AMOS: An Internally Deployed Analyst System

Abstract

This document is the product architecture and build guide for AMOS. It defines the theory, system boundaries, data model, runtime protocol, component interfaces, security model, deployment topology, operating model, and phased implementation plan needed to build AMOS from an empty repository to a production service. The intended reader is a startup engineering team that must make the system work, operate it safely, and know when each layer is complete.

AMOS is an internally deployed analyst system that connects to company data and tools, answers business questions, performs verified analysis, and produces graphs, reports, and presentation slides. The customer receives a complete product with a local analyst agent, beginning with Gemma 4 support, rather than supplying a separate agent. The agent proposes plans, queries, interpretations, and report structure; the AMOS control layer retains authority over access, execution, verification, evidence, review, and publication.

The architecture is deliberately incremental. A startup first ships a modular monolith that completes one read-only analytical workflow from employee request through verified calculations, graphs, report, and editable slide deck. It then adds durable feedback, scheduling, multi-tenant governance, connector breadth, and controlled autonomy only after explicit correctness, security, reliability, and customer-value gates pass. The guide defines the complete subsystem map and the normative contracts that detailed designs must implement. Appendices A–F freeze the most dangerous first-release contracts. Appendices G–K describe the current Rust control-layer fixture and its known product gaps. Database DDL, generated schemas, and adapter conformance fixtures remain versioned engineering artifacts rather than claims hidden in prose.

Contents

1 Purpose, Audience, and Definition of Complete **6**
1.1 Product Definition . . . 6
1.2 Who Uses This Guide . . . 6
1.3 Normative Language and Completeness Standard . . . 6

2 The Product Problem and System Boundary **7**
2.1 Responsibilities . . . 7
2.2 First Product Scope . . . 7

3 Theory: Analytical Memory as an Operating Layer **7**
3.1 Memory Object . . . 7
3.2 Bounded Active Context . . . 8
3.3 Authority, Time, and Supersession . . . 8
3.4 Consistency Classes . . . 8

4 The Analytical State Transaction **8**
4.1 Transaction Record . . . 8
4.2 Lifecycle . . . 8
4.3 Isolation, Concurrency, and Side Effects . . . 9
4.4 Runtime Invariants . . . 9
4.5 Outcome State Machine . . . 9

5 End-to-End Product Architecture **9**
5.1 Component Responsibilities . . . 10

6 Product Surfaces and Operating Workflows **10**
6.1 Customer Journey . . . 10
6.2 Experience Invariants . . . 11

7 Data Architecture and Persistent Contracts **11**
7.1 Core Relational Model . . . 11
7.2 Memory Object Contract . . . 13
7.3 Indexes and Access Paths . . . 13
7.4 Retention and Encryption . . . 13

8 Connector Architecture **13**
8.1 Connector Build Order . . . 13

9 Context Compiler **14**
9.1 Compilation Algorithm . . . 14
9.2 Failure Behavior . . . 14
9.3 Compaction Contract . . . 14

10 Agent Runtime and Tool Execution **14**

10.1 Typed Planning . . . 14
10.2 Capabilities and Workers . . . 14
10.3 Repair Loop . . . 15

11 Verification, Claims, and Publication 15
11.1 Pre-Execution Verification . . . 15
11.2 Post-Execution Verification . . . 15
11.3 Claim Dependency Graph . . . 15
11.4 Publication Linearization Point . . . 15

12 Feedback, Learning, and Invalidation 16
12.1 Replay Levels . . . 16

13 API and Event Contracts 16

14 Security and Multi-Tenancy 17
14.1 Trust Boundaries . . . 17
14.2 Policy Decision Contract . . . 18
14.3 Tenant Isolation . . . 18
14.4 Privacy and Governance . . . 18

15 Deployment and Scalability Architecture 18
15.1 Initial Production Topology . . . 18
15.2 When to Extract Services . . . 18
15.3 Scaling Strategy . . . 18

16 Operations, Reliability, and Support 19
16.1 Service Objectives . . . 19
16.2 Observability . . . 19
16.3 Runbooks . . . 19

17 Testing and Quality Gates 19
17.1 Definition of Done for a Feature . . . 19

18 Step-by-Step Build Plan 20
18.1 Reference Repository Shape . . . 21
18.2 First Vertical Slice . . . 22
18.3 Team and Ownership . . . 22

19 Current Rust Implementation to Production 22

20 Low-Level Execution and Performance Engineering 22
20.1 Hot-Path Cost Model . . . 22
20.2 Concurrency, Allocation, and I/O . . . 23
20.3 Performance Gates . . . 24

21 Architecture Decisions and Non-Goals 24

22 Harsh Qualification Checklist **25**

23 Conclusion **25**

A Specification A: A-TXN, Concurrency, and Publication **26**
A.1 Persistent State Machine . . . 26
A.2 Isolation, Compare-and-Swap, and Fencing . . . 26
A.3 Normal Task and Commit-Time Revocation . . . 27
A.4 External Publication Protocol . . . 27

B Specification B: Persistent Model and Database Invariants **28**
B.1 Keys, Constraints, and Roles . . . 28
B.2 Example A-TXN Record . . . 29
B.3 Object Finalization and Retention . . . 29

C Specification C: Task Definitions and Context Manifests **30**
C.1 Versioned Task Definition . . . 30
C.2 Example Context Manifest . . . 31
C.3 Selection and Compaction Rules . . . 31

D Specification D: Connector SDK and Certification **32**
D.1 Typed Connector Interface . . . 32
D.2 Certification Matrix . . . 32
D.3 Example Source-Change Event . . . 33

E Specification E: Workers, Capabilities, and Verification **33**
E.1 Capability Claims . . . 33
E.2 Typed Plan and Verifier Result . . . 34
E.3 Worker Protocol and Deferred Python . . . 34
E.4 Verifier Repair and Worker Crash Sequences . . . 34

F Specification F: Claims, Review, Invalidation, and Replay **35**
F.1 Structured Claim and Dependency Edge . . . 35
F.2 Review Correction, Replay Package, and API Error . . . 36
F.3 Validity Transition Rules . . . 37
F.4 Review and Source-Change Sequences . . . 38
F.5 Replay Sequence . . . 38

G Current Rust Implementation Model **39**
G.1 Audit Baseline and Scope . . . 39
G.2 Complete Public Type Inventory . . . 39
G.3 Exact Local Persistence Schema . . . 42
G.4 Transport and CLI Surface . . . 43

H Foundational Function Reference **45**
H.1 Domain, Authentication, Errors, Policy, Memory, and Context . . . 45

I Executable Verification Matrix **48**
I.1 Original Library Unit and Storage Contracts (17 of 31) . . . 48
I.2 Original API and Binary Contracts (9 of 12) . . . 48
I.3 Original End-to-End, Security, and Concurrency Contracts (16 of 18) . . . 49
I.4 Completion Contracts Added After the Baseline (19) . . . 50

J Production Gap Register and Completion Checklist **51**
J.1 Build and Release Procedure . . . 52

K Service, Persistence, and Transport Function Reference **54**
K.1 Runtime Orchestration . . . 54
K.2 HTTP, UI, Seed, and Binary . . . 55
K.3 Evidence, Scheduling, and Persistence . . . 57
K.4 Connector, Workers, and Verification . . . 61

1 Purpose, Audience, and Definition of Complete

1.1 Product Definition

AMOS is an internally deployed analyst system that connects to company data and tools, answers business questions, performs verified analysis, and produces graphs, reports, and presentation slides. It is not a warehouse, a general document search engine, or only a language model. It includes the analyst agent and the control, execution, verification, evidence, and artifact services needed to deliver completed analytical work.

