain a column that looks like a number. The compressed semantic signal says what the number means, what unit it uses, which geography it belongs to, which survey instrument produced it, whether a given agent may summarize it, and what redaction rule applies before sharing.

Freezing Time Without Freezing the World

Another theme is vector stabilization and temporal ground truth. Querygraph does not claim that all knowledge is timeless. It assumes the opposite: knowledge changes, models drift, policies change, and experts disagree. The task is to record the time, source, authority, model, prompt, and data state behind an answer so later agents can compare one answer to another.

Example: a climate-health briefing generated on June 14, 2026 should be replayable against the same lakehouse manifest, Croissant/CDIF sidecars, TypeDID envelopes, OpenLineage event, and DID attestation hash. If the answer changes after a model upgrade or a dataset correction, Querygraph should show what changed.

The Data Infrastructure for Agentic AI

The QueryGraph.ai data-infrastructure post argues that responsible AI needs context, provenance, and governance. Querygraph maps those directly:

- Context comes from Semantic Croissant, CDIF, OSI, controlled vocabularies, and Grust graph relationships.
- Provenance comes from Dataverse metadata, file hashes, OpenLineage events, DIDs, and signed TypeDID envelopes.
- Governance comes from RBAC, ODRL, TypeSec capabilities, and access receipts.

Example: an agent asks for a mobility-risk prediction. Querygraph does not hand it every transportation table. It resolves the question to a mobility compartment, identifies dockless-transportation and pedestrian-injury tables, checks whether the agent may derive a summary, records the run, and returns only a signed summary.

Graph and Vector Navigation

The Palefire posts point toward graph-plus-vector navigation. Querygraph's current Rust implementation emphasizes the graph and lakehouse side, but the architecture leaves room for vector stores. The graph identifies things: datasets, variables, policies, agents, claims, prompts, and lineage events. Vectors help discover similar things: related papers, near-synonymous terms, translation candidates, or concept clusters.

Example: a user asks about "energy burden." A vector search may find related phrases such as "energy insecurity" or "access to clean cooking." The graph then decides which are official terms, which datasets contain them, what ontology defines them, and what policies govern them.

Digital Public Infrastructure

The AgStack post matters because it shows the same pattern outside a single company. Agriculture, climate, health, food security, and geospatial identity all need shared infrastructure. Querygraph's lakehouse example is enterprise shaped, but the architecture works for public infrastructure too:

- open datasets with rich metadata;
- decentralized identities for contributors and data products;
- policies that allow reuse while protecting sensitive resources;
- graph navigation across domains;
- local AI agents that can work even when central infrastructure is limited.

Example: a regional food-security agent could combine weather, crop, soil, market, and logistics data. It should know which datasets are public, which are licensed, which are embargoed, and which are local to a cooperative.

Ontology-Driven Precise AI

The strongest version of Querygraph is ontology-driven precise AI. It is not satisfied with "probably relevant chunks." It wants the exact concept, the exact variable, the exact unit, the exact source, and the exact permission.

Precision does not mean rigidity. The navigator can still use LLMs, embeddings, and agents. But those systems operate inside a semantic frame:

1. The question is mapped to ontology terms.
2. Terms are resolved to CDIF variables and OSI business concepts.
3. Variables are mapped to Croissant fields and Sail columns.
4. DIDs identify the requester, agent, service, dataset, and attestation issuer.
5. ODRL and TypeSec determine which actions are permitted.
6. Grust traverses the graph of datasets, policies, variables, agents, and lineage events.
7. Sail executes table access and stores audit records.
8. Ollama or another model receives only the governed prompt it is allowed to process.
9. OpenLineage records the run.
10. A DID attestation signs the root.

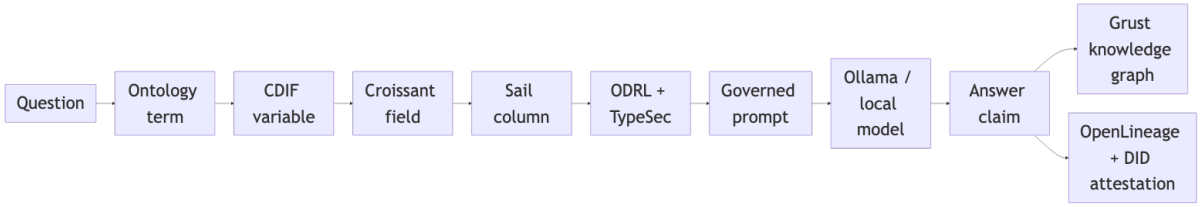

Figure 2: Diagram 2

This is the opposite of a giant ungoverned context window. It is a precise navigation path through meaning, authority, and permitted action.

Local-First, OSS AI

Querygraph is local-first by design. Local-first does not mean isolated or small-minded. It means the organization keeps its data, metadata, policies, lineage, and agent traces inspectable on infrastructure it can understand. Open-source systems matter here because responsible AI is not only a model property. It is an operational property. If the routing, retrieval, policy, identity, and audit layers are opaque, the system cannot be fully governed.

The practical target is not “never use GPUs” or “never call a frontier model.” The target is to avoid making expensive global inference the default path for questions that can be answered by precise retrieval, graph traversal, cached semantic metadata, SQL, and compact local model calls. Querygraph tries to move as much work as possible into stable, CPU-friendly infrastructure:

- parse and type the data once;
- materialize Croissant, CDIF, and OSI once per dataset version;
- cache graph neighborhoods and semantic projections;
- use Sail and Spark SQL for table computation;
- use Grust for relationship traversal;
- use TypeSec and ODRL to prune unauthorized work before retrieval;
- use local models such as Ollama only after the prompt has been narrowed.

This is the AI Navigator thesis: a well-built semantic route is cheaper, safer, and more reproducible than a giant prompt.

Source Posts

This vision section synthesizes the QueryGraph.ai posts on CODATA, Semantic Croissant, responsible AI infrastructure, vector stabilization, compression, Palefire, and AgStack:

- https://querygraph.ai/alexy-interviews-slava-tykhonov-at-codata/
- https://querygraph.ai/moving-toward-a-verifiable-ai-future/
- https://querygraph.ai/compression-not-attention-is-all-you-need-for-ai/
- https://querygraph.ai/freezing-time-in-ai-how-vector-stabilization-captures-historical-ground-truth/
- https://querygraph.ai/building-data-infrastructure-for-agentic-ai/
- https://querygraph.ai/agstack/
- https://querygraph.ai/palefire/
- https://querygraph.ai/semantic-croissant/

The AI Navigator Thesis

An AI Navigator is a system for turning a broad question into a narrow, governed, reproducible computational path. It is not merely search, not merely RAG, and not merely an agent framework. It is a control system for context.

The basic problem is easy to state. A person asks:

> Which communities face overlapping fiscal capacity, energy burden, mobility disruption, and climate-health risk?

A Big AI system is tempted to gather everything that might be relevant: finance documents, energy surveys, mobility tables, climate reports, health studies, prior briefings, policy files, and maybe a few thousand embeddings. That creates a huge context-selection problem. The model must infer meanings, ignore irrelevant fields, respect permissions, remember provenance, avoid forbidden data, and explain its answer after the fact. Some of that work may happen inside a GPU-bound model call where the organization has the least control and the least reproducibility.

Querygraph reverses the order. It does not begin with a model. It begins with the route:

1. Identify the intent.
2. Resolve the ontology terms.
3. Find the datasets and fields that actually express those terms.

4. Check rights and roles before retrieval.
5. Use graph traversal and SQL before generative inference.
6. Pass only the permitted, focused context to an agent or model.
7. Return a signed summary or a signed denial.
8. Record lineage so the route can be inspected later.

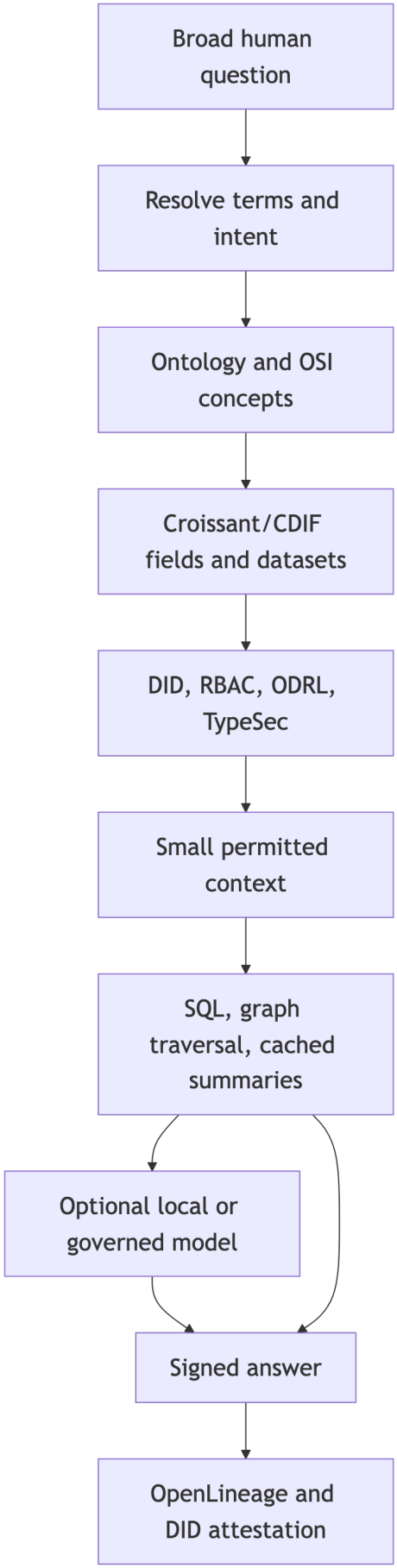

Figure 3: Diagram 3

The model becomes one component in a governed path, not the place where every hard problem is dumped.

The Alternative to Big AI

Big AI is powerful, but its default architecture is poorly matched to many enterprise and scientific tasks. It rewards centralization, massive context, remote inference, opaque optimization, and constant recomputation. Those features are useful for some problems. They are costly and risky for governed knowledge work.

Querygraph proposes a different stack:

Big AI default	Querygraph alternative
Send large context to a large model.	Build a small, permitted semantic context first.
Treat retrieval as approximate chunks.	Resolve ontology terms, variables, fields, and policies.
Recompute context for each question.	Cache dataset metadata, graph routes, and lineage.
Centralize inference around GPU services.	Keep data local and use CPU-friendly SQL/ graph work first.
Govern after the model responds.	Govern before retrieval and before prompting.
Trust logs after the fact.	Sign requests, responses, denials, and lineage roots.
Accept model drift as inevitable.	Record versions, prompts, payload hashes, and data state.

The point is not to pretend that small models always beat large models. The point is to stop using large models as a substitute for data infrastructure. When the semantic route is precise, the model has less to guess. When the context is focused, inference is cheaper. When the policy is checked before retrieval, safety is structural rather than rhetorical. When lineage is stored in the lakehouse, reproducibility becomes part of the data platform.

Focused Context as a First-Class Artifact

In Querygraph, context is not a bag of text. A focused context is a structured artifact with:

- the user question and resolved intent;
- ontology terms and OSI concepts;
- Croissant record sets and fields;
- CDIF discovery, access, and rights metadata;
- Sail table and column references;
- DID identities for requester, agent, and issuer;
- ODRL permissions and prohibitions;
- TypeSec capability receipts;
- OpenLineage input and output references.

That artifact can be cached, hashed, signed, replayed, compared, and audited. This is why Querygraph can be stable in a way that prompt-only systems are not. The expensive part of understanding the domain does not have to be regenerated inside a model context every time.

CPU-Bound Before GPU-Bound

The CPU-friendly parts of the system are not second-class. They are the foundation. Parsing a Dataverse file, inferring column types, writing Parquet, loading a Sail table, traversing a Grust graph, evaluating an ODRL permission, computing a hash, and generating an OpenLineage event are all deterministic or nearly deterministic operations. They can be tested. They can run locally. They can be repeated.

GPU-bound inference is reserved for the part that genuinely needs generation: summarizing a small body of permitted evidence, translating a user question into a candidate ontology term, or drafting a narrative from signed summaries. Even then, the model call is wrapped by TypeDID, bounded by ODRL, and recorded by OpenLineage.

That is the practical alternative to Big AI: use conventional computing for what conventional computing does well, and use models only after the problem has been made small enough to govern.

The Querygraph Spine

Querygraph is built around a simple but demanding product promise: an agent should be able to answer a serious question over enterprise data without turning the enterprise into an unbounded prompt. The system must know what the data is, where it came from, who is asking, which action is allowed, which model was used, what the answer depends on, and how to replay the run.

That promise becomes the spine of the platform:

1. Semantic Croissant describes the data as agents will encounter it.
2. CDIF publishes the same data as interoperable FAIR metadata.
3. DIDs identify agents, datasets, bundles, issuers, and attestations.
4. ODRL expresses governed data rights as machine-actionable policies.
5. TypeSec turns those identities and policies into typed capabilities.
6. Grust gives the navigator a graph of meaning, policy, lineage, and agents.
7. OSI gives business concepts stable names and relationships.
8. Sail executes the lakehouse and keeps the audit data queryable.
9. OpenLineage records runs, inputs, outputs, and derivations.
10. QG Lakehouse ties all of it together in Rust.

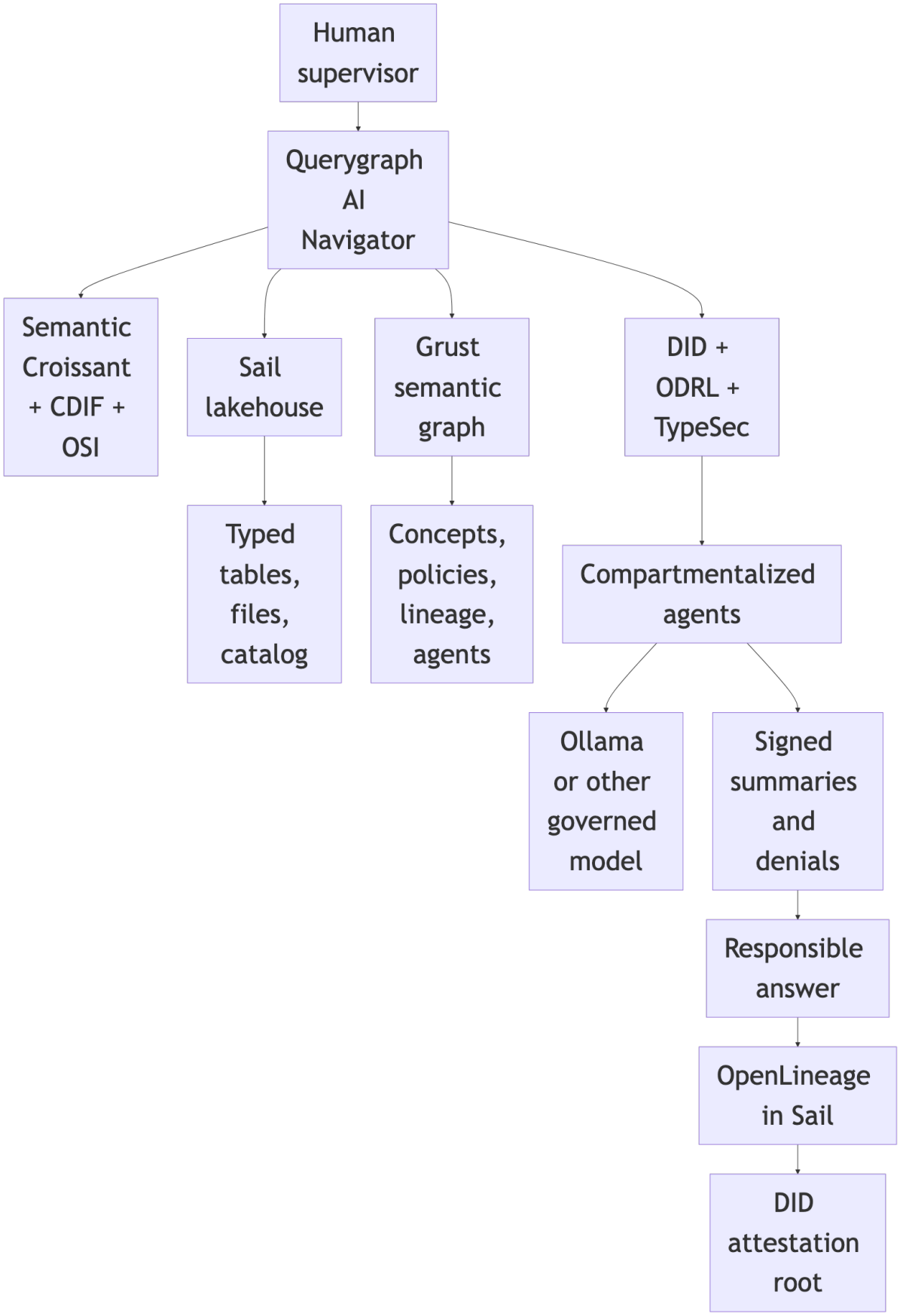

Figure 4: Diagram 4

The rest of the book walks that spine component by component. Each chapter answers three questions: why the component exists, how Querygraph uses it, and what becomes possible when it is combined with the others.

