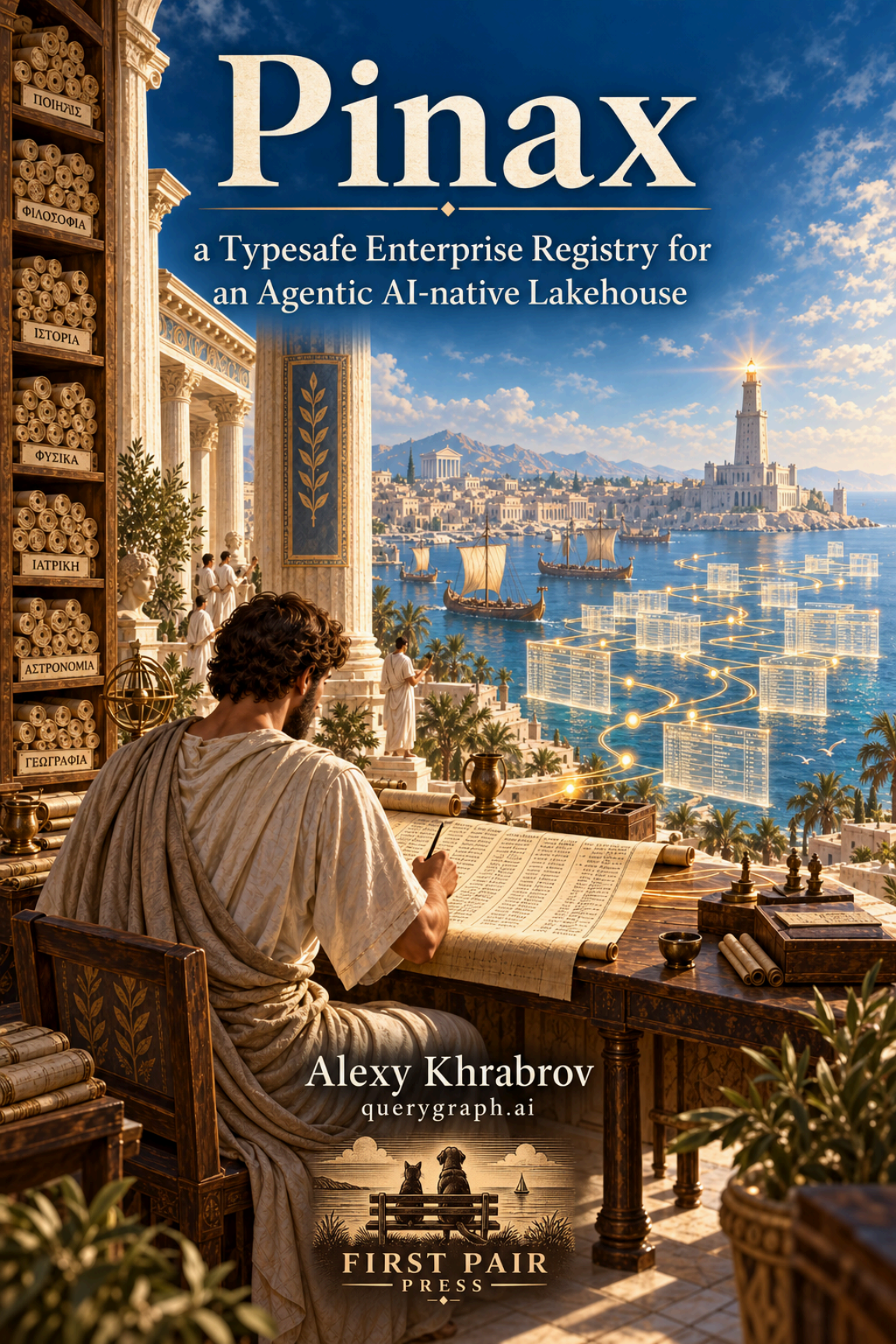

Pinax
a Typesafe Enterprise Registry for an Agentic AI-native Lakehouse
ΠΟΙΗΤΑΙ
ΦΙΛΟΣΟΦΙΑ
ΙΣΤΟΡΙΑ
ΦΥΣΙΚΑ
ΙΑΤΡΙΚΗ
ΑΣΤΡΟΝΟΜΙΑ
ΓΕΩΓΡΑΦΙΑ
Alexy Khrabrov
querygraph.ai
FIRST PAIR
PRESS