The long-term product goal is to replace the end-to-end work performed by data analysts and business analysts for supported workflows. This outcome is established workflow by workflow through correctness, coverage, review burden, reliability, cost, and customer evidence; it is not inferred from a fluent model response or a completed control-layer fixture.

The product has three surfaces over one substrate:

1. **Analytical memory:** a typed and versioned representation of the company's metrics, schemas, data states, documents, policies, decisions, prior work, and feedback.
2. **Analyst and control runtime:** a customer-local model proposes plans and interpretations; a governed execution environment controls read-only SQL, deterministic statistics and chart tools, verification, review, scheduling, and recovery.
3. **Artifact product:** deterministic compilers produce graphs, reports, editable presentation slides, dashboards, and spreadsheets with claim-level evidence.

The required end-to-end flow is: employee or scheduled request → local analyst agent (initially Gemma 4) → AMOS control layer → company data and tool connectors plus bounded SQL/statistics/chart workers → verification and evidence → report and slide plan → deterministic artifact compiler → charts, PPTX, PDF, dashboard, and spreadsheet.

1.2 Who Uses This Guide

The primary readers are the founding engineer, data-systems engineer, agent-runtime engineer, product engineer, security owner, and design-partner lead. Sections 2–5 define why the system works. Sections 6–12 define what to build. Sections 13–16 define how to ship, operate, test, and expand it.

1.3 Normative Language and Completeness Standard

MUST: marks a release-blocking invariant. **SHOULD:** marks the default whose exception requires an architecture decision record. **MAY:** marks an optional compatible behavior. **DEFERRED:** identifies a capability intentionally excluded from the first release. **NON-GOAL:** identifies behavior outside the product boundary. Unlabeled explanatory text describes rationale; the appendices contain the executable first-release contract.

The architecture map is complete only when an engineer can answer all of the following without inventing a new subsystem:

- where every governing definition, permission, execution, claim, and artifact is stored;
- how identity and policy are enforced before model-visible context is constructed;
- how a task moves from admission to publication, abort, repair, or human review;
- how source versions are observed and revalidated at commit time;
- how every important claim resolves to computation and governing state;
- how feedback changes future work without silently rewriting history;
- how a source change invalidates dependent conclusions;
- how the service is deployed, monitored, recovered, and tested;
- what every production function and trait method does, which state it reads or writes, and how it fails;
- which current implementation behaviors are measured facts versus target production contracts;
- which capabilities are intentionally deferred and what gate unlocks them.

2 The Product Problem and System Boundary

An analytical task depends on independently changing state. At time t:

$$\mathcal{S}_t = (q_t, D_t, C_t, M_t, X_t, H_{<t}, F_{<t}, P_t), \tag{1}$$

where q_t is the request or trigger, D_t is the data snapshot or stream position, C_t is schema and catalog state, M_t is metric and semantic state, X_t is governed document state, $H_{<t}$ is prior analytical work, $F_{<t}$ is reviewed feedback, and P_t is identity and policy.

These sources do not share one transaction manager. A metric can change after retrieval, a table can migrate during execution, a watermark can advance, and a permission can be revoked before publication. A correct-looking answer can therefore be state-invalid even when its SQL ran successfully. The product problem is to control this cross-source state transition without copying all company data into a prompt or replacing the systems of record.

2.1 Responsibilities

AMOS owns:

- typed metadata, governed summaries, external references, aliases, versions, authority, and supersession;
- identity-aware context construction and task-scoped capabilities;
- plan, tool, verification, claim, artifact, review, replay, and invalidation records;
- a stable runtime contract above replaceable models, indexes, warehouses, and application surfaces.

Source systems continue to own raw data, primary access control, warehouse transactions, catalog records, semantic definitions, document originals, and identity truth. AMOS product surfaces own the default employee workflow and artifact experience; approved customer applications may also call the same runtime contract. Models own no durable authority: model output is a proposal until a deterministic gate accepts it.

2.2 First Product Scope

MUST: The first production slice supports one tenant, one warehouse, one approved metric family, a customer-local analyst model, read-only SQL, deterministic built-in statistics, structured chart generation, a verified report, an editable presentation deck, typed claims, reviewer approval, and replay. The model provider is replaceable; Gemma 4 is the first supported local family. **DEFERRED:** General Python, free-form authoritative claim extraction, arbitrary production writes, unrestricted notebook code, general multi-agent scheduling, unreviewed causal claims, and autonomous external communication. Expansion occurs only through the gates in Section 18.

3 Theory: Analytical Memory as an Operating Layer

3.1 Memory Object

Persistent analytical memory is a set $\mathcal{M}_t$ of typed objects. Each object is:

$$m = (id, k, v, \tau_e, \tau_r, a, p, s, r, h), \tag{2}$$

where k is type, v is content or an external reference, τ_e is the effective-time interval, τ_r is recording time, a is authority, p is permission scope, s is lifecycle status, r is provenance, and h is a content hash. Logical identity and physical version are separate: multiple versions can describe one metric or schema, but only versions valid for the task interval and policy may govern a run.

Core types are data or stream state, schema/catalog state, semantic definition, governed document, prior analysis, feedback, permission policy, execution, artifact, claim, and provenance edge. Raw fact tables remain external; memory stores identifiers, governed summaries, versions, and immutable references.

3.2 Bounded Active Context

The model-visible working set $\mathcal{W}_t \subset \mathcal{M}_t$ is compiled for one task. It is not ordinary top-k retrieval. Selection must satisfy:

$$permitted(u, p_t, m) \qquad \forall m \in \mathcal{W}_t, \tag{3}$$

$$effective(m, t_q) \wedge active(m), \qquad \forall m \in \mathcal{W}_t, \tag{4}$$

$$\sum_{m \in \mathcal{W}_t} tokens(m) \leq B, \tag{5}$$

$$\forall r \in R(q) : \exists m \in \mathcal{W}_t \text{ covering role } r. \tag{6}$$

The required-role set $R(q)$ typically includes metric, schema, data state, policy, and supporting evidence. A missing mandatory role produces an explicit non-final outcome rather than a guessed answer. This is the central scale separation: persistent memory may grow to millions of objects while model context remains bounded by B.