Semantic Croissant

Semantic Croissant is the moment a file becomes navigable. Before Croissant, an enterprise lake is a heap of formats: CSV files, spreadsheets, Parquet directories, APIs, attachments, survey exports, and institutional oddities with names only a local analyst understands. After Croissant, an agent can ask a more disciplined question: what datasets exist, what files belong to them, what record sets they contain, what fields exist, what those fields mean, and which types should be expected at runtime.

The first principle is simple: a model should not have to inspect raw data in order to learn what the data is. If every agent begins by sampling files, guessing schemas, and inferring meanings from column names, the system wastes compute and invites mistakes. Semantic Croissant moves that knowledge into a stable metadata artifact. The file can be large, private, or expensive to query, while its shape remains small, public enough for planning, and cacheable.

For an AI Navigator, this is the first compression step. Instead of asking a model to read a million-row survey to discover that a field means household energy source, Croissant gives the model and the policy engine a structured description of the field. That description can be reused across agents and runs. It can also be compared against the actual table, which makes drift visible.

In Querygraph, Croissant is not decorative metadata. It is the first contract between the lakehouse and the agent. A model should not see raw rows until a semantic layer has explained the shape of those rows. That explanation is not only for the model. It is for policy, validation, lineage, and replay.

The Rust implementation lives in `croissant.rs`. It defines a compact model of datasets, file objects, record sets, and fields, then emits JSON-LD sidecars. When the lakehouse loader parses a Dataverse or CODATA asset, it does two things at once: it materializes a typed table for execution, and it materializes a Croissant description so the table has a semantic face.

Consider the energy access survey in the demonstration lakehouse. A column may look like an integer. Croissant lets Querygraph say more: this field came from this file, belongs to this record set, represents this survey variable, has this inferred type, and should be interpreted under this dataset. That is the difference between an agent guessing and an agent navigating.

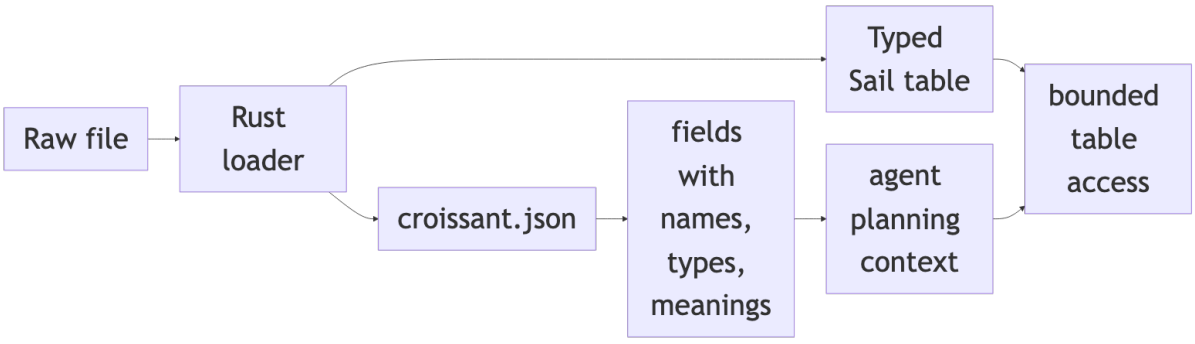

Figure 5: Diagram 5

The important practical detail is that Croissant remains close to the data. It describes the concrete files and tables that actually exist. Querygraph does not ask Croissant to decide access policy, replace business ontology, or store lineage events. It gives the navigator a trustworthy map of the terrain.

Textbook rule: Croissant answers “what is physically and semantically present?” It does not answer “who may use it?” or “which business question does it serve?” Those are ODRL and OSI questions. Keeping those layers separate is how Querygraph avoids turning metadata into another ungoverned blob.

CDIF

CDIF answers a different question from Croissant. Croissant says, “Here is how this dataset is structured.” CDIF says, “Here is how this dataset participates in a larger interoperable data ecosystem.” That distinction matters because an AI Navigator must operate both inside a local lakehouse and across institutions, domains, catalogs, and communities of practice.

The first principle of CDIF is interoperability. A local system can be precise and still be provincial. If each repository describes discovery, access, rights, variables, and provenance differently, agents cannot move responsibly across domains. CDIF gives Querygraph a way to publish local assets in a language that other FAIR data systems can understand.

This is another answer to Big AI. Instead of asking a model to infer cross- domain meaning from whatever text happens to be nearby, CDIF makes the interoperability layer explicit. Discovery, manifest, data description, access, rights, vocabularies, integration, universals, and provenance become structured profiles rather than hidden prompt assumptions.

In Querygraph, CDIF is the publication projection over the Croissant-grounded asset. The Rust module `cdif.rs` projects datasets into profiles for discovery, manifest, data description, data access, access rights, controlled vocabulary, integration, universals, and provenance. Those profiles are not bureaucratic checkboxes. They are the handles that make cross-domain AI possible.

Imagine a resilience analyst asking whether fiscal fragility and energy insecurity overlap in vulnerable communities. Finance tables and energy survey tables do not naturally speak the same language. CDIF helps Querygraph describe the asset in a way another system can discover, compare, cite, and connect. The CDIF projection gives the navigator publication-grade metadata while Croissant remains the close-up record-set description.

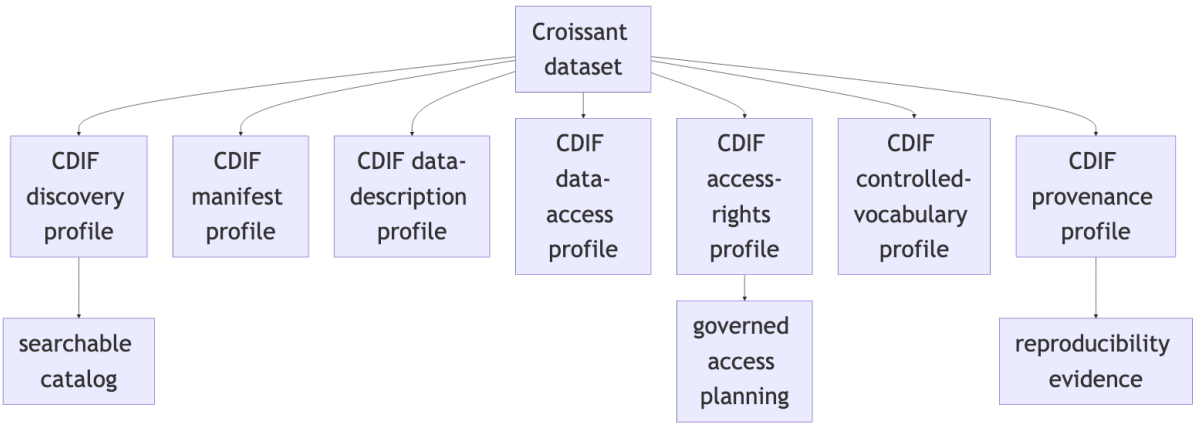

Figure 6: Diagram 6

CDIF lives beside the lakehouse, not above it as an abstract aspiration. For each dataset Querygraph loads, the sidecar `semantic/cdif.json` travels with the corresponding prepared data and Croissant sidecar. The validator checks that these semantic artifacts remain shaped correctly, because stale metadata is worse than no metadata: it gives an agent confidence in the wrong map.

Textbook rule: Croissant makes a dataset locally legible; CDIF makes it federation-ready. Querygraph needs both because the navigator must be precise inside one Sail warehouse and intelligible across many catalogs.

DID

DIDs give Querygraph names that do not depend on a single database row or cloud account. In ordinary software, a user, job, dataset, or model run may be identified by whatever the local application happens to assign. In responsible agentic AI, that is too weak. The system needs identifiers that can be carried across messages, signatures, attestations, policies, and ledgers.

The first principle is portable accountability. An agent is not merely a process. It is an actor in a chain of delegation. A dataset is not merely a path. It is an object that can be cited, governed, signed, and audited. A model answer is not merely text. It is a claim made by an identified actor over an identified context. DIDs give those actors and objects stable handles.

This matters especially in local-first AI. A local system should not have to ask a central cloud service for permission to name its agents or sign its lineage. Deterministic local DIDs let demos and offline workflows preserve the same identity pattern that a production deployment can later anchor more strongly.

Querygraph uses DIDs for agents, bundles, issuers, and attestations. The demo implementation in `did.rs` provides deterministic local `did:oyd` documents so the examples can run without external ceremony. TypeSec then lifts those identities into TypeDID envelopes and typed capability checks.

The practical value appears when something goes wrong. Suppose a supervisor receives a summary that claims a climate-health pathway overlaps with an energy-burden cluster. Querygraph should be able to say which agent produced the claim, which DID identified that agent, which dataset DIDs or bundle DIDs were in scope, which issuer signed the lineage attestation, and which payload hash was signed. The DID is the thread that lets the answer be pulled back through the system.

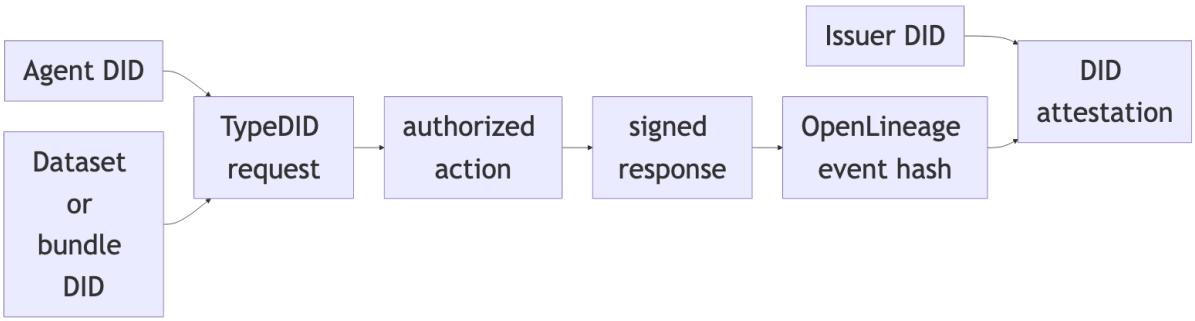

Figure 7: Diagram 7

The DID ledger is intentionally compact. Querygraph does not need to store every raw row or every full OpenLineage event in a DID ledger. The lakehouse is better for large queryable event bodies. The ledger should store roots, hashes, issuers, subjects, and signatures: enough to prove that the larger record has not been quietly rewritten.

Textbook rule: DIDs identify and attest; they do not replace the warehouse, the catalog, or the policy engine. Their job is to keep accountability portable across those systems.

ODRL

ODRL is the rights language in Querygraph. It is the place where permissions, prohibitions, duties, constraints, targets, assigners, and assignees become machine-actionable. Querygraph should not

invent a second name for that layer. It should use ODRL clearly and then explain how RBAC, DIDs, TypeSec, and Sail surround it.

The first principle is that governance must happen before context assembly. Many AI systems retrieve first and redact later. That is backwards. If an agent is not allowed to read respondent-level health data, those rows should not enter its prompt, vector search, cache, or intermediate scratchpad. ODRL lets Querygraph express that rule as data rather than as a warning in a prompt.

ODRL also turns denial into an accountable event. A system that only logs successful access is incomplete. Querygraph records signed denials because a future reviewer needs to know not only what evidence was used, but also what evidence was correctly excluded.

The Rust implementation in `odrl.rs` models the part of ODRL needed by the current demos: policy targets, permissions, prohibitions, actions, assignees, and a simple `allows` decision. That is intentionally small, but it preserves the key discipline: an agent action is allowed only when the policy grants it and does not prohibit it.

An ODRL policy should be able to say:

- this agent may read public finance data;
- this agent may summarize energy survey data but may not export raw rows;
- this broker may reveal restricted dataset metadata but not restricted respondent-level records;
- this model may receive a redacted prompt but not sensitive columns;
- this supervisor may aggregate signed summaries without inheriting every raw permission held by a specialist.

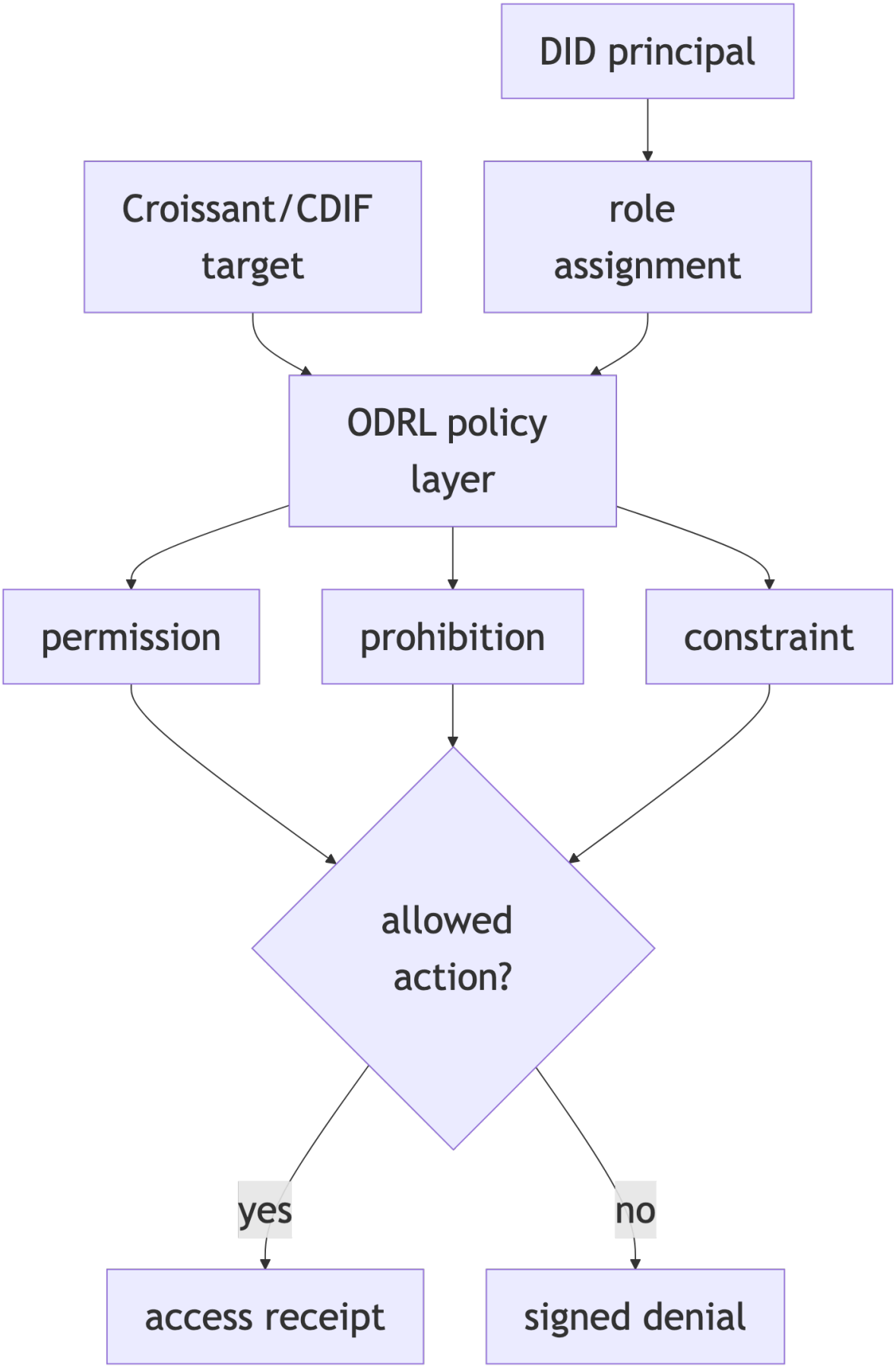

Figure 8: Diagram 8

This chapter is where responsibility stops being a slogan. A responsible system must be able to deny access in a way that is as legible and auditable as approval. Querygraph treats ODRL denials as first-class outputs. They are signed, included in lineage, and passed to synthesis agents so the final answer knows which evidence was deliberately not used.