Pinax

A Typesafe Enterprise Registry for an Agentic AI-native Lakehouse

Alexy Khrabrov

2026-09-13

Contents

1 A registry that an agent can trust 2
1.1 Who this book is for 3
2 The QueryGraph stack 3
2.1 Logical and physical identity 4
2.2 The minimum trust boundary 4
3 Anatomy of a table contract 4
3.1 Stable field IDs 5
3.2 Physical types 6
3.3 Stewardship and protection 6
4 Validation and immutable Rust values 6
4.1 Bounded input 7
4.2 Normalization and digests 7
5 Profiles and the authoring workflow 8
6 Compatible revisions 9
6.1 A concrete revision 9
6.2 Whole-registry identity 9
7 Catalog projection and reconciliation 10
7.1 Creation versus existing tables 10
7.2 Reconciliation states 10
8 Governed discovery and planning 11
8.1 Discovery 11
8.2 A bounded scan intent 11
8.3 Planning checks 12
9 Execution and evidence 12
9.1 Buffering before release 12
9.2 What the evidence establishes 13
10 MCP sessions and credentials 13

10.1 Two credential boundaries 13
10.2 Cancellation and overload 14
11 A complete demonstration 14
11.1 The row experiment 14
11.2 Changes during a real read 15
11.3 Reading a report honestly 15
12 Operating a registry 15
12.1 Failure diagnosis 16
12.2 Release discipline 16
13 The next contract 17
14 Source guide 17

1 A registry that an agent can trust

An enterprise lakehouse can contain thousands of tables and still be difficult to use. A catalog may identify a table's location and current schema without explaining what a column means, who maintains it, which revision an application expects, or whether a particular agent may read it for a particular purpose. An assistant that guesses these answers can produce a plausible result from the wrong data.

Pinax makes those expectations explicit. It is a Rust library and a versioned document contract for enterprise tables. A registry records logical names, stable field identities, physical types, business semantics, stewardship, classification and policy bindings. Validation turns a proposed document into an immutable registry. Compatibility checks compare that registry with its predecessor. The trusted host then uses the same contract during discovery, planning and execution.

The cover depicts a cataloger's work continuing into a connected landscape of records. That continuity is the book's organizing idea. Useful knowledge needs an inventory, but an inventory also needs accountable interpretation. In an agentic lakehouse, the inventory must survive a change of schema, a change of policy and the interval between planning a query and returning its rows.

This book is a practical guide to the `pinax.v1` reference contract and its QueryGraph integration. It explains what the components enforce, where the trust boundaries sit and how to demonstrate the complete request path. The integration examples distinguish prepared source builds from published

crate releases. A successful fixture is evidence for the behavior it exercises; it does not turn an unreleased extension into a released API.

1.1 Who this book is for

Data stewards can use Pinax to review table contracts. Rust developers can embed its validated types without adopting a new transport. Platform engineers can connect a registry to the existing catalog and policy system. People building assistants can use its MCP tools to propose changes and request bounded operations without receiving catalog credentials or arbitrary storage access.

The examples use a fictional enterprise named `acme`. Its identities, purposes and sample records are demonstration data. Real deployments supply their own identity roots, policy engines and lifecycle controls.

2 The QueryGraph stack

Pinax joins a family of components with different responsibilities. Keeping those responsibilities separate makes the security story inspectable.

Component	Responsibility in the stack
Grust	Property graphs and graph query infrastructure
TypeSec	Cryptographic identity, policy evaluation and authorization primitives
Marciana	Semantic catalog and cognitive workflow components
LakeCat	Iceberg catalog ownership, governed operations and execution evidence
Sail	Table-format interpretation, scan planning and execution
Pinax	Validated enterprise table contracts and compatibility
QueryGraph	Semantic and agent workflows, MCP transport and trusted service composition

A semantic model helps an assistant understand a question. A Pinax contract identifies the governed table and columns available to answer it. LakeCat owns the physical table and the authority to execute. Sail interprets that table's metadata and reads the selected snapshot. TypeSec supplies policy decisions at the boundaries where the request needs them.