3.3 Authority, Time, and Supersession

Recency alone does not determine truth. The default authority order is owner-approved $>$ reviewer-approved $>$ user note $>$ model hypothesis $>$ untrusted external content. Effective time determines when a statement applies; recording time determines when AMOS learned it. Supersession edges explicitly replace one logical version with another. Conflicting owner-approved objects for the same interval block publication until resolved.

3.4 Consistency Classes

Each connector declares the strongest attainable observation class:

- **C0 observed:** the version and observation time are recorded, but the source cannot guarantee re-readability.
- **C1 validated:** the version can be re-read or compared at commit, so changes are detectable.
- **C2 retained:** the exact version is retained or leased long enough for deterministic reconstruction.

Task policy declares minimum classes per required role. Regulated or high-impact artifacts may require C2 data and C1 policy/semantic state; exploratory tasks may accept C0 with a visible warning. **MUST:** A connector may advertise C1 or C2 only after passing the certification behaviors in Specification D; a timestamp label alone is never a version guarantee.

4 The Analytical State Transaction

A-TXN is the system contract connecting memory, tools, validation, publication, and later invalidation. It is not an ACID transaction across external systems. It creates a truthful AMOS commit record after optimistic revalidation of the external versions used.

4.1 Transaction Record

An A-TXN record contains:

$$A = (id, u, q, P^0, V^0, \mathcal{W}, K, E, C, \mathcal{G}, o, t), \tag{7}$$

where P^0 is the admitted policy epoch, V^0 is the source-version vector, K is the scoped capability set, E is execution evidence, C is the claim set, $\mathcal{G}$ is the dependency graph, o is outcome, and t contains lifecycle timestamps.

4.2 Lifecycle

1. **BEGIN:** authenticate identity, bind tenant, classify risk, assign budgets, record policy epoch, and create an idempotent A-TXN ID.
2. **OBSERVE:** ask connectors for source versions, freshness, retention class, and access capabilities.

3. **SELECT:** filter candidates before ranking; reconcile identity, time, authority, and supersession; require role completeness.
4. **PLAN:** produce a typed plan with declared inputs, tool operations, expected outputs, budgets, and review obligations.
5. **EXECUTE:** issue short-lived capabilities to isolated workers; verify each operation before execution; record parameters, hashes, timings, and results.
6. **COMPOSE:** construct typed claims from verified result objects, then render the artifact; prose extraction is not authoritative in the first release.
7. **VERIFY:** validate SQL, schema, metrics, freshness, permissions, claim support, and risk-specific rules.
8. **REVALIDATE:** re-read the policy epoch and every C1 version; confirm C2 retention leases.
9. **COMMIT:** atomically store artifact metadata, claims, dependencies, verification, replay descriptor, and audit event.
10. **DISSEMINATE:** enqueue an idempotent publication request only after evidence commit and object finalization; record external acknowledgment separately.
11. **INVALIDATE:** on later source events, traverse reverse dependencies and update semantic validity, policy visibility, replay availability, review, and supersession dimensions independently.

4.3 Isolation, Concurrency, and Side Effects

MUST: Long model and tool work never holds a database transaction open. Short PostgreSQL transactions use `READ COMMITTED`; every state transition compares the expected `state` and `state_seq`, and every worker lease carries a monotonically increasing fencing token. Terminal states are immutable. A state mutation and its outbox event commit in the same database transaction. Every external side effect uses a stable tenant-scoped idempotency key. Specification A defines the transition, locking, retry, object-promotion, and external-publication protocols.

4.4 Runtime Invariants

The implementation must enforce these invariants:

1. no inaccessible memory enters model-visible context;
2. every selected object is effective, active, non-superseded, and source-qualified;
3. every tool call is allowed by both policy and a task-scoped capability;
4. execution references only declared working-set objects and observed source versions;
5. numeric claims have computation, data-state, schema, and metric support;
6. causal or high-impact recommendations remain review-gated unless an approved design permits automation;
7. commit fails or becomes explicitly non-final when governing C1 state changed;
8. memory writes preserve authority, history, source, and the transaction that produced them;
9. stale workers cannot commit after a newer fencing token is issued;
10. evidence commit, object finalization, and external dissemination have separate persistent states;
11. every final artifact has a declared replay level and complete audit record.

4.5 Outcome State Machine

5 End-to-End Product Architecture

The startup should implement the first release as a modular monolith with strict internal interfaces. This provides one transactional database, one deployment unit, and one debugging surface while preserving seams for later service extraction.

Outcome	Meaning	Required behavior
pass	All required checks succeed.	Commit as final within the declared risk class and replay level.
warning	Work is structurally valid with a disclosed limitation.	Commit as non-final or limited-use; surface the exact warning and affected claims.
repair	A bounded, policy-safe correction is possible.	Record failed attempt, patch the plan or operation, and retry within budget.
needs_review	Evidence exists but judgment or approval is required.	Commit evidence and draft artifact; create a review task; prohibit final publication.
reject	Policy, safety, validity, or support fails.	Store the rejection and diagnostics; do not publish the artifact.
abort	Infrastructure, lease, source, or concurrency failure prevents a truthful decision.	Preserve partial evidence, release capabilities, and allow idempotent retry.

Table 1: Every task ends in an explicit state; silence and best-effort finalization are forbidden.

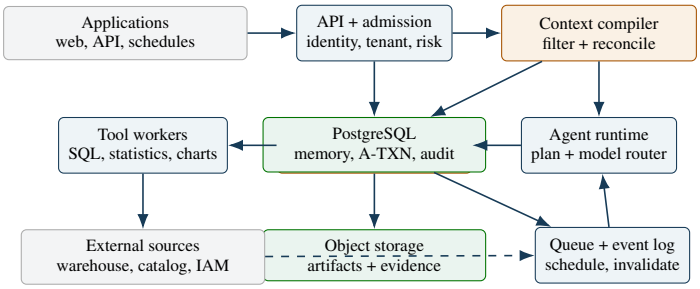

Figure 1: Production logical architecture. Internal boundaries remain explicit even when deployed as one application.

5.1 Component Responsibilities

6 Product Surfaces and Operating Workflows

The control plane is useful only when its state is legible to customers. AMOS therefore ships four product surfaces over the same APIs and permission checks; none receives privileged database access.

The current Rust MVP implements authenticated, server-rendered versions of all four surfaces. The Analysis Workspace still submits a fixed legacy payment-health fixture and displays the artifact and evidence ledger; this fixture is implementation evidence, not the product scope. Memory Studio lists only policy-visible memory; Review Queue requires reviewer authority and links to artifact evidence; Operations Console requires administrator authority and lists audit events. The administrator API exposes jobs, metrics, retention, erasure, connector health, and source-event processing. Connector configuration, metric editing, publication destinations, model integration, report and slide planning, and interactive retry/recovery controls remain incomplete product interfaces rather than claims made for these four local pages.