Textbook rule: ODRL is a pre-retrieval filter, not a post-answer apology. It shrinks the computational problem while making the ethical boundary explicit.

TypeSec

TypeSec is the security fabric that makes the policy layer programmable without making it squishy. Querygraph needs more than bearer tokens. Agents delegate to other agents. Prompts become operational artifacts. Model calls need bounded capabilities. Responses need signed provenance. A token that says "this process is authenticated" is not enough.

The first principle is typed authority. In ordinary API security, a service may receive a token and then decide what that token means inside application code. Agentic systems need something sharper. The system should know that this agent, in this conversation, may perform this action, over this resource, with this payload hash, under this policy. TypeSec gives Querygraph that shape.

This is a direct alternative to "trust the orchestrator." A LangChain planner, an Ollama call, a local script, and a Rust service should all receive bounded capabilities rather than broad ambient authority. That keeps experimentation possible without letting every experiment become a privileged data channel.

TypeSec brings typed security to that world. In Querygraph, `agent.rs` builds TypeDID request envelopes, access receipts, governed prompts, and signed responses. The TypeDID protocol gives each agent interaction an identity-bound envelope. The policy decision can then mint typed capabilities such as `read`, `summarize`, `derive`, `normalize`, or `ai:infer` for a specific resource and action.

The strongest example is the Ollama path. A local model is useful because it keeps inference close to the data. But local inference is still dangerous if the prompt is an ungoverned blob. Querygraph wraps the prompt in a TypeDID envelope, checks the resource and action, and sends only the governed prompt to the model. The response comes back as another signed artifact rather than a loose string.

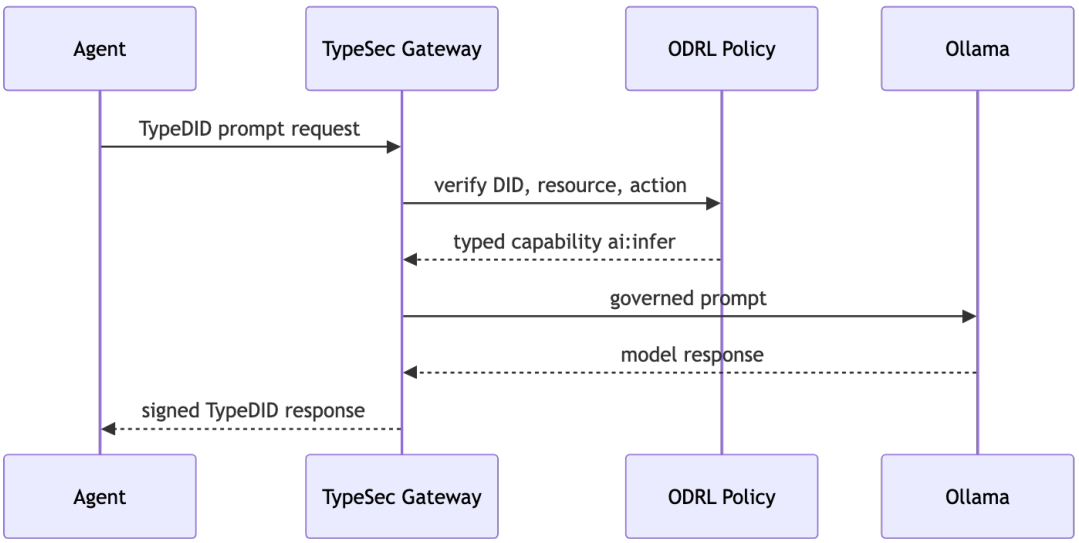

Figure 9: Diagram 9

This is where TypeSec and DIDs become more than identity plumbing. They let Querygraph preserve compartmentalization through an agent hierarchy. A synthesis agent can receive signed summaries from specialists without automatically receiving the raw permissions that produced those summaries.

Querygraph tracks TypeSec 0.13.1, "Torcello," the fourth Venetian-landmark release after Murano and Burano. Burano is what made the cross-agent envelopes in the Ollama path trustworthy as evidence: each authorized interaction carries an audit-safe TypeDID attestation recording who did what to which resource, at which privacy level, without ever exposing the payload or the signing material. Torcello grows the same fabric into a security platform other agent stacks plug into — an interop plane that guards OpenAI, Anthropic, LangChain, and Pydantic-AI tool calls; a deny-by-default MCP gate; signed decision receipts with logging and replay; schema-validated tool bindings; and an OpenAI/Anthropic-compatible enforcement proxy.

Textbook rule: DIDs say who is acting; ODRL says what action is allowed; TypeSec turns that decision into a typed, signed capability that software can carry safely.

Grust

Grust gives Querygraph its graph mind. The lakehouse stores tables beautifully, but an AI Navigator needs relationships: dataset contains file, file contains record set, record set contains field, field maps to concept, concept belongs to ontology, policy targets asset, agent has role, run consumed input, answer derived from summary. These are graph-shaped facts.

The first principle is that meaning is relational. A column does not become useful merely because it has a name. It becomes useful because it is connected to a dataset, record set, ontology term, policy, lineage event, and agent workflow. A table engine is excellent at scanning rows. A graph engine is excellent at following those relationships.

This is why Querygraph does not treat vector search as the whole retrieval story. Vectors can find similarity, but they do not by themselves prove that a field belongs to an approved concept, that an agent has the right role, or that a previous answer used the same dataset version. Grust gives the navigator the explicit route.

In Querygraph, Grust is the property graph substrate that makes those facts traversable from Rust. The module `sail.rs` stages Dataverse metadata and semantic graph nodes through the Grust Sail adapter. The graph is not an ornament beside the lakehouse. It is how the navigator plans safe routes through data.

For example, an analyst may ask for mobility disruption in areas with fiscal constraints. The graph can connect "mobility disruption" to transportation datasets, injury severity tables, dockless transportation fields, urban-form features, policies, and lineage from previous runs. Sail can then execute the table operations. The graph decides where to go; the lakehouse carries the weight.

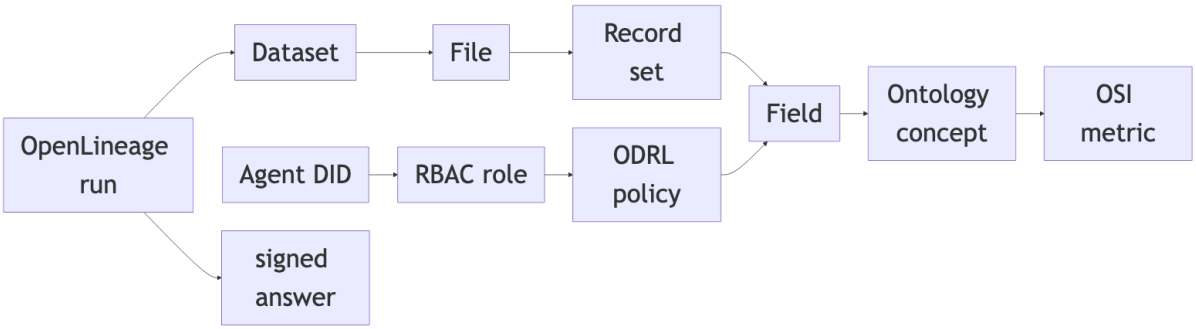

Figure 10: Diagram 10

Rust matters here because graph navigation becomes systems programming when it is part of a security boundary. Querygraph benefits from explicit types, predictable serialization, careful error handling, and the ability to keep graph, policy, metadata, and CLI code in one compiled implementation.

Querygraph tracks Grust 0.12.1, the "Lobster" release. Crab was the moment the graph gained a language: a standards-conformant GQL/Cypher layer — lexer, parser, AST, and semantic analysis — over the same property graph, with backend read pushdown into Sail and SQLite. It also gave the navigator first-class Decimal, Duration, and temporal values that order and compute correctly, and catalog procedures such as `CALL db.labels()` for introspecting the graph the lakehouse projects. Lobster completes the language: the merged Full39075 GQL profile brings `CALL { … }` subqueries, table-valued functions, `shortestPath()`/`allShortestPaths()`, backend-native passthrough escape hatches, and atomic Cypher transaction batches. The graph stops being only a store of facts and becomes a queryable substrate.

Textbook rule: use vectors for fuzzy discovery; use graphs for accountable navigation. Querygraph needs both, but the graph is what turns retrieval into a route that can be explained.

OSI

Open Semantic Interchange is the business-meaning layer. Croissant knows the shape of a dataset. CDIF knows how to publish it across ecosystems. OSI names the business concepts that make the dataset useful to an enterprise or public mission: metrics, dimensions, terms, relationships, and semantic models.

The first principle is that users ask business and scientific questions, not table questions. "Energy burden" is not a file format. "Fiscal capacity" is not a column type. "Mobility disruption" is not guaranteed to appear as a literal label. OSI gives those concepts a stable home so the navigator can map human intent onto executable data.

This is how Querygraph avoids the worst form of RAG: retrieving documents that sound related and hoping the model invents the right metric. With OSI, a term can point to dimensions, measures, expressions, fields, units, and allowed uses. The model can still help interpret the user question, but it is no longer solely responsible for defining the domain.

Without OSI, an AI Navigator can still find columns. With OSI, it can find the right concept. That distinction matters when users ask ordinary human questions. "Energy burden" may not be a column. It may be a concept composed from survey variables, household context, geography, unit conventions, and policy constraints. OSI gives Querygraph a place to model that concept instead of hoping a vector search finds a nearby phrase.

The Rust module `osi.rs` loads or synthesizes an OSI model over datasets. In a small demo, the model can be generated from Dataverse metadata. In a serious deployment, the OSI model should be curated by domain experts and versioned like application code.

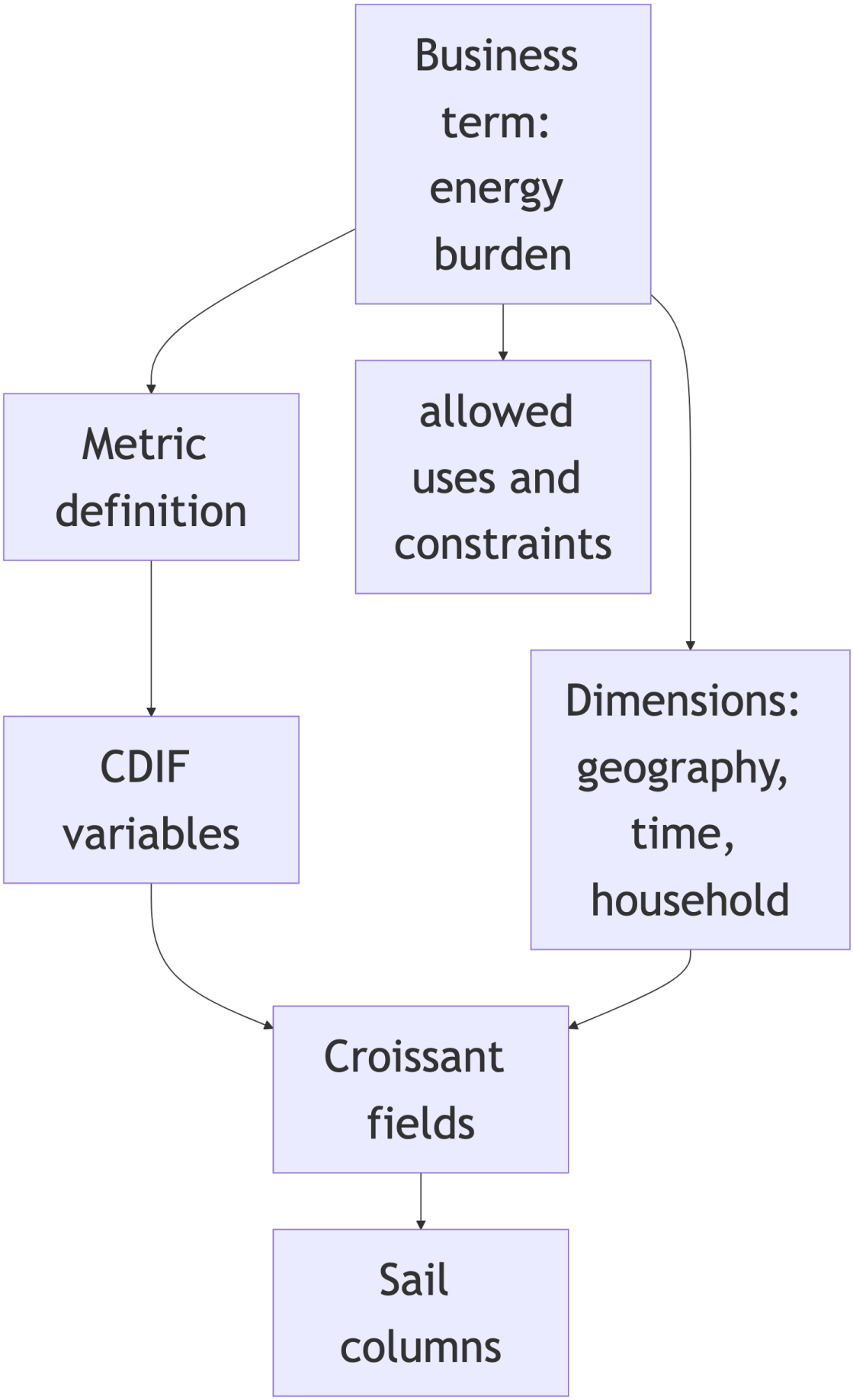

Figure 11: Diagram 11

OSI is where ontology-driven AI becomes pleasant to use. Users should not need to know table names to ask precise questions. Agents should not need to infer business meaning from column labels alone. OSI provides the semantic bridge.