No single metadata document replaces the other components. A classification label cannot authenticate an agent. A graph edge cannot authorize a storage read. A valid registry cannot prove that the current Iceberg metadata still matches it. Each component contributes a check whose inputs and result remain visible to the next boundary.

2.1 Logical and physical identity

Pinax identifies a table by enterprise and logical name. A trusted `CatalogBinding` selects the warehouse and maps that identity to a LakeCat namespace and table. For `acme.customers` in warehouse `production`, the enterprise becomes the namespace and `customers` remains the table name.

The requesting agent cannot substitute another warehouse, a metadata URL or an object-store path. Those choices belong to the deployment. This prevents a request from presenting an attractive logical contract while redirecting the reader to unrelated physical data.

2.2 The minimum trust boundary

The host owns the immutable registry, deployed policy identity and revision, policy engine, authenticated principal, catalog origin and backend credentials. The agent owns a proposed document or a bounded intent. Descriptions inside registry documents remain data. An instruction hidden in a column description does not change the host's policy or execution rules.

3 Anatomy of a table contract

A registry document contains `version`, `enterprise` and `tables`. Its version is exactly `pinax.v1`. Each table has a positive revision, an explicit profile, metadata, security declarations, ordered columns and a primary key.

This small custom table is a complete authoring example:

```
{
  "version": "pinax.v1",
  "enterprise": "acme",
  "tables": [{
    "name": "customers",
```

```
    "revision": 1,
    "profile": "custom",
    "metadata": {
      "owner": "platform",
      "steward": "data",
      "description": "Customer identifiers",
      "retention_days": 30
    },
    "security": {
      "policy": "enterprise",
      "revision": 1,
      "classification": "internal",
      "purposes": ["analytics"]
    },
    "columns": [{
      "id": 1,
      "name": "customer_id",
      "data_type": {"kind": "string"},
      "nullability": "required",
      "semantic": "customer.id",
      "description": "Customer identifier",
      "classification": "internal"
    }],
    "primary_key": [1]
  }]
}
```

Save it as `customers.json`. The explicit `kind` field describes a domain alternative. Unknown alternatives and unknown object fields fail validation. A misspelled protection is therefore an error rather than an ignored comment.

3.1 Stable field IDs

Field IDs identify columns across revisions. They are positive, strictly increasing and unique within the contract. Column names are also unique. The primary key refers to required fields by ID, so a key cannot quietly point to a different field after an edit.

The contract preserves column order. An existing field's name, type, nullability, semantic term, description and classification cannot change in a compatible successor. This rule is intentionally stricter than simply asking whether an old SQL query still parses.

3.2 Physical types

Pinax type	Meaning at the Iceberg boundary
`boolean`	Boolean value
`int32`, `int64`	Signed integer of the stated width
`float64`	Double-precision floating point
`string`, `binary`	Text or bytes
`date`	Calendar date
`timestamp_utc`	Timestamp with time zone, microsecond precision
`decimal`	Exact decimal with explicit precision and scale

Decimal precision ranges from 1 through 38. Scale ranges from zero through the selected precision. A financial amount uses an exact decimal together with a currency field. Neither a decimal type nor a currency column decides the enterprise's accounting conventions.

Nested structures, generated SDKs and relationship constraints are outside the v1 registry contract. A consumer must reject unsupported requirements instead of silently dropping them.

3.3 Stewardship and protection

Owner and steward identify responsibility. The description supplies human meaning. Retention days declare a lifecycle expectation. The security section binds the table to an explicit policy identity and revision, a sensitivity floor and allowed purposes.

Classification increases through `public`, `internal`, `confidential` and `restricted`. Every column must meet its table's floor. Classification does not grant access. TypeSec still evaluates the authenticated principal, and Pinax still checks the declared purpose.

Retention is also a declaration. Pinax does not delete stored rows when the declared number of days elapses. Ingestion and lifecycle systems must enforce that requirement, just as they must enforce primary-key uniqueness and reference integrity in the actual data.

4 Validation and immutable Rust values

Authoring values are mutable data-transfer objects. The trusted registry is immutable and can be created only through validation.