6.1 Customer Journey

Onboarding is an auditable promotion workflow. An administrator creates the tenant, connects a read-only source, confirms identity synchronization, and chooses retention and region. The connector discovers schemas and capabilities into a staging namespace. A data owner resolves aliases, defines or imports a metric, assigns authority, and approves a version. A product owner then configures an application template with task schema, required roles, risk class, review rule, schedule, budget, and artifact shape. Production activation is prohibited until connector, permission, metric, task-fixture, and recovery checks pass.

During normal use, a request or schedule creates a transaction and immediately displays its policy scope. The workspace streams durable state changes rather than model tokens as the source of truth. A user can open the exact context manifest, each execution, each verifier result, and the dependency chain of every material claim. Final status distinguishes *published*, *published with warning*, *awaiting review*, *rejected*, *stale*, and *revoked*; a draft is never visually equivalent to a governed artifact.

Component	Owns	Must not own
API and admission	Authentication, tenant binding, request validation, idempotency, risk class, quotas.	Prompt construction, warehouse credentials, or final publication logic.
Connector manager	Source discovery, version observation, freshness, access probes, change events, read handles.	Cross-source reconciliation or company-wide policy decisions.
Memory service	Typed objects, versions, authority, effective time, aliases, supersession, indexes.	Arbitrary raw customer data or model prompts as the system of record.
Context compiler	Required-role derivation, policy-first filtering, ranking, reconciliation, compaction, manifest.	Tool execution or hidden fallback when a required role is missing.
Agent runtime	Typed planning, model routing, budgets, checkpoints, repair loop, artifact draft.	Credentials, unrestricted tools, or authority to bypass verifier outcomes.
Tool workers	Read-only SQL, deterministic statistics, structured charts, result capture; later sandboxed Python behind the same interface.	Long-lived tenant secrets, arbitrary alpha code, or direct publication.
Verifier	SQL safety, schema, metrics, freshness, permissions, claim support, risk and review rules.	General truth or causal certainty beyond declared rules.
Evidence service	Claims, dependency edges, hashes, verification, artifacts, replay descriptors.	Silent mutation of committed evidence.
Scheduler and invalidator	Durable jobs, retries, source-change fan-out, revalidation and review tasks.	Unbounded automatic reruns or publication without current policy.
Review application	Evidence inspection, approve/reject/correct, scoped feedback, audit export.	Direct mutation of historical objects or unscoped high-authority feedback.

Table 2: Ownership boundaries prevent the language model or any single service from becoming an implicit superuser.

Surface	Primary user	Required capabilities
Memory Studio	Data owner and administrator	Connect and test sources; inspect discovered objects and versions; define aliases, metrics, authority, retention, and policies; approve changes; view source health and impact before activation.
Analysis Workspace	Analyst and business operator	Submit or schedule a typed task; see admitted scope, plan, budget, progress, warnings, and terminal state; inspect an artifact down to claim, computation, and source; export approved results.
Review Queue	Reviewer and domain owner	Compare evidence with the draft; approve, reject, or correct at claim, metric, template, or policy scope; assign effective time and authority; see which future tasks and existing claims are affected.
Operations Console	Support, security, and platform operator	Search transactions and audit events; inspect connector, queue, worker, and policy health; retry an idempotent step, revoke a capability, quarantine a connector, revalidate claims, and run recovery procedures.

Table 3: The product exposes governed state and explicit controls instead of presenting an opaque chat transcript.

After review, corrections append new governed objects and show their scope before confirmation. The next applicable transaction selects them by authority and effective time. Source or policy change computes impact, marks dependent claims, and either revalidates, schedules bounded re-analysis, or requests review. The user can always answer: what changed, what AMOS did, what remains valid, and who has authority to resolve the rest.

6.2 Experience Invariants

Every UI action uses the same idempotent API and produces an audit event. Hidden background work has a visible job and owner. Destructive actions are modeled as revocation, supersession, or policy-governed erasure. Restricted object names and failure details are redacted at the API, not merely in the interface. No screen offers a bypass around review, verification, retention, or tenant boundaries.

7 Data Architecture and Persistent Contracts

7.1 Core Relational Model

Use PostgreSQL as the authoritative store. Object storage holds large artifacts, source exports, chart images, and retained execution payloads. A queue transports durable jobs; it does not replace the transaction log.

Table	Primary identity	Required content and rule
tenants	tenant_id	Status, retention policy, encryption key reference, feature gates. Every governed row includes tenant_id.
identities	tenant_id + subject_id	Provider, roles, groups, policy attributes, last synchronized epoch.
source_systems	source_id	Connector type, configuration reference, attainable consistency classes, health. No plaintext secret.
memory_objects	object_id	Type, logical key, source, authority, permissions, effective interval, recorded time, status, content hash, external reference.
memory_versions	logical_key + version	Immutable content or reference, source version, transaction time, supersedes link. Updates append; they do not overwrite.
alias_edges	edge_id	Source-qualified logical identity mapping, authority, effective interval, reviewer state.
policies	policy_id + epoch	Declarative admission, retrieval, tool, publication, retention, and review rules.
policy_decisions	decision_id	Subject, action, resource, attributes, allow/deny, obligations, policy references, epoch, safe explanation.
task_definitions	task_type + version	Required/optional roles, consistency minima, tools, claim types, verifier and publication rules, budgets, artifact schema.
atxn_transactions	atxn_id	Request, identity, risk, budgets, state/state sequence, policy epoch, version vector, timestamps, terminal outcome.
context_manifests	manifest_id	Selected object versions, required-role coverage, omissions, conflicts, token counts, ranking trace.
plans	plan_id	Typed steps, declared inputs/outputs, tools, budgets, dependencies, model/configuration identity.
executions	execution_id	Capability, worker, tool version, parameters hash, input versions, output reference/hash, latency, cost, status.
artifacts	artifact_id	Type, immutable object reference, audience, risk, creator transaction, object-finalization state.
artifact_publications	publication_id	Adapter, idempotency key, lifecycle state, external ID, acknowledgment, attempts, revocation state.
claims	claim_id	Typed payload, rendered span, risk, support plan, review state, semantic validity, policy visibility, replay and supersession state.
dependency_edges	edge_id	Claim-to-query, data, metric, schema, document, execution, feedback, or decision relation.
verification_records	verification_id	Check type/version, inputs, result, warnings/errors, repairability, timestamp.
replay_packages	package_id	Replay level, retained inputs, tool/model versions, parameters, hashes, environment descriptor.
reviews	review_id	Reviewer identity, decision, comment, correction scope, authority, affected claims, timestamp.
audit_events	event_id	Append-only actor, action, target, request/A-TXN IDs, outcome, policy epoch, timestamp.
jobs	job_id	Schedule/trigger, idempotency key, state, retry policy, lease, fencing token, next run, dead-letter reason.