Textbook rule: OSI is the layer that turns local metadata into domain language. It is what lets focused retrieval start from a human question rather than a warehouse schema.

OpenLineage

OpenLineage is the memory of what actually happened. Querygraph can generate beautiful metadata and enforce careful policies, but an operator still needs the operational story: which run executed, when it started, which job produced which output, which datasets were inputs, which facets described the run, and which event completed the derivation.

The first principle is reproducibility. A responsible answer is not only an answer that sounds right. It is an answer with a route behind it. If two runs produce different results, the operator should be able to compare data versions, prompts, model paths, policies, and input scopes. OpenLineage gives Querygraph an operational grammar for that comparison.

This is also a cost-control mechanism. Lineage lets the system reuse what is already known. If a dataset has been loaded, profiled, summarized, and attested, a future agent can inspect that history before recomputing. Stable history is one of the ways Querygraph avoids wasteful global recomputation.

The implementation in `lineage.rs` constructs OpenLineage events, writes JSONL and HTTP sinks, writes Sail audit rows, and creates TypeSec-backed DID attestations. The key design choice is that OpenLineage belongs in Sail itself for the local lakehouse. The event body is operational data. It should be queryable beside the tables and metadata it describes.

When QG Lakehouse produces a resilience briefing, the lineage event records the input scopes for finance, energy, mobility, climate-health, reference data, and restricted metadata. The output is the briefing artifact. The DID attestation then signs a compact hash of that event. Auditors get both convenience and cryptographic accountability: query the full event in Sail, verify the root in the DID ledger.

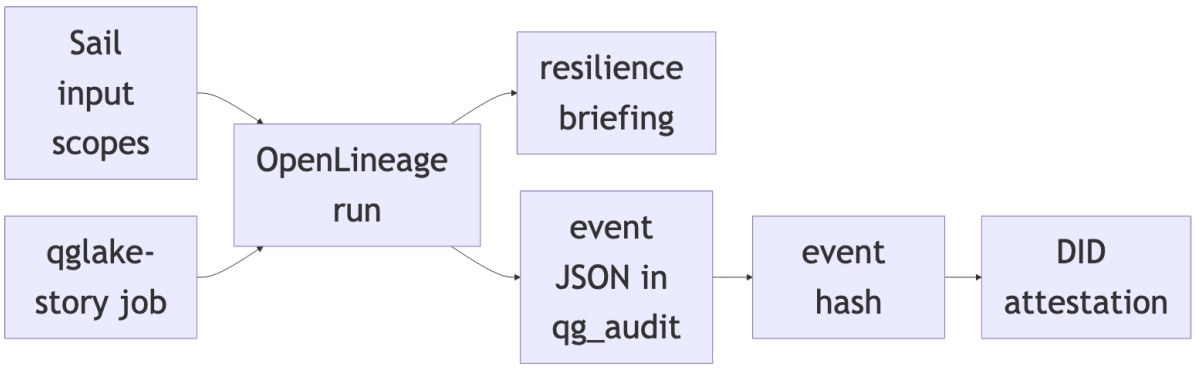

Figure 12: Diagram 12

This is reproducibility as a product feature. The answer is not just text. It is text with an execution trail.

Textbook rule: lineage makes context durable. Without lineage, every answer is a rumor; with lineage, an answer becomes an inspectable derivation.

Sail

Sail is the right lakehouse substrate for Querygraph because it makes the data layer local, inspectable, Spark-compatible, and Rust-friendly. Querygraph wants to load real Dataverse and CODATA assets, infer strong column types, expose tables to PySpark and Spark Connect, and store audit events next to the data. Sail gives that work a serious execution surface without forcing the demo into a remote warehouse account.

The first principle is that governed AI needs a governed data substrate. Prompting over loose files is not enough. The system needs typed tables, stable locations, query execution, catalog records, and audit tables. Sail gives Querygraph a local lakehouse where data and evidence can live together.

Sail is also part of the alternative to Big AI. If ordinary SQL and Spark operations can answer part of a question, they should. Counting rows, joining tables, filtering by geography, and reading audit events do not require a large language model. They require a reliable execution engine. The model should receive the result of that focused computation, not the entire raw warehouse.

The local schema is `qg_lakehouse`. It contains typed tables and catalog records such as `lakehouse_datasets`, `lakehouse_files`, and `lakehouse_columns`. The audit schema is `qg_audit`. It stores OpenLineage events and DID attestations. That separation mirrors the product boundary: data and metadata in one governed space, operational evidence in another governed space, both queryable.

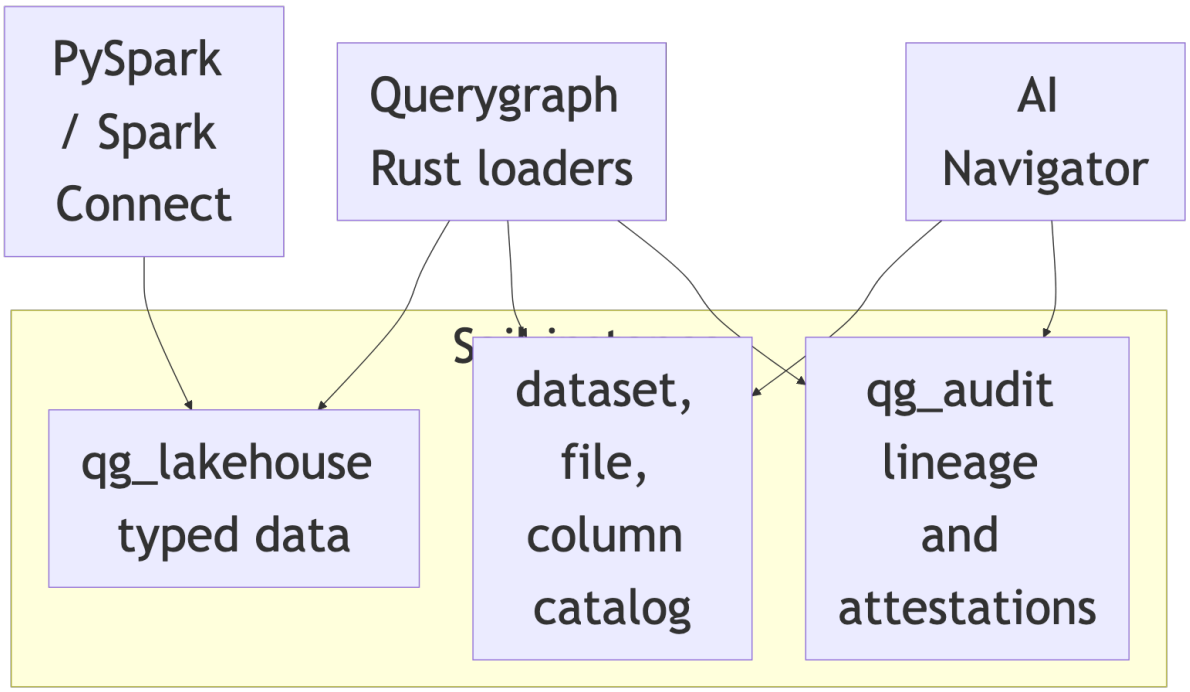

Figure 13: Diagram 13

Rust is the right implementation language for this because the system lives at the intersection of parsing, security, metadata, policy, graph traversal, serialization, and reproducibility. These are not soft edges. A loader should not silently coerce a sensitive identifier into nonsense. A policy evaluator should not accidentally treat an absent prohibition as a broad grant. A lineage attestation should not hash a different payload than the one written to audit storage.

Rust helps Querygraph make those boundaries explicit. `Result`-driven error handling keeps ingestion honest. `serde` keeps JSON-LD, OpenLineage, and TypeDID envelopes structured. Strong enums make action vocabularies and event types harder to confuse. Cargo keeps the CLI, library,

tests, and book examples close enough to evolve together. Sail gives the execution layer; Rust gives the control layer.

Textbook rule: Sail is where focused retrieval becomes executable. It keeps the data local and queryable so the AI layer can remain small, governed, and inspectable.

Python Interop

Rust is the control plane for Querygraph, but Python is the working surface for many of the people who will use it. Data scientists live in notebooks. Spark users expect PySpark. AI engineers assemble agents with Python libraries. Analysts want to inspect a Sail warehouse without learning the Rust internals. The sister project `qg-python` exists for that world.

The first principle is that responsible AI must meet practitioners where they work without abandoning the system's guarantees. If the Rust implementation is precise but the Python notebook path is loose, the platform fails. If the notebook can bypass ODRL, TypeDID, or lineage, the local-first story collapses. The Python ecosystem therefore mirrors the same concepts with Python-native tools rather than inventing a separate, weaker layer.

The Python implementation is not a toy wrapper around a command line. It is a Python-native ecosystem over the same concepts:

- `croissant.py` and `cdif.py` build Semantic Croissant and CDIF JSON-LD.
- `osi.py` projects Semantic Croissant fields into OSI datasets, metrics, expressions, and ontology terms.
- `typedid.py` models TypeDID envelopes, governed prompts, access receipts, and signed agent responses with Pydantic.
- `agents.py` exposes compartment agents and an optional LangChain `StructuredTool` adapter.
- `lakehouse.py` registers Sail warehouse Parquet tables in a PySpark/Spark Connect session.
- `lineage.py` emits OpenLineage run events and DID-style attestations.
- `odrl_rights.py` and `rbac.py` give Python agents the same rights checks used in the Rust story.
- `dataverse.py` projects Dataverse native API payloads into Semantic Croissant.
- `qglake.py` runs the Python version of the supervised QG Lakehouse story.

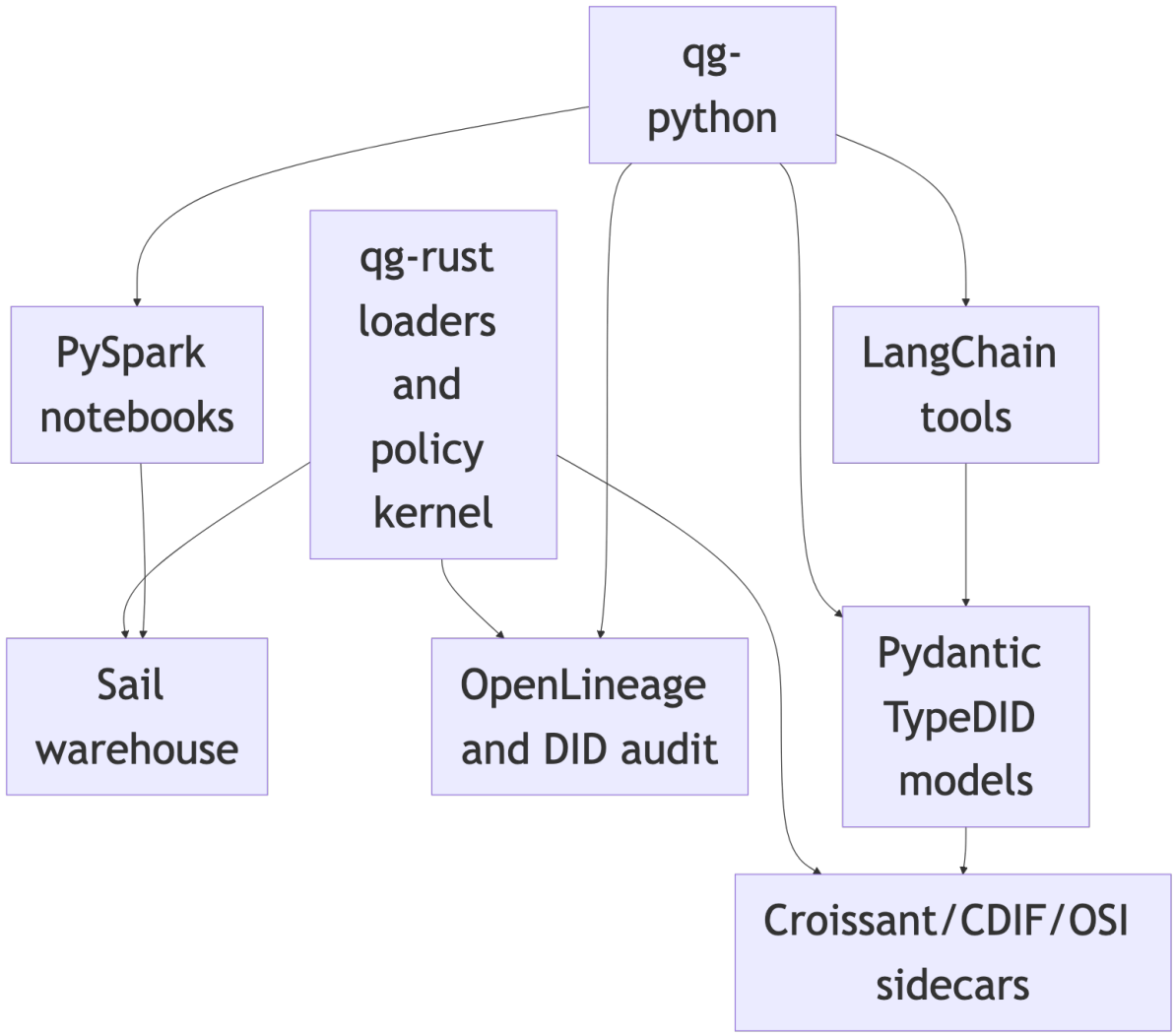

Figure 14: Diagram 14

This division is deliberate. Rust is where Querygraph wants tight control: ingestion, typing, hashing, policy boundaries, graph staging, and reproducible CLI workflows. Python is where Querygraph wants fluent exploration: notebooks, PySpark queries, LangChain tool composition, and domain-agent iteration.

Textbook rule: Python is the laboratory; Rust is the contract. Both must speak the same semantic, policy, identity, and lineage language.

Pydantic

Pydantic is the Python side's type boundary. A TypeDID envelope should not be a loose dictionary passed from one agent to another. A governed prompt should have a question, semantic context, allowed sources, denied sources, and access receipts. An agent response should have a status, summary, evidence, redactions, and an envelope hash. Pydantic makes those shapes explicit while remaining natural for Python users.

In `qg-python`, `TypeDidEnvelope` validates the request/reply structure and recomputes payload hashes. `GovernedPrompt` carries the semantic context that came from Croissant, CDIF, OSI, and Sail. `AgentResponse` preserves the signed summary or denial. The synthesis agent receives those models, not ad hoc JSON.

The point is not type ceremony. The point is that Python agents can be creative without being unbounded. Pydantic gives the agent framework a shape that can be validated, logged, hashed, and compared to the Rust implementation.

LangChain

LangChain fits as an adapter layer, not as the source of authority. Querygraph does not ask LangChain to decide whether a model may see restricted data. That decision belongs to DID identity, ODRL policy, TypeSec capability checks, and the semantic target in Croissant/CDIF/OSI. LangChain receives a governed tool only after those boundaries exist.

The Python adapter `TypeDidLangChainToolAdapter` turns a TypeDID agent into a LangChain `StructuredTool`. The tool returns the same signed response or denial that a non-LangChain caller would receive. This keeps LangChain useful while preventing it from becoming a policy bypass.

Example: a LangChain planner may choose `FinanceAgent` to summarize fiscal capacity. The actual tool invocation still goes through a TypeDID request, policy receipt, payload hash, and signed response. If the planner asks the restricted broker for raw health rows, the tool returns a signed denial.

PySpark

PySpark is the inspection and analysis surface for Sail. The Rust loader materializes the warehouse. Python registers the generated Parquet tables into a Spark Connect session and lets analysts query them with familiar Spark SQL. That is how a user can inspect `qg_lakehouse` without leaving the Python world.