`Registry::from_json` parses a document and applies the same domain rules as `Registry::new`.

```rust
use pinax::{PinaxError, Registry};

fn registry_digest(document: &[u8]) -> Result<String, PinaxError>
{
    let registry = Registry::from_json(document)?;
    registry.digest()
}
```

Use Rust edition 2024 for applications and examples in this stack. The library's typed constructors make a stronger boundary than a convention that every caller remembers to invoke a separate validation function after parsing.

4.1 Bounded input

The registry JSON limit is 8 MiB. A snapshot contains at most 10,000 tables, and a table contains between 1 and 1,024 columns. A table may declare at most 64 purposes. Required text must be nonblank, free of control characters and no longer than 4,096 UTF-8 bytes.

Identifier syntax is deliberately narrow: `[a-z][a-z0-9_]{0,62}`. Enterprise, table, column, warehouse and purpose names use this form. A registry can begin empty, but a populated table cannot omit the structural and governance information its contract requires.

These limits make validation predictable. They are not a replacement for transport limits, process memory limits or request deadlines. QueryGraph adds its own limits at the MCP and execution boundaries.

4.2 Normalization and digests

Pinax sorts tables and purpose sets before encoding. Column order and primary-key order remain significant. The reference digest hashes the `pinax.v1` domain, one zero byte and compact Serde JSON of the normalized typed registry. It returns lowercase hexadecimal prefixed with `sha256:`.

This is a versioned Rust reference encoding, not a claim of RFC 8785 canonicalization. Input whitespace and JSON object-key order do not change the normalized digest. A different registry does.

A digest identifies content. It does not identify the person who approved that content. Publisher authentication and review remain separate controls.

5 Profiles and the authoring workflow

Pinax provides HR, customer and transaction profiles as useful starting contracts. They are enterprise conventions, not universal models of every organization.

Profile	Required fields
HR	Employee ID, legal name, department ID and employment start date
Customer	Customer ID, display name and creation timestamp
Transaction	Transaction ID, customer ID, amount, currency and occurrence timestamp

The executable `standard_columns` function supplies the exact field IDs, types, semantics and nullability. Standard fields are required and at least confidential. The first field is the primary key. Extensions can add columns while preserving the standard's required meaning.

From a Pinax source checkout, generate and validate an initial registry:

```
cargo run --locked -- init --enterprise acme \
  --owner data_platform --steward enterprise_data \
  --policy enterprise > registry.json
cargo run --locked -- validate --registry registry.json
cargo run --locked -- check \
  --current registry.json --next registry.json
cargo run --locked -- catalog-plan \
  --registry registry.json --warehouse production
```

Initialization chooses an illustrative 365-day retention declaration and the `analytics` purpose. Review those values before deployment. The commands write machine-readable JSON and return a nonzero status on failure. They do not publish the registry, create a namespace or grant an agent access.

QueryGraph exposes the same authoring commands under its `pinax` subcommand. Its `validate_pinax` MCP tool accepts a JSON-string `document` and optional `previous` document. An assistant can iterate on a proposal without obtaining publication privileges.

6 Compatible revisions

An independently valid document may still be an invalid successor. Publication must check both properties. `check_successor` compares the proposed snapshot with the previous validated registry.

An identical table keeps its revision. A changed table advances exactly one revision. New tables begin at revision one. Existing tables cannot disappear, and the enterprise identity cannot change.

For an existing table, a compatible change appends optional columns with fresh, greater IDs. Existing fields, profile, primary key, stewardship, retention and policy binding stay unchanged. A revision bump does not authorize a breaking change.

6.1 A concrete revision

The custom customer table can advance from revision one to revision two by appending this field:

```
{
  "id": 2,
  "name": "note",
  "data_type": {"kind": "string"},
  "nullability": "optional",
  "semantic": "customer.note",
  "description": "Optional customer note",
  "classification": "internal"
}
```

Making the new field required would fail the successor check. Renaming `customer_id`, changing its sensitivity or replacing its description would also fail. The conservative rule prevents a familiar field from quietly acquiring a different business interpretation.

Changes outside this compatibility model need a separately reviewed migration or new logical contract. The current API does not pretend that a permissive compatibility flag can safely automate them.