Table 4: Minimum production schema. JSON payloads may supplement but never replace indexed identity, time, policy, and state columns.

7.2 Memory Object Contract

Every memory object must contain a stable ID, source-qualified logical key, type, human-readable summary, structured payload or external reference, source, authority, effective start/end, recorded timestamp, permission labels, sensitivity, version, status, supersedes/superseded-by links, provenance reference, and content hash. Optional confidence cannot override authority.

Write rules are strict. Connectors may write source-observed objects within their namespace. Reviewed corrections may create reviewer-approved feedback. Only designated owners can create owner-approved definitions. Model-inferred memory starts as a hypothesis and cannot govern a high-risk task until approved. Deletion becomes revocation or tombstoning unless a retention policy requires physical erasure.

7.3 Indexes and Access Paths

MUST: The first release uses relational indexes and PostgreSQL full-text search with mandatory tenant and coarse permission predicates. Required indexes include tenant/type/status, logical key/version, effective interval, permission label, source version, authority, supersession, transaction state, job lease, and reverse dependency target. Object-level authorization runs after candidate retrieval. Restricted candidates cannot affect user-visible counts or ranking explanations. Indexes, embedding caches, lexical caches, and object-store keys are tenant-scoped and policy-epoch-aware. **DEFERRED:** Shared cross-tenant vector indexes. A vector or hybrid backend may later generate candidates only behind the same permission-first interface.

7.4 Retention and Encryption

Encrypt transport and storage. Use per-environment keys initially and tenant-specific keys for enterprise isolation. Store secret references in a secret manager. Define retention independently for prompts, model responses, query results, raw evidence, artifacts, audit events, and C2 snapshots. A replay claim is allowed only when the required data remains retained for its declared interval.

8 Connector Architecture

Each connector implements a small, testable contract:

- `discover(scope)` returns source-qualified entities and capabilities;
- `observe(ref)` returns version, effective time, freshness, sensitivity, access, and consistency class;
- `read(ref, version, capability)` returns governed content or a bounded read handle;
- `validate(ref, observed_version)` confirms whether a C1 observation still holds;
- `subscribe(cursor)` emits version, schema, permission, freshness, or deletion events;
- `health()` reports authentication, rate limits, lag, and degraded capabilities.

Connector output is untrusted until normalized and policy-labeled. Source text never carries executable policy. Each connector receives least-privilege, tenant-scoped credentials and runs behind egress restrictions. Pagination, retries, backoff, rate-limit state, and source cursors are durable.

8.1 Connector Build Order

Build one warehouse adapter first, then identity/policy, semantic definitions, catalog/schema, documents, and event/stream state. Each new adapter must ship with fixtures, contract tests, access-revocation tests, version-change tests, pagination tests, cache-staleness tests, deletion tests, source-outage behavior, and a failure-mode table. C1 certification additionally proves stable version identity, delayed revalidation, overwrite detection, and permission-change observation. C2 additionally proves exact version reads, retention/lease deadlines, expiry behavior, and reconstruction checksums. Do not add a connector solely for breadth; add it when a paid workflow requires one of its roles.

9 Context Compiler

The context compiler turns an approved, versioned task definition into a permission-safe working set and an auditable manifest. **MUST:** Required roles are stored in `task_definitions`; they are never inferred only by a prompt. The record also freezes consistency minima, allowed tools, claim types, verifier rules, publication requirements, budgets, and artifact schema. Specification C provides the contract.

9.1 Compilation Algorithm

1. derive required and optional roles from the task type and risk policy;
2. query candidate indexes with tenant and permission pushdown;
3. remove inaccessible, revoked, expired, and superseded objects before semantic ranking;
4. group by source-qualified logical key and resolve aliases explicitly;
5. reconcile effective time and authority; surface equal-authority conflicts;
6. choose at least one valid object for every mandatory role;
7. fill remaining budget by relevance, authority, freshness, verification state, and diversity;
8. compact only by producing a non-governing derived object with links to every source object;
9. emit a context manifest containing selections, omissions, conflicts, warnings, token cost, and source versions.

9.2 Failure Behavior

Missing role coverage yields `needs_review` or `reject`. Ambiguous approved metrics yield a clarification request. Conflicting owner definitions block. Insufficient token budget never silently removes policy, schema, metric, or data-state roles; it drops optional evidence first. Restricted-object counts and identifiers are not revealed to unauthorized users.

9.3 Compaction Contract

MUST: A compacted summary is non-governing evidence, carries the strictest permission scope of every source, links every source version, and is invalidated when any source changes or is revoked. It may not replace a required metric, schema, policy, or data-state object. At most one summary generation may depend on another summary in the first release. Promotion requires measured token reduction and a regression test for factual and permission leakage. **DEFERRED:** Authority-bearing learned summaries.

10 Agent Runtime and Tool Execution

10.1 Typed Planning

A plan is data, not free-form prose. Each step declares purpose, required object IDs, tool, parameters schema, expected output, dependency steps, timeout, row/data limits, cost budget, retry policy, verification rules, and review obligation. The runtime validates the plan before any capability is issued.

Model routing is policy-driven. A small classifier may assign task type, a coding model may propose SQL, and a reasoning model may plan an investigation. The stored plan records the exact model provider, model ID, configuration, prompt template version, input manifest, latency, token use, and response hash. Models are replaceable processors; persistent memory and safety controls are not provider-specific.

10.2 Capabilities and Workers

The runtime exchanges a plan step for a short-lived, single-purpose capability. A signed capability binds issuer, audience, tenant, A-TXN ID, plan step, identity, tool, allowed source and operations, resource limits, issued/not-before/expiry times,

unique token ID, policy epoch, and fencing token. Workers reject expired, mismatched, superseded, or replayed capabilities and record the token ID without storing the bearer token.

MUST: Alpha workers are limited to read-only SQL, deterministic built-in statistics, predefined transformations, and structured chart specifications. SQL workers use source credentials unavailable to the model and enforce approved schemas, statement timeouts, row/byte limits, and query tagging. Every execution writes a hash-addressed result and audit record. **DEFERRED:** General Python requires a separate sandbox safety case covering image provenance, packages, filesystem, network, secrets, resources, determinism, and replay.

10.3 Repair Loop

Repairs are bounded. The verifier may return a machine-readable reason and permitted repair class, such as rename a superseded column or add a required metric predicate. The runtime may attempt at most the policy limit, preserving every failed plan and execution. Permission, policy, or authority failures are never model-repairable.