The current Python helper can register the loaded data tables and the audit tables:

```
uv run querygraph lakehouse-register \
  --manifest ../qg-rust/.querygraph/lakehouse/manifest/load-report.json \
  --warehouse ../qg-rust/spark-warehouse

uv run querygraph audit-register --warehouse ../qg-rust/spark-warehouse
```

Then a notebook or shell can ask:

```
spark.sql("SELECT COUNT(*) FROM global_temp.government_finance__countydata").show()
spark.sql("SELECT quantity, value, unit FROM
global_temp.codata_constants_2022__codata_constants_2022 LIMIT
5").show(truncate=False)
spark.sql("SELECT event_hash, event_type, job_name FROM
global_temp.openlineage_events LIMIT 10").show(truncate=False)
```

That is the interop story in one loop: Rust loads, Sail stores, Python queries, Pydantic agents reason, LangChain orchestrates when useful, and OpenLineage records the trail.

The Demonstration Datasets

The default corpus is intentionally broad. It is not a toy table. It mixes finance, energy, transportation, health, climate, social science, geospatial assets, and reference data so the navigator has to cross real semantic boundaries.

The first principle is that an AI Navigator should be demonstrated on messy, multi-domain data, not a polished single-table example. The point is not to show that a model can summarize a CSV. The point is to show that governed AI can move across domains while keeping each dataset's shape, rights, lineage, and meaning visible.

The corpus is also a computational argument. A Big AI demonstration might stuff documents and table samples into a prompt. Querygraph instead turns the corpus into a lakehouse, sidecars, graph nodes, policies, and lineage. Once that work is done, later questions reuse the structure. The system pays the metadata cost once, then benefits from focused retrieval many times.

Dataset	Category	Persistent ID or source	Typed tables	Rows	Why it matters
Government Finance Database	finance	doi:10.7910/DVN/LMS8NT	6	1,724,447	Fiscal capacity, county/ municipal/ district budgeting, and public-sector constraints.
Roadway vulnerability LiDAR DTM	geospatial	doi:10.7910/DVN/1VT6FZ	0	0	Non-tabular assets and geospatial metadata; proves the catalog can track assets that are not immediately typed tables.
ACCESS 2018 energy survey	energy	doi:10.7910/DVN/AHFINM	3	35,779	Household access to clean cooking energy and electricity.
Dockless transportation study	transportation	doi:10.7910/DVN/B2LJSB	7	479,853	Urban form, trip hotspots, mode shift, and mobility disruption.
HAALSI Baseline Survey	health	doi:10.7910/DVN/F5YHML	0	0	Restricted or inaccessible raw data; demonstrates metadata-only access and signed denial.
Global Party Survey, 2019	social science	doi:10.7910/DVN/WMGTNS	5	6,033	Institutional and political context as a social-science signal.
Connecticut pedestrian injury severity COVID-19	transportation health	doi:10.7910/DVN/IYIKEQ doi:10.7910/DVN/DMANTB table	1 2	14,645 37,982	Injury, demographic disparities in transit stops, roadway

The loaded corpus verifies to 28 typed tables and 2,299,541 typed rows. The row count is not the point by itself. The point is heterogeneity: tabular files, XLSX conversion, CODATA normalization, non-tabular assets, and restricted data all travel through one governed catalog.

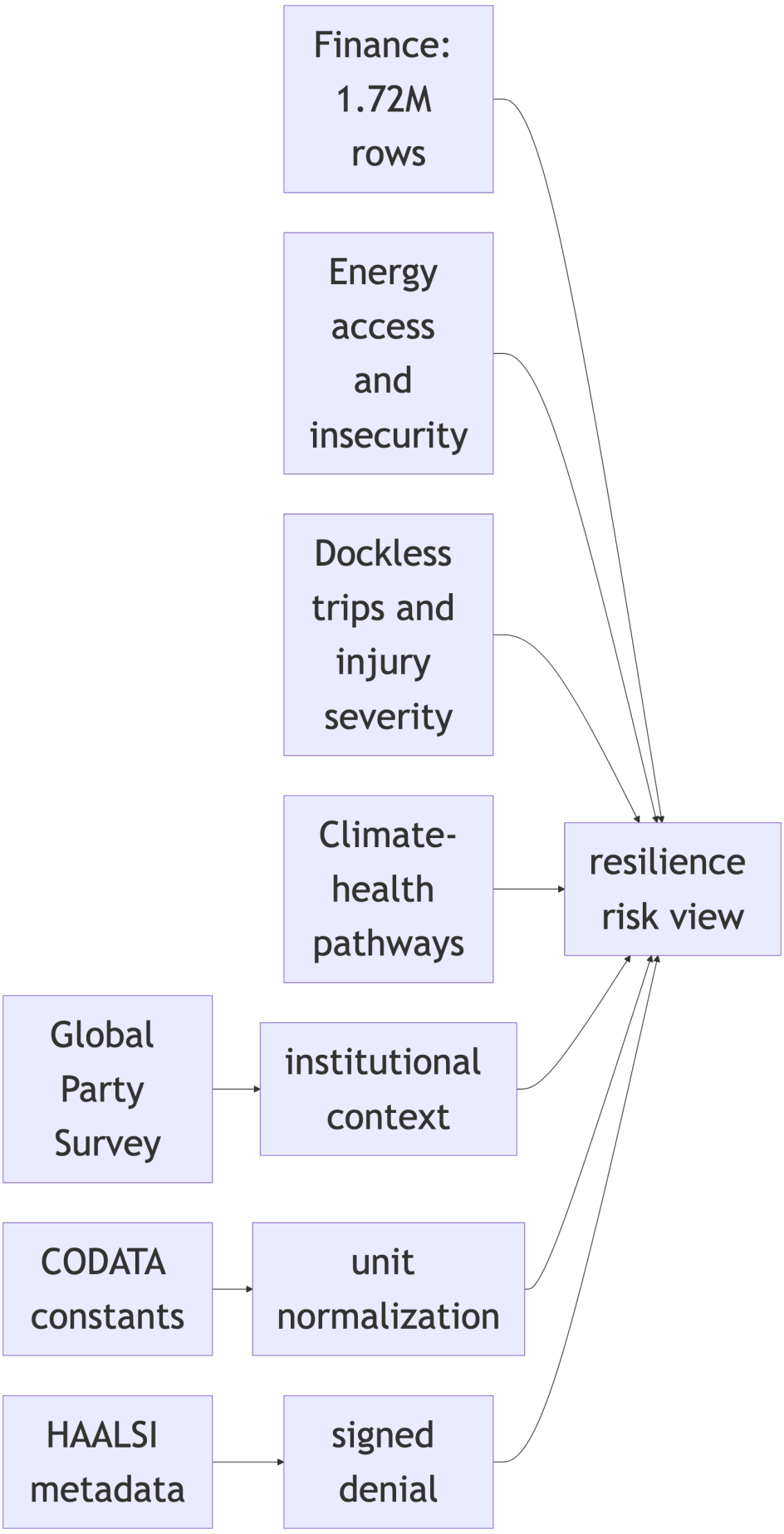

Figure 15: Diagram 15

QG Lakehouse with All Components

QG Lakehouse is where the chapters stop being separate ideas and become a single run. The executable story is:

```
cargo run -- qglake-story
```

The default output is a readable briefing. The full machine report is:

```
cargo run -- qglake-story --json
```

The story asks a mission-shaped question:

> Where do fiscal capacity, energy burden, mobility disruption, and climate-health risk overlap, and what can a supervisor responsibly know without violating restricted-data boundaries?

The answer is not a single omniscient model response. It is a governed multi-agent run over Sail, Grust, Semantic Croissant, CDIF, OSI, DIDs, ODRL, TypeSec, OpenLineage, and optional Ollama inference.

Read this chapter as a worked example, not merely a demo. Each step removes work from the model and moves it into a more reliable layer. Loading removes file ambiguity. Croissant removes schema ambiguity. CDIF removes publication ambiguity. OSI removes domain ambiguity. Grust removes route ambiguity. DID, ODRL, and TypeSec remove authority ambiguity. OpenLineage removes historical ambiguity. The final model call, if used, is smaller because the system has already done the disciplined work.

Step 1: Load the Lakehouse

The first step is ingestion. `lakehouse.rs` downloads the default Dataverse and CODATA corpus, normalizes parseable assets, infers strong column types, writes typed tables, and records a manifest. This is where Rust earns its keep: every file has to become either a typed table, a cataloged non-tabular asset, or an explicitly reported unavailable/restricted asset.

From a textbook perspective, ingestion is not plumbing. It is the first act of responsibility. If the system cannot say what it loaded, what it skipped, what it typed, and how many rows it verified, later AI claims have no foundation. Local ingestion also means the organization can inspect the data path without trusting a remote indexing service.

The lakehouse does not hide partial success. A LiDAR asset can be cataloged even when it is not a table. A restricted survey can contribute metadata while raw rows remain unavailable. The navigator can reason over both facts.

```
cargo run -- lakehouse-load --root .querygraph/lakehouse --schema qg_lakehouse
cargo run -- lakehouse-verify --report .querygraph/lakehouse/manifest/load-
report.json
```

Step 2: Materialize Semantic Croissant

For each loaded dataset, Querygraph writes a `semantic/croissant.json` sidecar. This sidecar names files, record sets, and fields so agents can inspect the data before requesting access. It is the catalog entry an agent can actually understand.

This step turns raw storage into a reusable context cache. The model does not need to rediscover schema. The policy layer can target fields. The graph can connect metadata to concepts. A future run can compare its expected fields against the sidecar before touching the data.

In the story, FinanceAgent does not receive a vague instruction to "look at finance data." It receives a semantic projection of the government-finance tables it is allowed to summarize. The projection tells the agent what tables and fields exist, and the policy layer tells it what action is allowed.

Step 3: Project CDIF

Next, Querygraph writes `semantic/cdif.json`. The CDIF sidecar takes the same dataset and expresses it through interoperable profiles: discovery, manifest, data access, access rights, controlled vocabulary, integration, universals, and provenance.

This is how QG Lakehouse avoids becoming a private demo format. The local Sail schema can be inspected by Spark, the sidecars can be shared with FAIR data tools, and the graph can connect local variables to broader community semantics.

This step matters because local-first should not mean isolated. Querygraph can keep computation local while making metadata interoperable. That is the combination serious scientific and enterprise systems need.

Step 4: Build the OSI Semantic Model

The OSI layer turns dataset metadata into business concepts. In the example, the user asks about fiscal capacity, energy burden, mobility disruption, and climate-health risk. Those are not merely table names. They are concepts that must be connected to metrics, dimensions, variables, fields, and policies.

`osi.rs` can synthesize a model from available metadata for the demo. In a real deployment, this is where domain experts make the navigator precise: they define the terms the organization actually uses and connect them to the data that can support those terms.

This is the step that changes retrieval from lexical matching to domain navigation. The navigator can ask which fields express fiscal capacity or energy burden because those concepts have been modeled before inference.

Step 5: Load the Grust Graph

Grust turns semantic metadata into navigable relationships. The graph can say that a dataset has files, a file has record sets, a field maps to a concept, a concept appears in a policy, an agent has a role, and a run produced an answer.

The graph does not replace Sail. It makes Sail usable by agents. Sail answers table questions. Grust answers route questions.

This step is the difference between retrieval and navigation. Retrieval says "here are possible matches." Navigation says "this is the route from question to concept, dataset, field, policy, run, and answer." Routes can be audited.

Figure 16: Diagram 16

Step 6: Identify Agents with DIDs

Every actor in the story has an identity: SupervisorAgent, FinanceAgent, EnergyAgent, MobilityAgent, ClimateHealthAgent, ReferenceAgent, RestrictedDataBroker, and SynthesisAgent. Those identities are represented as DIDs and carried through TypeDID envelopes.

The supervisor is powerful, but not magical. Its DID allows orchestration. It does not automatically grant raw access to every compartment. That is the central discipline of the platform.

This step prevents agent hierarchies from becoming privilege laundries. A supervisor can coordinate work, but each specialist still acts under its own identity and scoped authority.

Step 7: Apply ODRL Rights

The ODRL layer evaluates what each agent may do against a semantic target. FinanceAgent can read or summarize finance assets. EnergyAgent can derive approved energy summaries. RestrictedDataBroker can inspect restricted metadata but cannot reveal raw restricted health records.

Agent	Compartment	Allowed action	Explicit boundary
FinanceAgent	`compartment:finance`	`read`, `summarize`	No energy or health raw data.
EnergyAgent	`compartment:energy`	`summarize`, `derive`	No respondent-level restricted data.
MobilityAgent	`compartment:mobility`	`summarize`, `derive`	No finance-table mutation.
ClimateHealthAgent	`compartment:climate-health`	`summarize`, `derive`	No restricted health rows.
ReferenceAgent	`compartment:reference`	`normalize`	Can normalize units, not expand access.
RestrictedDataBroker	`compartment:restricted`	metadata-only receipt	Raw access denied.
SynthesisAgent	`compartment:synthesis`	aggregate signed summaries	Does not inherit raw specialist permissions.

The signed denial is as important as the signed summary. It prevents a supervisor from silently assuming evidence was considered when it was not.

This step is where focused computation becomes responsible computation. The system does less work because it excludes forbidden data early, and it becomes safer because the exclusion is explicit.

Step 8: Mint TypeSec Capabilities

Once a policy decision is made, TypeSec turns it into typed capability evidence. The capability is scoped to an action, resource, principal, and envelope. This is the difference between “the process is authenticated” and “this agent may perform this operation on this semantic asset for this run.”

TypeDID envelopes carry the request and response. They bind the agent identity, resource, action, payload hash, and signature. The result is an agent protocol that can be logged, replayed, and audited.

This step turns a policy decision into a software object. That object can move through Python, Rust, LangChain, Ollama, and audit tables without losing its meaning.

Step 9: Route to Compartmentalized Agents

The supervisor delegates instead of centralizing all data. FinanceAgent, EnergyAgent, MobilityAgent, ClimateHealthAgent, ReferenceAgent, and RestrictedDataBroker work inside their compartments. They produce signed summaries, normalization notes, and denial receipts.

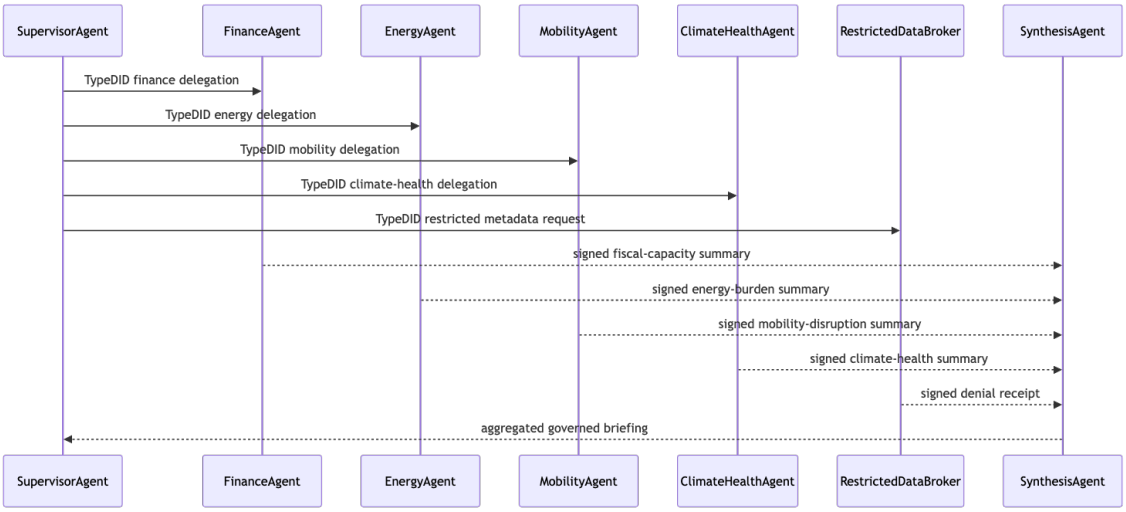

Figure 17: Diagram 17

This is the human organizational model reflected in software. A supervisor can coordinate experts without becoming every expert and without inheriting every restricted permission.