6.2 Whole-registry identity

Every bound table carries the digest of the complete registry snapshot. Adding a table or changing one table therefore changes the expected digest on all bound tables. Deployment must reconcile every binding before activating a consumer for the new snapshot.

This trades rolling multi-version flexibility for an explicit, inspectable contract. The current deployment workflow uses a controlled consumer cutover. It does not provide a distributed transaction across the source repository, catalog and running processes.

7 Catalog projection and reconciliation

`CatalogBinding::create_plan` produces a LakeCat REST create-table request. The request contains the typed Iceberg schema and string properties carrying the full contract, enterprise, table identity, revision, wire version and registry digest. LakeCat determines physical metadata and storage location.

The host is responsible for namespace creation and authentication. The creation plan itself conveys no authority to execute the request.

7.1 Creation versus existing tables

LakeCat's standard create path assigns field IDs in declaration order. Pinax consequently requires IDs contiguous from one for a create plan. It rejects a sparse creation contract rather than silently renumbering fields.

Existing tables may have sparse IDs. `table_contract` derives the expected schema and properties without applying the creation restriction. `matches_schema` checks structural agreement while ignoring schema-ID allocation and field documentation. `matches_metadata` additionally checks the registry pins and serialized contract.

These checks can establish deployment readiness for an empty table. A scan has a stronger requirement: it needs a positive current snapshot and must check authorization again.

7.2 Reconciliation states

QueryGraph's prepared deployment service reports readiness, pending work or a blocked deployment. It compares current catalog state with the reviewed previous and target contracts. An authoritative table-not-found response can justify creating a new table. A transport failure cannot.

Existing-table changes use LakeCat's commit protocol and an expected table state token. A stale token rejects the mutation before the owner prepares a new write. This protects against a concurrent change between observation and application.

If a connection fails after a write, the result is uncertain. The deployment service requires reconciliation instead of blindly repeating the operation. On restart, it reloads state and skips updates that already completed. The final readiness check covers the entire target registry.

8 Governed discovery and planning

The host constructs `GovernedRegistry` from an immutable registry, a catalog binding and a trusted TypeSec engine with a deployed policy identity and revision. The host supplies the authenticated principal. A subject string inside a request is not authentication.

8.1 Discovery

Discovery checks whether the principal may read each table and its columns for the requested purpose. Denied items are omitted. An unresolved policy decision or a policy-binding mismatch fails the page.

Discovery examines a bounded page of registered tables and performs no catalog I/O. A page can contain no visible tables and still have a continuation. The cursor records a registry digest and position, not an authorization grant. Every next page rechecks policy, and a different registry rejects the old cursor.

The returned projection omits security and primary-key declarations that could reveal hidden field identities. Discovery is therefore its own disclosure boundary, rather than a serialization of the whole trusted registry.

8.2 A bounded scan intent

```
{
  "table": "customers",
  "columns": ["customer_id"],
  "purpose": "analytics",
  "limit": 100
}
```

The intent contains a logical table, explicit projection, allowed purpose and row limit. It contains no SQL, arbitrary filter, warehouse selector or storage path. Duplicate columns, unknown columns and * fail validation.

The Pinax planning limit ranges from 1 to 1,000,000 rows. The current QueryGraph execution extension accepts only 1 to 1,000. A caller must satisfy the stricter boundary for the operation it invokes.

8.3 Planning checks

Pinax checks the configured policy binding, rejects anonymous principals and checks the purpose allowlist. TypeSec then evaluates `read` on the table's revision resource and each requested column resource. Only an allow decision succeeds.

The host loads current catalog metadata. Pinax compares field IDs, names, types, nullability, order and complete registry pins. It requires a current snapshot and asks the configured engine to plan that exact snapshot. A plan for another snapshot is rejected.

Authorization can run before catalog I/O, so a denied request need not expose catalog existence or cause backend work. Planning repeats the check. A previous allow decision is not cached as a reusable capability.

9 Execution and evidence

A plan describes work. It does not authorize releasing its result. LakeCat's governed execution boundary checks current authority, retains the normalized request and proof, executes the read, and revalidates before returning rows.