11 Verification, Claims, and Publication

11.1 Pre-Execution Verification

For SQL, the verifier parses an AST and checks read-only safety, approved relations and columns, metric-required filters, time semantics, groupings, joins, unsafe functions, row limits, and permission scope. It checks schema version, data freshness, snapshot/watermark state, and whether all referenced objects are in the context manifest. Unsupported constructs produce review or rejection; string matching alone is insufficient.

11.2 Post-Execution Verification

After execution, verify result shape, empty or partial outputs, freshness, numeric consistency, denominator rules, uncertainty requirements, chart-data equality, and output sensitivity. **MUST:** The alpha constructs typed claims from verified result objects before rendering prose. Each governed template field declares its claim type and support schema; numeric values never originate from model-written text. **DEFERRED:** Claim extraction from arbitrary prose, notebooks, dashboards, and application payloads is a compatibility feature and cannot be the source of truth.

Claim types include numeric, comparison, trend, forecast, causal, and operational recommendation. Each type has a support plan. Numeric claims require query/result, data state, metric, schema, execution hash, and verification. Causal claims require evidence and remain review-gated unless a pre-approved causal design applies. Recommendations require supporting facts, policy, decision authority, and an explicit reviewer or automation rule.

11.3 Claim Dependency Graph

For claim c, AMOS records edges to computations, data versions, metrics, schemas, documents, feedback, model outputs, verification records, and review decisions:

$$traceable(c) \not\Rightarrow correct(c) \not\Rightarrow causally\_justified(c). \tag{8}$$

The graph proves traceability and supports change impact. Verification supports limited consistency. Causal justification remains a separate methodological judgment.

11.4 Publication Linearization Point

Immediately before commit, the runtime revalidates policy epoch and all C1 observations. If any governing state changed, it aborts, repairs, or requests review. The evidence linearization point is the atomic database commit that stores artifact metadata, typed claims, dependencies, verification, replay descriptor, audit event, and object-finalization outbox entry. Large blobs upload first under a temporary key and promote idempotently by content hash.

External dissemination is a separate transaction with explicit forward and revocation paths:

```
draft -> prepared -> evidence_committed -> publication_pending -> published
  publication_pending -> publication_failed; revocation_pending -> revoked
```

An outbox worker invokes a publication adapter with a stable `publication_id`; it stores the external identifier and acknowledgment. Lost acknowledgment is recovered by querying or retrying with the same idempotency key, never by creating a second publication. Email, chat, dashboard, ticket, and webhook adapters cannot claim exactly-once behavior unless their target exposes compatible idempotency.

12 Feedback, Learning, and Invalidation

Human feedback is a governed state transition. A correction records author, role, authority, scope, effective interval, target artifact/claims, supporting decision, applicability conditions, and provenance. It never silently edits the old artifact. The next applicable context compilation can select the correction according to time, scope, and authority.

Source events trigger reverse dependency traversal. A changed metric, schema, document, permission, or data state identifies affected claims. **MUST:** Validity is multidimensional: `publication_validity`, `semantic_validity`, `policy_visibility`, `replay_availability`, `review_state`, and `supersession_state` change independently. Thus a claim can remain historically valid at publication time while becoming semantically stale, hidden from a newly unauthorized viewer, or unreplayable after retention expiry. Fan-out is bounded by tenant quotas, risk priority, deduplication keys, and cooldown windows. External notifications occur only after a new evidence commit and current policy decision.

12.1 Replay Levels

Level	Promise	Required retention
0	Disposable exploration	Audit event only; no regeneration claim.
1	Evidence reconstruction	Object versions, queries, documents, results, and review records.
2	Artifact regeneration	Level 1 plus structured artifact inputs, templates, and deterministic render configuration.
3	Computational replay	Level 2 plus C2 data, code, parameters, tool environment, and result hashes.
4	Audit package	Level 3 plus policy, identity, approvals, signed manifests, and exportable chain of custody.

Table 5: Replay language is precise; model trajectory replay is recorded when possible but is not guaranteed.

13 API and Event Contracts

The production public API is versioned under `/v1`. All responses carry validated request and correlation identifiers in headers, and failures use a stable error envelope and bearer challenge. Task, replay, review, job, retention, and erasure commands require explicit idempotency keys. Immutable memory source versions, execution effects, state transitions, connector/source events, invalidation pages, and outbox effects derive stable semantic identities at the domain boundary. The original direct-resource success shapes remain unchanged for v1 compatibility; a universal success wrapper would be a breaking versioned API change, not a property silently attributed to v1.

Production events use an envelope containing event ID, type, schema version, tenant, source, subject, occurred/observed times, correlation and A-TXN IDs, payload hash, and trace context. Consumers are idempotent. The local outbox persists tenant, event ID/type, aggregate ID, idempotency key, JSON payload, creation/completion times, delivery state, attempt budget, lease owner/expiry, fencing token, next attempt, and last error. Its bounded dispatcher completes, exponentially retries, or dead-letters under owner/fence checks. Schema-version and distributed-trace fields remain contracts for the deployment broker adapter. Required event families are:

- source and policy: `source.version.changed`, `policy.epoch.changed`, `permission.revoked`;
- task and evidence: `task.state.changed`, `artifact.evidence_committed`, `claim.validity_changed`;

Endpoint	Purpose	Contract
`POST/v1/tasks`	Admit and run analysis	Request and required idempotency key; returns the complete typed run result.
`GET/v1/tasks/{id}`	Observe lifecycle	Owner-or-reviewer authorized transaction state and outcome.
`POST/v1/memory/search`	Inspect governed memory	Permission-first typed results over a bounded time interval.
`GET/POST/v1/memory`	List or create memory	Read is policy-filtered; write authority and source are caller constrained.
`POST/v1/memory/{id}/supersede`	Append a replacement	Atomically marks the old logical version superseded and inserts the replacement.
`POST/v1/verify/sql`	Preflight proposed SQL	Structured pass/warning/repair/reject, rule IDs, and referenced versions.
`GET/v1/artifacts[/{id}]`	List or inspect artifacts	Owner-or-reviewer authorization plus claim visibility and dependency edges.
`GET/v1/artifacts/page`	Page artifacts	Bounded policy-filtered page with a resource-typed opaque cursor.
`POST/v1/artifacts/{id}/replay`	Computational replay	Re-executes retained level-3 plan steps and compares output hashes.
`POST/v1/artifacts/{id}/revalidate`	Check current validity	Atomically updates semantic and replay validity with an audit and outbox event.
`GET/v1/claims/{id}`	Inspect a claim	Exact payload, independent validity dimensions, and support edges.
`POST/v1/reviews`	Approve, reject, or correct	Idempotent scoped review; feedback, claim state, audit, job, and outbox commit together.
`GET/POST/v1/jobs`	Inspect or enqueue work	Administrator-only durable jobs with idempotent enqueue.
`GET/v1/audit`	Inspect audit history	Administrator-only, tenant-scoped and bounded.
`GET/v1/metrics`	Inspect operations	Administrator-only task/recovery counters and latency buckets.
`POST/v1/retention`	Set retention or legal hold	Administrator-only idempotent policy command.
`POST/v1/retention/memory/{id}/erase`	Erase due memory	Administrator-only, legal-hold-aware atomic erasure and proof.
`GET/v1/connectors/health`	Probe the source	Administrator-only connector status.
`POST/v1/source-events/process`	Process source changes	Administrator-only bounded invalidation entry point.