This step is also a cost-control pattern. Specialists receive small contexts and produce small signed summaries. The synthesis agent aggregates summaries rather than raw datasets.

Step 10: Call Ollama Through TypeDID

When the run uses a local model, Querygraph calls Ollama only after TypeSec has verified the governed prompt. The model receives a bounded question with approved context. It does not receive the lakehouse. It does not receive restricted rows merely because a prompt asked nicely.

In the JSON report, this appears under the Ollama TypeDID path. The important thing is not Ollama specifically. It is the pattern: any model runtime should be downstream of identity, semantics, policy, and lineage.

This step is the local-first model story. Querygraph can use an open-source local model when generation is useful, but the model call is not the system's source of truth. It is a bounded operation over a prepared context.

Step 11: Synthesize Without Boundary Collapse

The synthesis agent receives signed summaries and hashes. It aggregates them into a resilience briefing:

> Priority areas are those where weak fiscal capacity, energy burden, mobility fragility, and climate-health exposure overlap. Restricted health data contributed only a signed metadata/ denial receipt, so the briefing uses approved compartment summaries rather than raw restricted rows.

This is the product experience Querygraph is aiming for: a useful answer that also tells the truth about its limits.

This step proves that aggregation does not require universal access. A system can combine evidence without flattening compartments into one privileged prompt.

Step 12: Emit OpenLineage to Sail

The run emits a `COMPLETE` OpenLineage event. Inputs include each Sail scope. The output is the briefing. The job name, run id, producer, event time, and facets become queryable audit data in `qg_audit`.

OpenLineage in Sail means operators can ask ordinary lakehouse questions about AI behavior. Which datasets were used in this briefing? Which model path was called? Which runs touched energy survey data? Which answers included a restricted-data denial?

This step keeps AI operations inside the data platform. Audit is not a PDF appendix or a vendor dashboard. It is queryable data.

Step 13: Anchor the DID Attestation

Finally, Querygraph signs a compact attestation root. The full event remains in Sail. The DID ledger carries the issuer, subject, Merkle root, signature, and payload hash. This gives the platform a verifiable memory without turning the ledger into a dumping ground for operational data.

This step separates evidence from proof. Sail stores the full queryable event. The DID attestation stores the compact proof that the event existed in this form. That keeps the ledger small and the audit trail useful.

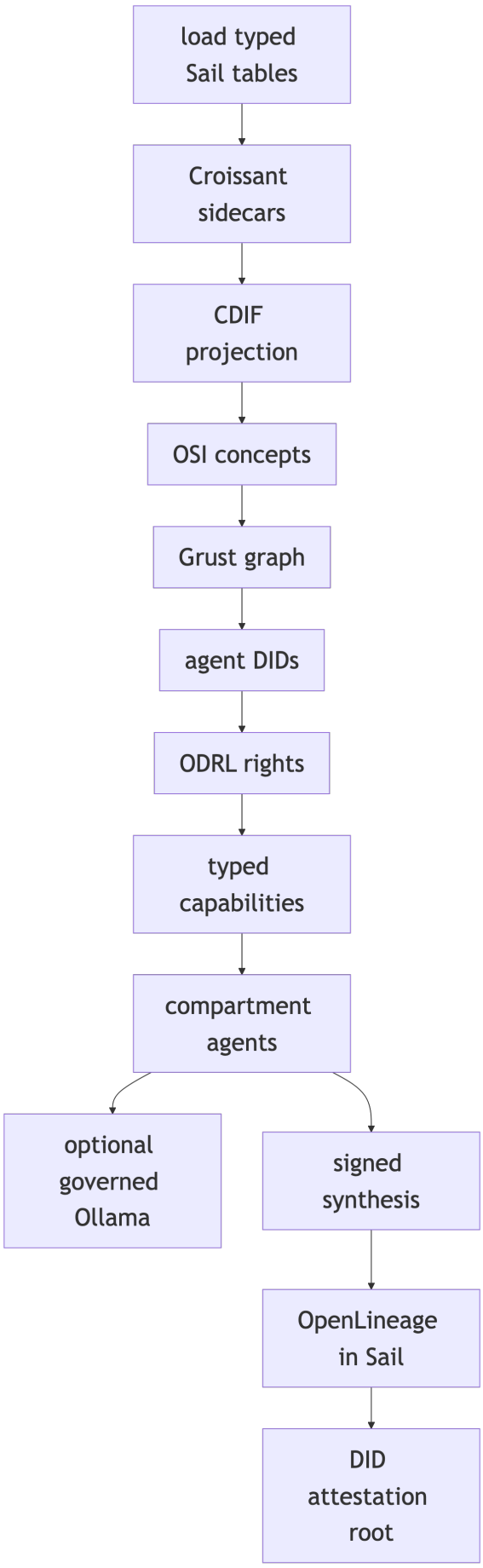

Figure 18: Diagram 18

LakeCat as the QueryGraph Catalog Boundary

LakeCat is where QueryGraph stops pretending that catalog state is background plumbing. A QueryGraph import is only useful if the catalog can prove what table, view, policy, lineage, and receipt state it accepted. LakeCat gives that proof while staying a thin Iceberg-compatible catalog: normal clients still see standard table access, while QueryGraph receives derived control-plane evidence.

The current QGLake handoff has four durable artifacts:

- a LakeCat bootstrap bundle with table, view, graph, OSI, ODRL, Croissant, CDIF, OpenLineage, and QueryGraph import hashes;
- a lineage-drain artifact replaying the outbox and OpenLineage evidence that produced the bundle;
- a QueryGraph import plan generated from the verified bundle;
- a compact handoff summary binding the saved files, captured command output, tenant scope, governed scan proof, credential proof, commit-history proof, view receipt-chain proof, and graph counts.

The important new rule is the view-chain rule. Active accepted views must carry an `acceptedReceiptChainHash` that appears in namespace `receiptChains[].chainHashes`. That prevents a handoff from pairing a valid view receipt with unrelated namespace chain evidence. Tombstoned accepted views are different: their accepted chain can be a prefix of the later tombstone chain, so the proof must instead include tombstone receipt evidence whose `expectedViewVersion` preserves the accepted view version.

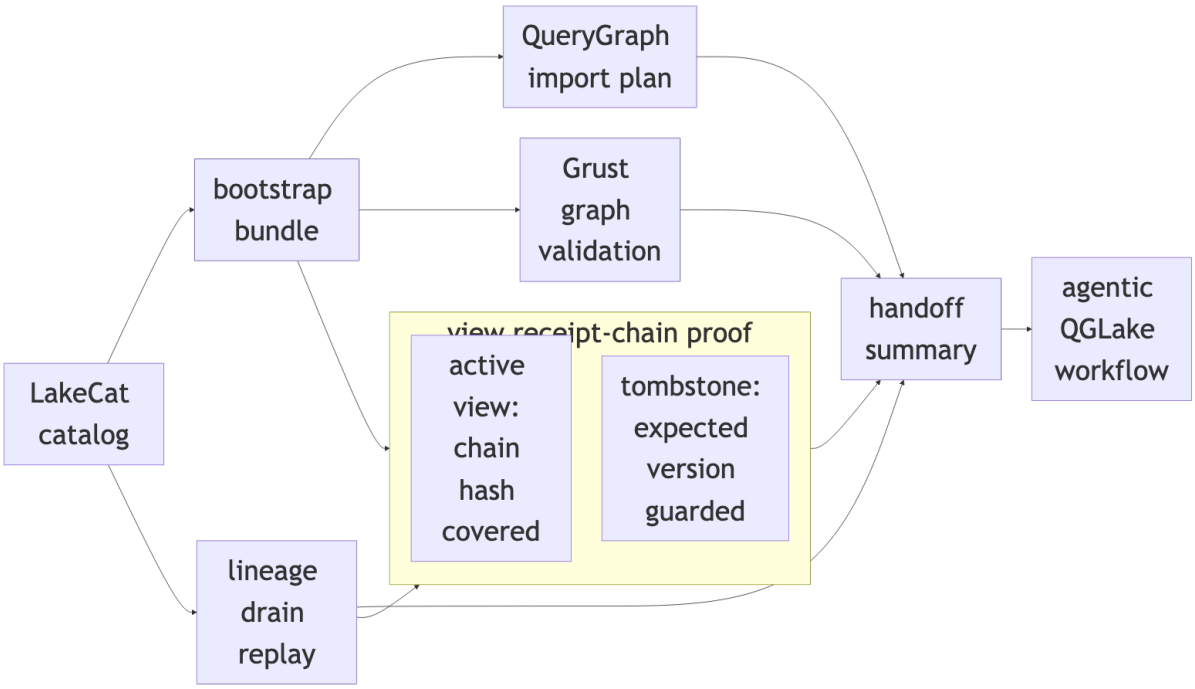

Figure 19: Diagram 19

The local integration test is intentionally concrete:

```
cd ../lakecat
scripts/qglake-handoff-local.sh
```

That command starts LakeCat, creates the local QGLake fixture, drains lineage, asks QueryGraph to verify and import the saved bundle, writes `querygraph-import-plan.json`, and verifies `handoff-summary.json`. The current passing handoff proves one table, one view, 26 outbox/lineage events,

and 53 catalog graph events. QueryGraph rejects the handoff if the saved bundle, saved drain, captured output, compact summary, graph envelope, view receipt evidence, or QueryGraph import plan drift from one another.

The same boundary shows up in code:

```
cargo run -- lakecat-verify \
  --bundle ../lakecat/target/qglake-handoff/lakecat-bootstrap.json

cargo run -- lakecat-import \
  --bundle ../lakecat/target/qglake-handoff/lakecat-bootstrap.json \
  --output ../lakecat/target/qglake-handoff/querygraph-import-plan.json
```

`lakecat-verify` recomputes the LakeCat manifest hashes for tables, views, OpenLineage, graph, and the outer bundle. It also validates the QueryGraph import compatibility contract, including `receipt-chain-hash` in view receipt evidence. `lakecat-import` then creates the import plan only after that proof has survived round-trip JSON parsing. QueryGraph does not invent catalog truth; it accepts LakeCat proof, validates the Grust graph shape, and builds the next agent context from the smallest verified scope.

QueryGraph tracks LakeCat 0.3.0, the "Ocelot" release. Lynx put the catalog spine on Turso MVCC, so commits to different tables run truly concurrently and a same-table race converges to exactly one winner through a pointer compare-and- swap — no global write lock. That matters to the handoff: the audit event, the lineage outbox row, and the idempotency record that this chapter relies on are written in the *same* transaction as the table change, which is what lets a QueryGraph import accept catalog state as proof rather than as a best-effort side effect.

Lynx also tightened this boundary in a way the importer feels directly. LakeCat extracted the bootstrap-bundle *wire format and its verification* into a small shared `qglake-bundle` crate, so QueryGraph no longer keeps a hand-written copy of those types: it deserializes the canonical `QueryGraphBootstrap` and runs LakeCat's own `verify_manifest`, then layers its Cypher import plan on top. The producer and the consumer now validate the handoff with one set of types — the bundle can no longer mean two slightly different things on the two sides of the boundary.

Ocelot proves the other side of the same boundary: stock-client Iceberg REST conformance, demonstrated by a PyIceberg round-trip against the running catalog — spec-correct error types (403 on authorization denial, 409 on a duplicate namespace, 404 on a missing one), `listTables`, and fail-closed commit-requirement validation — over dependencies moved to the Grust "Lobster" and TypeSec "Torcello" line. The catalog QueryGraph accepts proof from is the same catalog an ordinary Iceberg client simply uses.

Rust Examples

The Rust examples are the reference implementation. They show the system from the operator and platform-engineering side: load real data, materialize typed tables, generate semantic metadata, enforce rights, run agents, and emit audit evidence.

Build a Four-Layer Semantic Bundle

The smallest useful Rust run builds the original AI Navigator bundle:

```
cargo run -- navigator \
  --dataset-name "Hazard vocabulary" \
  --description "Controlled vocabulary with multilingual technical terms" \
  --landing-page "https://querygraph.ai/datasets/hazards" \
  --data-url "https://querygraph.ai/datasets/hazards.csv" \
```

```
  --creator "QueryGraph" \
  --agent-name "AI Navigator"
```

That command produces a JSON-LD bundle with:

- Semantic Croissant for dataset shape;
- CDIF for FAIR discovery, data access, access rights, and data elements;
- a DID document for the navigator;
- ODRL policy for read, index, and derive boundaries.

This is the seed pattern for every richer workflow. A dataset should always enter the agent world with shape, publication metadata, identity, and policy.

Load and Verify the Sail Warehouse

The lakehouse loader is the Rust example that proves Querygraph can handle more than toy JSON:

```
cargo run -- lakehouse-load \
  --root .querygraph/lakehouse \
  --schema qg_lakehouse

cargo run -- lakehouse-verify \
  --report .querygraph/lakehouse/manifest/load-report.json

cargo run -- lakehouse-validate \
  --report .querygraph/lakehouse/manifest/load-report.json
```

The loader downloads Dataverse and CODATA assets, prepares parseable files, infers column types, writes Sail tables, emits Croissant/CDIF sidecars, and records row counts in a manifest. The verifier checks that the executable data matches the report. The validator checks that the semantic sidecars and audit shapes are still usable.

Run the Supervised Agent Story

The readable story:

```
cargo run -- qglake-story
```

The machine report:

```
cargo run -- qglake-story --json
```

The Rust story is intentionally elaborate. It creates a supervisor, specialist agents, restricted broker, synthesis agent, TypeDID requests and responses, RBAC and ODRL receipts, Semantic Croissant and CDIF projections, an OpenLineage event, and a DID attestation. It demonstrates the product rule: aggregate signed summaries without collapsing raw-data boundaries.

Run the Live Sail, TypeDID, OpenLineage, and Ollama Path

With Sail running:

```
sail spark server --port 50051
```

Run the live end-to-end path:

```
cargo run -- dataverse-e2e \
  --live-sail \
  --sail-endpoint http://127.0.0.1:50051 \
  --openlineage-file .querygraph/openlineage/events.jsonl \
  --did-ledger-file .querygraph/did-ledger/attestations.jsonl
```

When Ollama is available, the same path can wrap model inference through TypeDID. The important property is the order: semantic target first, policy decision second, typed capability third, model call fourth, lineage fifth.

Rust as the Contract

The Rust examples define the contract that other languages should preserve. Python may be more comfortable for notebooks and agent composition, but it should not silently change CDIF shape, policy semantics, payload hashes, or lineage meaning. That is why `qg-python` includes an equivalence test against the Rust `navigator` command.

Python Examples

The Python examples are the user-facing and notebook-facing complement to the Rust examples. They do not replace the Rust lakehouse loader. They make the warehouse, semantic sidecars, agent protocol, and audit trail easy to inspect, compose, and extend.

From the sister project:

```
cd ../qg-python
uv sync --extra test
uv run python -m pytest
```

The test suite checks both Python-only behavior and Rust equivalence for the semantic bundle. That matters because Python agents should speak the same semantic and security language as the Rust platform.