The prepared QueryGraph integration adds an owner execution endpoint and a bounded Sail reader. The reader selects the exact Iceberg snapshot, applies the mandatory owner predicate, projects the allowed columns and respects the requested row limit. A tenant column can participate in filtering without appearing in the returned projection.

9.1 Buffering before release

The execution boundary buffers results so it can discard them if authority changes during the read. It does not stream partial protected rows before revalidation. The prepared implementation bounds retained Arrow data and encoded JSON, uses a bounded DataFusion memory pool and applies request deadlines.

The strict predicate compiler accepts its supported operator vocabulary and rejects unknown operators, transforms and fields. It limits expression depth, node count and scalar sizes. Unsupported security requirements cannot fall through to an unfiltered scan.

After execution, LakeCat emits lineage and checks authority again before release. QueryGraph verifies the returned principal, table, purpose, projec-

tion, snapshot, row shape, row count and catalog state. It independently recomputes the row and evidence digests, then reloads catalog state and checks the Pinax contract again.

9.2 What the evidence establishes

The result binds rows to the selected projection, snapshot, governed proof, authorization observations and lineage receipt. These bindings make a replaced row set or mismatched scope detectable.

Evidence hashes are integrity checks within the trusted service composition. Their presence alone does not establish that an arbitrary external server is honest. Deployments still need authenticated transport, pinned identities and trusted owner configuration.

Nor does final revalidation freeze the world. A later policy change can affect a later operation. The evidence records the checked operation and its release boundary rather than promising a perpetual right to the same data.

10 MCP sessions and credentials

QueryGraph's Rust MCP server owns the governed session. The optional Python CLI handoff replaces its process with the configured Rust executable, preserving stdio and request IDs. It does not create a second authority or duplicate the Pinax implementation.

```
querygraph mcp-serve --registry-config registry-service.json
```

An MCP client initializes the session, sends the initialized notification and then calls tools. Governed discovery, planning and execution each require an attached trusted service. Without one, they report that the backend is not configured.

10.1 Two credential boundaries

An MCP governed intent carries an operation-specific signed envelope. Its signature binds the exact intent body and selected operation. A valid planning signature cannot be reused to authorize execution.

The catalog HTTP adapter has a separate credential boundary. It mints a fresh encrypted TypeSec envelope for every request, using deployment-owned key material and a pinned recipient document. Replay protection prevents a static envelope from serving as a reusable session credential.

The catalog subject must match the authenticated agent. A configured subject string does not override the identity derived from the signing key. Test fixtures use fixed demonstration seeds; deployed services need real protected key material.

10.2 Cancellation and overload

The transport continues reading control messages while up to 16 governed requests await backend work. It rejects overload and duplicate in-flight IDs. A valid cancellation notification aborts the matching request without a response. EOF aborts and joins owned work. Governed operations have a 30-second deadline.

Cancellation cannot retract a completed response or guarantee that a remote HTTP server has stopped its own work. Synchronous tools retain their ordering, and an output write can wait on a slow reader. These limitations matter when operating the service under load.

11 A complete demonstration

The QueryGraph demonstration retains the semantic import, signed agent workflow and lineage path, then adds Pinax discovery and real governed Iceberg execution. The executable extension lives in QueryGraph's `demo/pinax/` directory. Its prepared source tree contains QueryGraph, Pinax, LakeCat and Sail; Cargo resolves the remaining released stack dependencies.

```
bash demo/pinax/build-ec2.sh "$DEMO_ROOT"
bash demo/pinax/run-ec2.sh "$DEMO_ROOT"
```

The build compiles on the demonstration host. It uses an explicit local source override for the unreleased Pinax integration and temporary fixture manifests for the prepared owner and engine. These choices are visible demo inputs, not committed release dependency shortcuts.

11.1 The row experiment

The fixture creates real Iceberg metadata and Parquet rows. One customer belongs to another tenant and one belongs to `acme`. A private email column exists in storage. The allowed request returns only the permitted customer ID. It does not return the other tenant or the private email.

The fixture checks the owner endpoint and both MCP entry paths. It recomputes the result digests independently. It then requests forbidden columns

and a forbidden purpose, submits a stale state token and removes a backend data file. Each failure must return no protected row set.