Table 6: Minimum API surface. Administrative configuration and connector secrets use separate privileged interfaces.

- dissemination: `artifact.publication_requested`, `artifact.published`, `artifact.publication_failed`;
- workflow: `review.completed`, `job.dead_lettered`.

14 Security and Multi-Tenancy

14.1 Trust Boundaries

Retrieved content, model output, user-provided text, and external tool output are untrusted. Identity tokens, policy decisions, capabilities, source credentials, verification rules, commit logic, and audit events remain outside model text. A successful prompt injection may influence a proposed plan; it must not grant a capability, reveal filtered data, write high-authority memory, or publish restricted output.

Threat	Preventive control	Detection/recovery
Cross-tenant leakage	Tenant key on every row, database row-level security, tenant-scoped indexes/keys/caches.	Canary records, access-log analytics, immediate key and token revocation.
Indirect prompt injection	Evidence/policy separation, capability checks, sandboxed tools, output validation.	Malicious-content flags, plan diffs, red-team corpus, quarantine.
Permission-revocation race	Policy epoch and permission revalidation at commit.	Abort commit, invalidate affected claims, audit event.
Memory poisoning	Authority tiers, source namespaces, review gates, immutable versions.	Conflict alerts, provenance inspection, rollback by supersession.
Credential exposure	Secret manager, worker-side credential exchange, no secrets in prompts/logs.	Automated scanning, short TTL, rotation and incident runbook.
Unsafe SQL or code	AST checks, read-only roles, isolated workers, network/resource limits.	Kill switch, query tags, execution audit, dead-letter evidence.
Provenance leakage	Permissioned graph traversal and redacted support views.	Link-access tests, export review, access anomaly alerts.
Denial of service	Admission quotas, budgets, concurrency limits, fan-out controls, circuit breakers.	Queue depth/latency alerts, tenant isolation, graceful degradation.

Table 7: Security is enforced by external controls, not by asking the model to follow policy.

14.2 Policy Decision Contract

MUST: Policy is declarative and versioned behind one engine-neutral interface:

$$evaluate(subject, action, resource, task, environment, epoch) \rightarrow decision + obligations + policy\_refs.$$

The persisted result includes allow/deny, row and column constraints, redactions, review and retention obligations, applied policy versions, input-attribute hash, and a caller-safe explanation. Policies may depend on classified result metadata but not arbitrary model prose. Owners author policy drafts; a separate authorized reviewer activates them after simulation against stored decision fixtures. One epoch identifies an immutable set of policy versions. Runtime caches key on tenant, subject attributes, action, resource version, and epoch; epoch activation evicts the prior namespace. The interface permits Rego, Cedar, or a typed internal evaluator, but product code cannot embed bypass policy in prompts or route handlers.

14.3 Tenant Isolation

MUST: Tenant identity appears in every primary key and every foreign-key path through composite keys; no object ID is globally addressable without tenant binding. PostgreSQL row-level security is forced for runtime tables, and runtime roles cannot use `BYPASSRLS` or own those tables. Migration and runtime roles are separate. Application repositories repeat tenant predicates; property tests cover cross-tenant joins. Cache keys, indexes, object prefixes, encryption context, queues, capabilities, jobs, and audit queries are tenant-scoped. Canary rows and automated database-policy checks continuously detect bypass. Offer dedicated databases, worker pools, and customer-managed keys when required. Every background job restores tenant context explicitly; missing tenant context is a hard failure.

14.4 Privacy and Governance

Minimize retained raw rows. Support field classification, masking, purpose limitation, regional storage, retention, legal hold, export, and erasure. Erasure must update both primary records and derived artifacts while retaining only the minimum audit proof permitted by policy. High-risk model providers receive only the bounded, redacted context allowed by tenant configuration.

15 Deployment and Scalability Architecture

15.1 Initial Production Topology

Deploy the modular monolith as stateless API/runtime replicas behind a load balancer. Use managed PostgreSQL with point-in-time recovery, object storage with versioning, a managed queue, isolated worker pools, secret manager, identity provider, metrics/logs/traces backend, and a private network path to customer sources. Separate development, staging, and production accounts. Infrastructure is declarative and reviewed.

Run interactive and scheduled workloads in separate queues. Partition workers by tool type and risk. Use A-TXN IDs and idempotency keys across retries. Database transactions own state transitions; queue acknowledgements occur only after durable state is written. Long operations renew leases and checkpoints.

15.2 When to Extract Services

Do not distribute the system prematurely. Extract a component only when one of these conditions is measured: tool workers require a stronger isolation boundary; connector traffic has independent scaling; context indexes exceed primary-database capacity; invalidation fan-out interferes with interactive latency; enterprise deployment requires separate data planes; or team ownership demands an independent release cycle. Preserve the same typed contracts when extracting.

15.3 Scaling Strategy

Scale persistent memory and model context independently. Keep structured filters in the database, move large blobs to object storage, and introduce a dedicated lexical/vector candidate index behind the memory interface when needed. Shard by

tenant before sharding within a tenant. Maintain reverse dependency indexes for invalidation. Cache only version-addressed, permission-scoped results; policy-epoch changes evict them.

Capacity planning tracks active tenants, memory objects, version rate, dependency edges, scheduled tasks, tool concurrency, result bytes, model tokens, p50/p95/p99 control-path latency, queue age, and invalidation fan-out. Load tests include one-million-object tenants, policy churn, source delay, and noisy neighbors.

16 Operations, Reliability, and Support

16.1 Service Objectives

Initial paid-pilot objectives are 99.5% API availability, 99% successful scheduled runs excluding declared source outages, p95 admission/context compilation below two seconds at the supported scale, zero confirmed cross-tenant or permission leaks, and 100% of final artifacts with complete claim/evidence manifests. Objectives tighten only after error budgets and workload shape are understood.

16.2 Observability

Every request carries request, A-TXN, tenant, job, plan, execution, artifact, publication, and trace IDs. Metrics cover task outcomes, verifier results by rule, context role coverage, retrieval leaks, repair counts, review backlog, capability denials, query cost, model cost, queue age, source health, invalidation lag, publication acknowledgment, replay success, and storage growth. Logs are structured, redacted, immutable according to policy, and linked to traces. Customer-visible status distinguishes AMOS failure from source or provider failure.