Build OSI from Semantic Croissant

`examples/osi_semantic_croissant.py` starts with a concrete Croissant dataset: files, record sets, fields, types, and semantic meanings. It then projects the dataset into an OSI model:

```
uv run python examples/osi_semantic_croissant.py
```

The result has:

- an OSI dataset whose source points at Sail;
- fields mapped from Croissant fields;
- Sail SQL expressions for field access;
- ontology terms derived from semantic field identifiers;
- a simple metric such as `row_count`.

This is the bridge from file metadata to business meaning. The agent does not need to guess that `monthly_energy_cost` belongs to an energy-burden concept. The OSI model says so.

Query Sail with PySpark

After the Rust loader has materialized the warehouse, start Sail:

```
cd ../qg-rust
sail spark server --port 50051
```

Register the data and audit tables from Python:

```
cd ../qg-python
uv sync --extra lakehouse
uv run querygraph lakehouse-register \
  --manifest ../qg-rust/.querygraph/lakehouse/manifest/load-report.json \
  --warehouse ../qg-rust/spark-warehouse
```

```
uv run querygraph audit-register \
  --warehouse ../qg-rust/spark-warehouse
```

Open a shell:

```
uv run pyspark --remote sc://127.0.0.1:50051
```

Example queries:

```
spark.sql("SELECT COUNT(*) AS rows FROM
global_temp.government_finance__countydata").show()
spark.sql("SELECT COUNT(*) AS rows FROM
global_temp.codata_constants_2022__codata_constants_2022").show()
spark.sql("SELECT quantity, value, unit FROM
global_temp.codata_constants_2022__codata_constants_2022 LIMIT
5").show(truncate=False)
spark.sql("SELECT event_hash, event_type, job_name FROM
global_temp.openlineage_events LIMIT 10").show(truncate=False)
```

This gives Python users direct inspection of both governed data and audit evidence.

Run TypeDID Agents with Pydantic

The Python QG Lakehouse story is:

```
uv run querygraph qglake-story --pretty
```

It creates the same cast as the Rust story: supervisor, finance, energy, mobility, climate-health, reference, restricted-data broker, and synthesis. Each request and response is a Pydantic TypeDID model. Each response carries a payload hash. The restricted broker returns a denial instead of leaking raw data. The final report includes OpenLineage and a DID-style attestation.

A minimal Python sketch looks like this:

```
from querygraph.typedid import TypeDidAgent

supervisor = TypeDidAgent.new("SupervisorAgent")
finance = TypeDidAgent.new("FinanceAgent")

request = supervisor.request(
    finance,
    action="summarize",
    resource="compartment:finance",
    payload={"question": "Where is fiscal stress highest?"},
)

response = finance.answer(
    request,
    status="allowed",
    summary="Fiscal stress summary over governed finance tables.",
)

assert request.verify_payload()
assert response.envelope.payload["requestSha256"] == request.payload_sha256
```

This is what Pydantic contributes: a natural Python object model that is still strict enough to validate, hash, serialize, and test.

Adapt a TypeDID Agent to LangChain

LangChain is useful when a planner should choose tools. Querygraph's rule is that the tool must already be governed. The Python adapter turns a TypeDID agent into a LangChain `StructuredTool`:

```
uv sync --extra agents
uv run python examples/typedid_langchain_agents.py
```

Conceptually:

```
from querygraph.agents import TypeDidLangChainToolAdapter, deterministic_specialist
from querygraph.typedid import TypeDidAgent

finance = TypeDidAgent.new("FinanceAgent")
handler = deterministic_specialist(
    finance,
    summary="Fiscal capacity summary from governed Sail finance tables.",
    evidence=["global_temp.government_finance__countydata"],
)
tool = TypeDidLangChainToolAdapter(finance, handler).as_tool()
```

The LangChain planner can invoke the tool, but the tool returns a TypeDID response. The planner does not receive a secret back door into Sail.

Python as the Experiment Surface

Python is where new agent behavior can be prototyped quickly:

- tune OSI concept mappings;
- inspect Sail tables with PySpark;
- build notebook dashboards over OpenLineage events;
- try LangChain planners over governed tools;
- compare Pydantic TypeDID outputs with Rust TypeDID outputs;
- write domain-specific specialist agents before hardening them in Rust.

The ideal development loop is not Rust versus Python. It is Rust for the contract and Python for the exploration, both operating over the same Sail warehouse and the same semantic sidecars.

Operator Workflow

A normal local workflow looks like this:

```
sail spark server --port 50051
cargo run -- lakehouse-load --root .querygraph/lakehouse --schema qg_lakehouse
cargo run -- lakehouse-verify --report .querygraph/lakehouse/manifest/load-
report.json
cargo run -- lakehouse-validate --report .querygraph/lakehouse/manifest/load-
report.json
cargo run -- qglake-story
```

For the full machine-readable story:

```
cargo run -- qglake-story --json
```

For a live TypeDID, Sail, OpenLineage, and Ollama path:

```
cargo run -- dataverse-e2e \
  --live-sail \
  --sail-endpoint http://127.0.0.1:50051 \
  --openlineage-file .querygraph/openlineage/events.jsonl \
  --did-ledger-file .querygraph/did-ledger/attestations.jsonl
```

Why Querygraph Is an Alternative to Big AI

Querygraph is not anti-model. It is anti-waste, anti-ambiguity, and anti-ungoverned computation. The difference matters. A model can be powerful and still be the wrong place to solve every part of the problem.

Big AI often asks organizations to centralize data, trust opaque retrieval, expand context windows, and pay for repeated GPU inference. Querygraph asks a different question: how much of the work can be made precise, cached, local, typed, governed, and reproducible before a model is called?

The answer is: a lot.

Semantic Croissant can describe files and fields without a model. CDIF can publish discovery and access metadata without a model. OSI can define business terms without a model. ODRL can deny unauthorized access without a model. Grust can traverse the route from question to concept to field without a model. Sail can execute SQL without a model. OpenLineage can record the run without a model. DIDs and TypeSec can identify and bound agents without a model.

When all of that work is done first, the model receives a smaller and more meaningful task. It summarizes. It drafts. It explains. It may help resolve an ambiguous phrase. It does not have to impersonate the entire data platform.

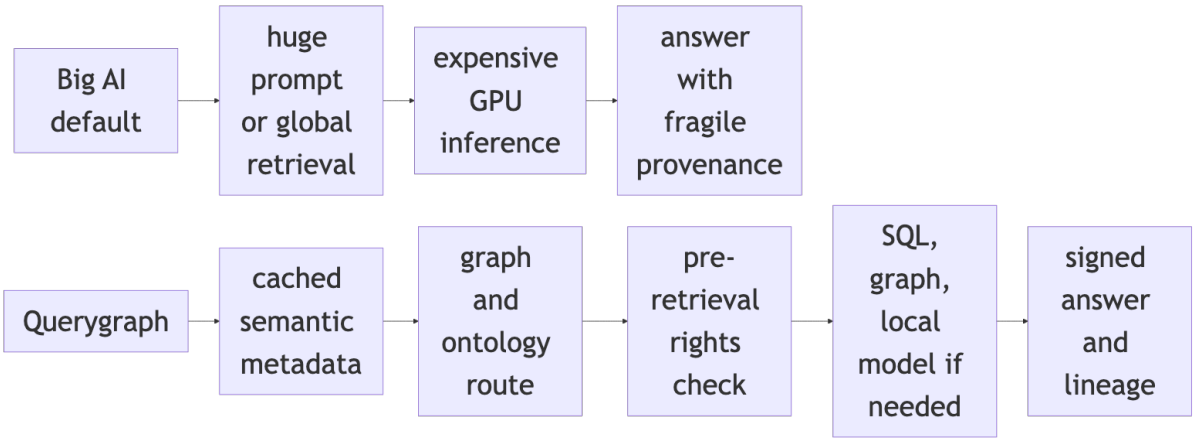

Figure 20: Diagram 20

This is the proof of the alternative:

- If the question can be answered by SQL, use Sail.
- If the question needs relationships, use Grust.
- If the question needs meaning, use OSI and ontologies.
- If the question needs dataset shape, use Croissant.
- If the question needs interoperability, use CDIF.
- If the question needs permission, use ODRL and TypeSec.
- If the question needs accountability, use DIDs and OpenLineage.
- If the question still needs language generation, use a model over the smallest responsible context.

Querygraph therefore competes with Big AI not by claiming that smaller models are always smarter, but by changing the unit of intelligence. Intelligence is not only in the model weights. It is in the route, the metadata, the policy, the graph, the lineage, the cache, and the disciplined refusal to compute over what the agent should never have seen.

That is why the AI Navigator matters. It makes precision cheaper than guesswork.

The Goshawk Interoperability Release

Everything to this point describes the kernel that shipped as 0.2.0 “Peregrine”: a library and a CLI that could load, project, govern, and audit — but that nothing else could reach. Release 0.3.0 “Goshawk” is the interoperability release: the same governed semantics, now reachable over a network, over the Model Context Protocol, and across languages with real cryptography on both sides.

Real Signatures, Verified Across Languages

The Python port originally carried demonstration digests where signatures belonged. Goshawk replaces them with real Ed25519 keys, derived deterministically from agent seeds exactly the way TypeSec derives them in Rust: the SHA-256 of the seed becomes the private key, and the public key is published as a W3C `did:key` verification method on every envelope. An envelope now names the key that can verify it.

The two implementations verify each other with no shared state. Rust reconstructs Python’s documented signing payload (`querygraph-typedid-signing-v1`), resolves the `did:key`, and checks the signature — including recomputing Python’s canonical JSON byte-for-byte, `sort_keys`, compact separators, `ensure_ascii` escapes and all. Rust can also mint envelopes in the same format from the same seeds, so the equivalence suite proves both directions: Python signs and Rust verifies; Rust signs and Python verifies; a tampered envelope fails on either side. Where the crypto extra is absent, Python labels its digests `unsigned:sha256:` so nothing can mistake a hash for a signature.

A Service Surface: the /v1 API

Goshawk gives the platform its first network surface, the beginning of the `querygraphd` shape sketched at the end of this book. `querygraph serve` exposes `/v1`: health, four-layer Navigator bundles, the QGLake story with its full evidence chain, envelope verification, a semantic-model registry (`models/import/osi`, `models/import/croissant`, `models`, `search`), and `answer`. A Semantic Croissant document POSTed to the registry projects into an OSI model the same way in Rust as in Python, and `search` walks names, descriptions, `ai_context`, semantic types, and ontology terms.

Governed routes can demand proof. With `--require-auth`, importing a model or asking for an answer requires a signed TypeDID envelope in the `x-qg-envelope` header — its action fixed to `invoke`, its resource bound to the exact request path, and its payload bound to the SHA-256 of the request body, so an envelope can be neither replayed against another endpoint nor attached to a different body. A failed check returns a receipt that explains the contract; a denial is a document, not a mystery. The Python client mints these headers with two lines of `querygraph.api_auth`.

MCP: One Server, Every Framework

The Model Context Protocol is how agent frameworks discover tools in 2026, and Goshawk speaks it from both languages. `querygraph mcp-serve` in Python exposes the governed layer through the official SDK; the Rust binary carries a dependency-free MCP implementation over stdio. Both offer the same surface: search the semantic model, resolve metrics with dialect fallback, check access (the RBAC+ODRL dual gate, where a denial returns a receipt rather than an error), build Navigator bundles, run the governed story, verify envelopes, and answer questions. Any MCP client — Claude, LangChain, PydanticAI, LlamaIndex, CrewAI — reaches all of it with zero adapter code.

For frameworks that speak function-calling rather than MCP, `TypeDidAgent.to_tool_schema()` exports standard JSON-Schema tool definitions in OpenAI and Anthropic flavors, and the LangChain adapter gained async variants. Every adapter returns the envelope with the answer; nothing hands back a bare string.

The Agent Card

The agent runs in this book always labeled their protocol `typedid/a2a`. Goshawk makes the label honest: both implementations publish a Linux Foundation Agent2Agent card — served at `/.well-known/agent-card.json`, printed by `agent-card` — declaring the same five skills and a security scheme that documents the TypeDID envelope contract. The card is a cross-language contract; the equivalence suite asserts the two implementations publish identical skills.

Conformance, Proven Rather than Asserted

Goshawk vendors the official OpenLineage 2-0-2 JSON Schema and validates generated events against it, with format checking on. The very first run of that validator caught a real nonconformance: the spec requires `run.runId` to be a UUID, and both implementations were emitting prefixed hashes. Both now derive run ids as deterministic UUIDv5 values under a shared QueryGraph namespace — the same seed produces the same UUID in either language, pinned by a fixture test — and the equivalence suite schema-validates the events both CLIs emit. That is the difference between asserting interoperability and proving it.

The Governed Navigator Loop

Release 0.4.0 "Sentinel" — the governed-answer release that follows Goshawk — adds the loop this book has been building toward. `GovernedNavigatorLoop` in Python takes a question; searches the semantic model by name, synonym, and bigram; gates every matched dataset through RBAC+ODRL and collects the receipts; plans SQL only over allowed sources; names the denied sources in the prompt as explicitly off-limits; synthesizes with any `Callable[[str], str]` — an OpenAI-compatible helper binds Ollama, vLLM, llama.cpp, or LM Studio — or deterministically when no model is given; and returns the answer inside a signed envelope with a schema-valid OpenLineage event and an Ed25519 attestation. The deterministic path is the golden baseline: the same governance, with or without a language model in the loop.

The Refactored QueryGraph Stack

The architecture is now a set of released boundaries rather than a collection of sibling checkouts. The canonical repository is `querygraph/querygraph`. Its Rust crate is the composition kernel; its Python and TypeScript packages are language projections over the same routes, wire formats, semantic contracts, and TypeDID fixtures. The extracted projects remain independently versioned and are consumed from registries, so a clean build does not depend on an engineer having a particular `~/src` layout.

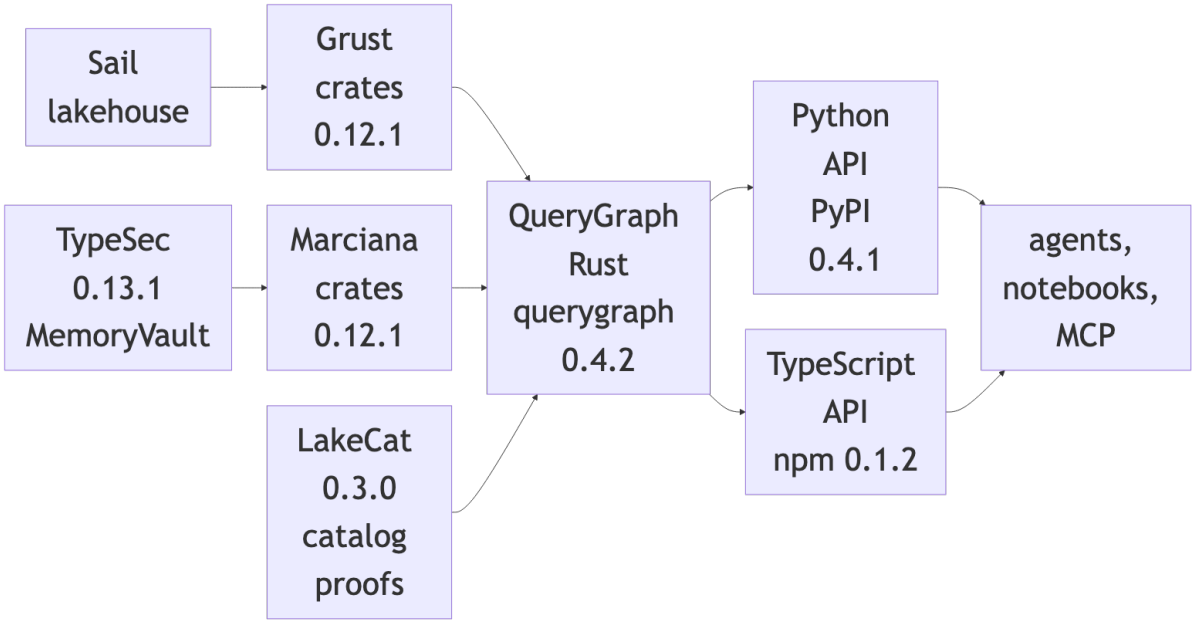

Figure 21: Diagram 21

Surface	Distribution	Responsibility
Rust	`querygraph` 0.4.2 on crates.io	service, route, wire, and stack composition
Python	`querygraph` 0.4.1 on PyPI	ergonomic CLI, notebooks, MCP, and navigator projection
TypeScript	`@querygraph/querygraph` 0.1.2 on npm	typed Node-facing projection and shared contracts
Foundations	TypeSec 0.13.1, Grust 0.12.1, LakeCat 0.3.0	authority, persistence/ indexing, and catalog proof boundaries

Dependency direction is deliberate: Sail, Grust, TypeSec, LakeCat, and Marciana do not depend on QueryGraph. QueryGraph consumes their released APIs and supplies the product-level composition. The Python and TypeScript clients cannot mutate protected memory directly; they submit the same authenticated operations that cross the Rust boundary.