11.2 Changes during a real read

Three adversarial cases mutate authority after Sail has read actual rows: purpose revocation, registry drift and snapshot drift. Each must cause the buffered result to be discarded. This tests the release guard at the relevant time boundary; merely rejecting a bad request before a read would not prove the same property.

The cutover phase closes the old consumers, applies a reviewed registry successor, reconciles its pins and starts consumers configured for the target digest. Those consumers must read real rows under revision two. A consumer with a premature, stale or mismatched activation digest must fail startup.

11.3 Reading a report honestly

The original semantic demo without `--live-sail` does not prove a running Spark Connect service. The Pinax fixture does exercise the real Sail engine inside the authenticated LakeCat owner. Preserve that distinction in a talk, benchmark report or deployment review.

The fixtures demonstrate correctness on small representative records. They make no throughput, latency, TPC-DS scale or distributed exactly-once claim. Keep source revisions, source modifications, compiler versions, binary hashes and individual test outcomes alongside the report so it can be reproduced.

12 Operating a registry

Treat publication as a controlled deployment. Review the proposed document, validate it, compare it with the current snapshot and record the expected target digest. Confirm that the policy binding names the actual deployed engine and that catalog credentials authenticate the intended principal.

Plan the catalog changes before applying them. Reconcile the full registry, use the owner state precondition for each mutation and stop on unexpected drift. After an ambiguous response, observe current state before deciding whether more work remains.

The activation configuration pins the expected registry digest and checks all bound tables before admitting MCP sessions. It is an observation, not a distributed lock or lease. The operator still owns the maintenance window, process lifecycle and recovery procedure.

12.1 Failure diagnosis

Symptom	First boundary to inspect
Proposal rejected	Identifier, type, profile and required-field validation
Successor rejected	Revision progression and immutable existing contract
Empty discovery page	Purpose and policy, including continuation cursor
Planning rejected	Exact policy binding, projected columns and catalog pins
Execution rejected	Separate execution permission, owner state and current snapshot
Read discarded	Authority or metadata changed before result release
Consumer will not start	Expected activation digest and whole-registry reconciliation

Avoid diagnosing failures by returning private rows, credentials or metadata locations to the requesting agent. Operator logs and client-facing errors serve different audiences.

12.2 Release discipline

The QueryGraph family records released sibling pins in `QUERYGRAPH.md`. Release checks follow the dependency graph. A prepared source build must not be described as a successful registry-only consumer build until the required versions are actually published and resolve from the registry.

Before a family release, run the authoritative stack dependency checker. Update its matrix in the same change as dependency pin updates. Commit no sibling path or Git dependency as a substitute for a missing release.

13 The next contract

Pinax v1 chooses a small set of guarantees that can be checked end to end: validated immutable contracts, stable fields, conservative successors, trusted policy bindings and exact catalog agreement. The execution integration adds bounded real reads and evidence checked before release.

Future extensions may add richer nested schemas, relationships, rolling multi-version activation or additional predicate operators. Each extension needs an explicit domain model, compatibility rules and an enforcement boundary. Adding a descriptive field is insufficient if the execution path cannot honor it.

The useful question for every addition is concrete: which invalid state or unauthorized operation becomes impossible, and which test demonstrates that property through the actual stack? That question keeps the registry connected to the data it describes and the people responsible for it.

14 Source guide

The book follows the source repositories and their checked contracts:

- Pinax's `docs/pinax.md` defines the `pinax.v1` reference protocol.
- `src/registry.rs`, `src/model.rs` and `src/standards.rs` implement validation, compatibility and profiles.
- `src/adapters/` implements catalog projection and governed discovery/planning.
- QueryGraph's `docs/mcp-pinax-service.md` documents trusted service setup, MCP sessions and deployment activation.
- QueryGraph's `docs/governed-execution-contract.md` describes the owner execution contract and evidence checks.
- QueryGraph's `integration/pinax/` contains the real-row and cutover fixtures.
- `demo/pinax/` contains the EC2 demonstration scripts and their limitations.

Read the public source at github.com/querygraph/pinax and github.com/querygraph/querygraph. The hosted book is available through the First Pair library.