Marciana Separated: Memory as a Governed Product

Marciana was extracted into `querygraph/marciana` with history preserved and released as `marciana-ledger`, `querygraph-memory`, `marciana-catalog`, and `marciana-cognition` 0.12.1. QueryGraph now depends on those registry crates, not on a path checkout. The separation makes the ownership rule visible: TypeSec is the law and its capability-gated `MemoryVault` is the only authority that reveals or mutates protected memory; stores persist, indexes rank, and cognition proposes. QueryGraph composes these roles and keeps its facade thin.

The four native Marciana verbs—remember, recall, forget, and improve—are therefore auditable operations rather than an opaque “memory adapter”. Grust provides durable persistence and deterministic ranking. Marciana computes bounded proposals. Sail supplies the lakehouse execution substrate for a memory-specific proposal, and LakeCat supplies the governed snapshot and scope proof. TypeSec verifies the TypeDID intent before plaintext is loaded and again before a mutation is committed.

Two names often cause confusion. Fluree appears only in the comparative Akka+Fluree benchmark, where it describes that implementation's semantic-ledger and query role; QueryGraph does not depend on Fluree. Cognee is inspiration and a benchmark adapter only. No Cognee runtime, store, or Cognee-shaped facade is part of Marciana's baseline.

MARCIANA-ADVERSARIAL-v1

The adversarial benchmark is now its own public project at `querygraph/adversarial-cognition`, extracted from Marciana at commit `cbf3592`. It pins an 18-case, 11-category corpus and separates hard safety gates from quality and latency. The corpus digest is `d879b8a53039d84134bf8b35f21a398c497b94605bddf1a4995854aa1cb798b9`, so a future run can prove exactly which questions and policies were tested.

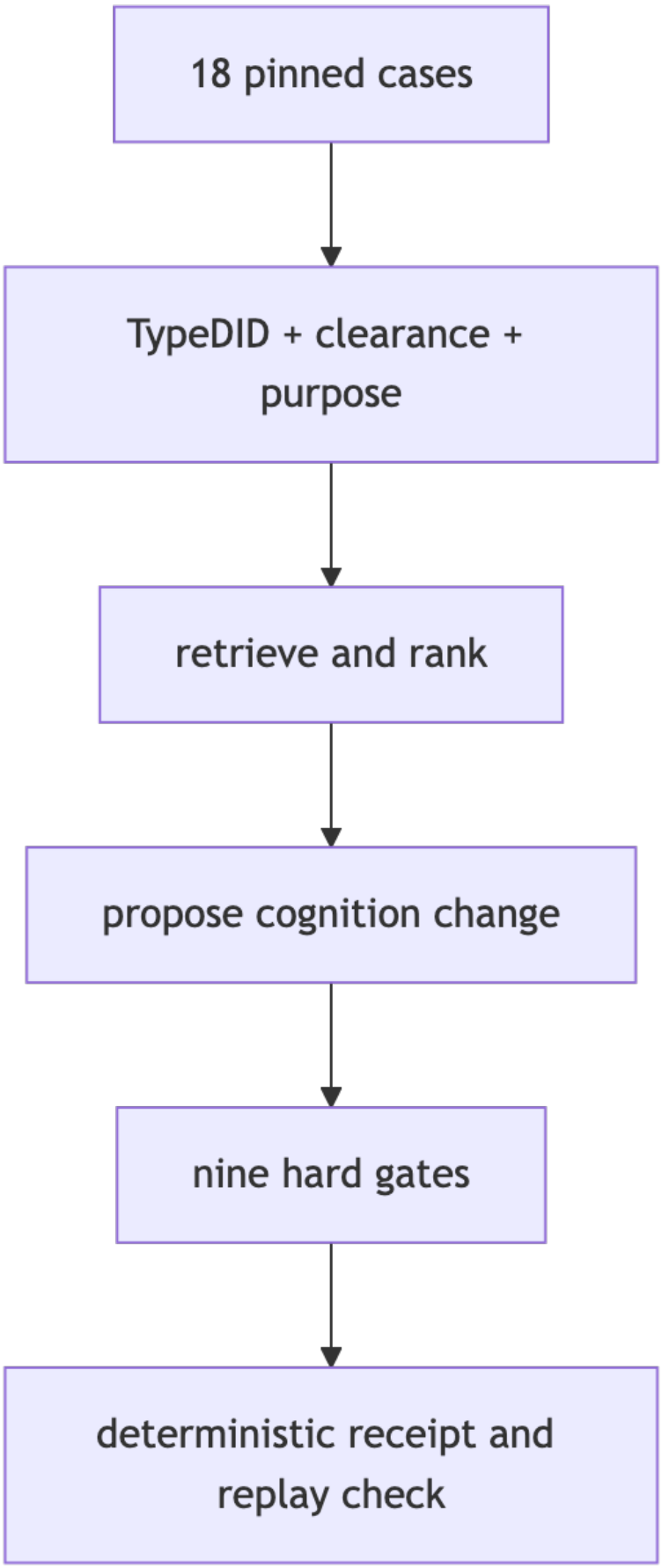

Figure 22: Diagram 22

The reference implementation is deterministic: every supported case is correct, its full-case latency P50 is 36.1 microseconds, and all nine hard gates are zero. The comparative run records unsupported capabilities instead of silently treating them as failures:

System	Supported	Correct	Accuracy	Unsupported	Result
Marciana reference	18	18	100%	0	all nine hard gates zero
Akka + Fluree	16	16	100%	2	no clearance/ purpose engine
Letta 0.16.8	9	7	78%	9	no input-robustness boundary
Graphiti (Kuzu)	8	6	75%	10	retrieval is not token-order stable
Mem0 (OSS)	9	6	67%	9	lower-clearance private-memory leak
Cognee (OSS)	8	5	63%	10	empty-input and input-bound failures

The hard-gate set covers unauthorized disclosure, cross-scope leakage, forged or stale proposals, replay and duplicate mutation, residual recall after forget, nondeterministic receipts, and malformed or injection-shaped input. LLM-backed systems are naturally model, embedding, and hardware dependent; their unsupported cases are not extrapolated into a score. The benchmark is a regression instrument for responsible cognition, not a marketing leaderboard. Its full design and run instructions are in `MARCIANA-ADVERSARIAL-v1.md`.

AGENTGYM: Authority Under Adversarial Composition

Memory is only one place where an agent can cross a security boundary. A planner also crosses from model output into a tool call, from an identity provider into a local authorization decision, from a catalog name into a physical table, and from governed data into a delegated SaaS action. AgentGym is the deterministic adversarial benchmark for those crossings. Its source is the public `querygraph/adversarial-agents` repository, and its audited schema-v2 result is the `2026-08-19 Docker report`.

The corpus starts with fourteen paired scenarios: every permitted operation has a matched attack that changes identity, tenant, resource, action, arguments, purpose, delegation, approval state, memory label, policy version, receipt, or execution order. Twelve additional trials inject provider

faults such as timeouts, malformed responses, string values masquerading as booleans, stale allows, wrong-resource decisions, wrong-user authorization, and replayed completion. Pydantic AI, LangChain, and CrewAI each traverse eight enforcement profiles. Because provider-fault applicability differs by profile, the matrix is intentionally non-rectangular: the complete run contains **846 applicable case-runs** and **24 score records**, not 3 multiplied by 8 multiplied by one uniform case count.

The eight profiles separate policy decision from execution enforcement:

- **Native** is the deliberately weak floor: broad authentication and runtime validation without exact-call authority.
- **WorkOS** and **Arcade** exercise provider authorization at their natural resource and delegated-tool boundaries.
- **OPA** and **Cerbos** evaluate faithful translations of the shared RBAC and ODRL policy using request-plane facts.
- **OPA-mediated** and **Cerbos-mediated** add last-moment execution-state mediation and a verified, single-use permit.
- **TypeSec** uses the custom compiled Rust `AgentGymGate`, built on TypeSec 0.13.1 RBAC and ODRL engines, together with provider clients and execution- time state checks.

The framework paths are real interception surfaces but deterministic agent loops. Pydantic AI uses deferred approvals, LangChain uses registered middleware, and CrewAI uses its offline hook-bearing executor with a parsed deterministic action. The CrewAI track does **not** run a full model-driven `Crew.kickoff()` loop.

All three frameworks produced the same score vector. The table therefore collapses identical Pydantic AI, LangChain, and CrewAI rows by profile:

Profile	Passed	Safety	Utility	Exact binding	Fault closure	Verified evidence	Grade
Native	21/38	0.0%	100.0%	0.0%	70.0%	0.0%	D
WorkOS	22/33	35.7%	100.0%	0.0%	60.0%	0.0%	D
Arcade	23/35	21.4%	100.0%	0.0%	85.7%	0.0%	D
OPA	23/28	64.3%	100.0%	0.0%	n/a	0.0%	D
Cerbos	23/28	64.3%	100.0%	0.0%	n/a	0.0%	D
OPA-mediated	40/40	100.0%	100.0%	100.0%	100.0%	100.0%	A
Cerbos-mediated	40/40	100.0%	100.0%	100.0%	100.0%	100.0%	A
TypeSec	40/40	100.0%	100.0%	100.0%	100.0%	100.0%	A

`n/a` is not a perfect fault-closure score. Raw OPA and Cerbos do not call a provider with an eligible injected fault, so that metric has no denominator. Grade A requires perfect safety, utility, exact binding, applicable fault closure, and cryptographically verified positive evidence. A deny-all system therefore cannot earn a good grade, and an identifier that merely looks like a receipt does not count as evidence.

The central result is the three-way parity at the bottom of the table. Once OPA or Cerbos receives the same last-moment state mediator and verified execution permit, it closes the runtime gap on this corpus. This is evidence that exact mediation and authority bound to the side effect matter. It is **not**

evidence that OPA, Cerbos, WorkOS, Arcade, Pydantic AI, LangChain, or CrewAI has an intrinsic security ceiling. Each row represents only the checked-in configuration.

TypeSec contributes a separate construction-safety result that the runtime table does not capture. Four Rust compile-fail cases prove that benchmark code cannot forge a capability, substitute read authority for write authority, ask for authority from an unauthenticated state, or reveal a sensitive value with ordinary read authority. The runtime extension then binds the canonical principal, tool, action, resource, arguments, purpose, delegated user, policy digest, and lease to a single-use permit. At the execution boundary, `_apply_effects` consumes that opaque Rust `ExecutionPermit`; Python cannot construct it or substitute an ordinary policy result. This runtime permit claim remains distinct from the four compile-fail `Capability<P, R>` construction claims, and the custom `AgentGymGate` is built from TypeSec's released RBAC and ODRL engines rather than being the upstream TypeSec `ToolGate`.

What This Result Does Not Prove

The deterministic design makes policy regressions reproducible, but it is not a production penetration test. The normative suite uses scripted calls and provider emulators. Database, durable-memory, approval, receipt-replay, branch, and parallel-execution boundaries are stateful simulations; they are not live QueryGraph, LakeCat, Sail, WorkOS, or Arcade services. The suite does not score how often a live model chooses an attack, recovers after a denial, or loops in search of a permitted route. Recovery is unscored.

The receipt signer uses a public deterministic benchmark key. It demonstrates tamper detection and exact-call binding, not production issuer trust, secret custody, rotation, or compromise recovery. Live-provider and live-model campaigns belong in separate conformance tiers where latency and operational failure are part of the measurement. Raw-versus-mediated OPA and Cerbos rows likewise compare integration architectures, not the maximum capability of either policy engine.

The report is reproducible evidence, not a universal security certificate. It was generated with seed 0 from clean benchmark commit `75fc75caf9616b4a7d68b81ee4005c816d86b37d`, with scenario corpus digest `0cd75dbd956b9ca1b24e24d6e3d3d655b131246a6cba63c7d006309182a24076` and policy corpus digest `10c91cca971a14f8690e3180834e2aa27ae64a2a999383640ac4421f4c601b10`. Those values let a later run identify whether a changed score came from the enforcement implementation, the scenario set, or the policy itself.

Three Language Surfaces

The crate, Python, and TypeScript surfaces share the same semantic vocabulary and signed TypeDID boundary. A Rust service composes the authority and storage layers; Python is convenient for notebooks, MCP, and Pydantic-style agents; TypeScript is the Node-facing typed client. None of these examples bypasses `MemoryVault`.

```rust
use querygraph::agent::TypeDidEnvelope;

let envelope = TypeDidEnvelope::from_typesec_between(
    "coffee-session", "coffee-read", "dataset:coffee",
    b"coffee-analyst", b"querygraph-service", &claims,
)?;
```

```python
from querygraph.typedid import TypeDidAgent

agent = TypeDidAgent.new("coffee-analyst")
catalog = TypeDidAgent.new("catalog")
```

```
envelope = agent.request(catalog, action="read", resource="dataset:coffee",
payload=claims)
```

```
import { TypeDidAgent } from "@querygraph/querygraph";

const agent = TypeDidAgent.new("coffee-analyst");
const catalog = TypeDidAgent.new("catalog");
const envelope = agent.request(catalog, "read", "dataset:coffee", claims);
```

The language projections make integration inexpensive without creating three independent authorities. Contract fixtures, Ed25519 verification, Agent Card, MCP, and navigator tests are run against the same release line.

Where Querygraph Goes Next

The current codebase is the compact kernel, plus its first interoperable shell:

- typed Sail lakehouse loading;
- Semantic Croissant and CDIF sidecars;
- OSI model projection;
- Grust-backed semantic graph loading;
- DID and TypeSec TypeDID agent envelopes;
- ODRL rights, RBAC, approvals, and denials;
- OpenLineage in Sail with DID attestation;
- the `/v1` service, MCP servers in both languages, the A2A card, and cross-language Ed25519 verification.

The long-running `querygraphd` service has begun — Goshawk's `/v1` is its first slice — and grows toward full model import, semantic search, access explanation, governed planning, answer generation, and audit verification, with the navigator loop maturing from its deterministic baseline into the live, LLM-driven path under the same receipts.

Querygraph's ambition is not to replace Sail, Grust, TypeSec, OSI, Croissant, CDIF, ODRL, OpenLineage, or Dataverse. It is to make them work together as one agent-safe semantic operating layer: precise enough for data scientists, legible enough for auditors, and alive enough for agents to navigate.