16.3 Runbooks

Ship runbooks for source authentication failure, permission leak suspicion, policy epoch mismatch, stuck transaction, worker compromise, runaway query, queue backlog, database failover, object-store mismatch, invalidation storm, replay mismatch, model-provider outage, and data erasure. Each runbook specifies trigger, containment, customer impact, evidence preservation, recovery, owner, and post-incident tests.

Backups are restored in staging on a schedule. Target RPO is five minutes and RTO is one hour for the control plane during paid pilot, with a longer declared window for reconstructible artifacts. Disaster-recovery drills verify database, object, queue, key, and connector configuration recovery together.

17 Testing and Quality Gates

Testing mirrors the architecture rather than only happy-path answers.

The current crate contains 61 passing unit, API, end-to-end, security, concurrency, migration, failure-injection, privacy, and recovery tests. In addition to the original controls, they cover crash recovery at fourteen lifecycle checkpoints, checksummed schema migration, durable connector cursors, full capability rebinding, SQL cancellation and incremental byte limits, transitive invalidation continuations, persisted replay comparisons, outbox retry/dead-letter delivery, lost object-promotion acknowledgments, retention/legal hold/erasure, opaque cursors, request limits, and browser security headers. Appendix I maps the named tests to their control boundaries. Production database isolation, customer connector conformance, KMS, worker containers, and disaster recovery remain infrastructure release gates rather than claims implied by this suite.

17.1 Definition of Done for a Feature

A feature is done only when its typed contract, migration, permissions, audit events, metrics, failure modes, idempotency behavior, retention behavior, tests, operator runbook, and user-facing status are implemented. Model prompt changes alone do not constitute a product feature.

Layer	Required tests	Release gate
Unit/property	Time intervals, authority ordering, supersession, policy evaluation, state transitions, CAS conflicts, fencing, hash stability.	Deterministic and exhaustive boundary cases; mutation/property tests for core invariants.
Connector contract	Versions, revocation, pagination, retry, cursor recovery, access denial, rate limit, source deletion.	Same suite passes every connector; no adapter-specific bypass.
Context compiler	Role completeness, permission-first filtering, conflicts, ambiguity, budget pressure, compaction lineage.	Zero restricted/superseded selections; mandatory roles never silently dropped.
Verifier	Valid/invalid SQL corpus, metrics, schema drift, freshness, unsafe functions, claim support.	Published precision/recall thresholds and zero acceptance of seeded critical violations.
End to end	Request, schedule, review, feedback, replay, invalidation, abort/retry, artifact export.	Every terminal outcome and recovery path exercised with persisted evidence.
Security	Cross-tenant, prompt injection, poisoning, capability replay, secret leakage, provenance inference, DoS.	Zero critical leaks; red-team findings have regression tests.
Reliability	Worker kill, stale lease, database failover, duplicate events, object promotion, lost publication acknowledgment, source timeout, model outage.	No duplicate publication; fenced, idempotent recovery preserves audit trail.
Performance	Memory/edge growth, concurrency, churn, fan-out, noisy neighbor, queue saturation.	SLOs hold at declared paid-pilot limits with recorded capacity margin.
Human review	Evidence usability, defect detection, correction scope, audit time.	Reviewers can locate support and submit bounded corrections without engineer help.

Table 8: A release is not qualified because a demo answer looks correct; each control boundary must pass its own oracle.

18 Step-by-Step Build Plan

The team advances by exit gates, not calendar optimism.

Phase	Product state	Build	Exit gate
0	Architecture foundation	Freeze Specifications A–F, generated schemas, domain types, migrations, A-TXN state machine, audit log, tenant/identity skeleton, CI, local fixtures, one-command environment.	Contract examples validate; all invariants have executable tests; a failed step is inspectable and idempotently retryable.
1	Governed memory	Memory/version schema, authority/time/supersession, PostgreSQL search, permission policy, seed tooling, admin inspection.	Permission, time, authority, and supersession property tests pass; no destructive update path.
2	One connector and context	Certified C1 warehouse adapter, versioned task definition, required roles, context compiler, manifest, permission-first search, ambiguity handling.	Frozen task fixtures always include required roles; zero restricted or superseded retrievals; connector certification passes.
3	Governed answer	Typed planner, SQL/statistics/chart workers, capability claims, AST verifier, structured claims, result hashes, report renderer, task API.	At least 90% safe-and-correct completion on frozen pilot tasks; every numeric claim resolves to evidence.
4	Persistent analyst	Review UI/API, scoped feedback, multidimensional claim validity, replay levels 1–2, schedules, bounded invalidation.	Four weekly shadow cycles complete; corrections affect the next applicable run; validity dimensions change correctly.

Phase	Product state	Build	Exit gate
5	Paid pilot	SSO/RBAC, defense-in-depth tenancy, object finalization, idempotent publication adapter, durable queues, observability, backups, runbooks, quotas.	Three design partners; zero critical security findings; lost-acknowledgment and recovery drills pass; onboarding under ten business days.
6	Platform API	Stable SDK, connector kit, application templates, model routing, admin controls, retention/erasure, exportable audit packages.	Two independent applications use the same memory/runtime without bespoke core changes.
7	Scale and enterprise	Hybrid indexes, worker/service extraction where measured, dedicated tenancy options, regional deployment, customer-managed keys.	SLOs pass at target tenant/edge/concurrency scale; noisy-neighbor and failover tests pass.
8	Controlled autonomy	Policy-scoped auto-publication, dependency-driven re-analysis, portfolio scheduling, advanced statistics, rollback controls.	Each autonomous task class has a separate safety case, customer approval, monitoring, kill switch, and rollback drill.

Table 9: Implementation sequence from empty repository to controlled autonomy.

18.1 Reference Repository Shape

Preserve architectural boundaries in code before extracting network services. The current Rust modular monolith uses the following repository shape:

```
src/domain.rs              IDs, values, states, claims, jobs, results
src/auth.rs                identity-provider port and explicit demo provider
src/policy.rs              memory, task, tool, review, and read authorization
src/store.rs               schema, repositories, CAS, atomic commits, outbox
src/memory.rs              write, retrieve, reconcile, supersede, compact
src/context.rs             required-role compiler and manifest construction
src/connectors.rs          connector port and SQLite warehouse adapter
src/workers.rs             capability issuer and SQL/statistics/chart workers
src/verification.rs        SQL AST, schema, metric, freshness, claim checks
src/evidence.rs            citations, review feedback, invalidation entry
src/scheduler.rs           leased jobs, fencing, retry, dead-letter behavior
src/runtime.rs             complete A-TXN orchestration and replay
src/api.rs                 Axum /v1 API and four server-rendered UI surfaces
src/seed.rs                deterministic governed-memory and warehouse fixture
src/main.rs                fail-closed demo CLI and HTTP server
tests/                     API, security, concurrency, and end-to-end contracts
docs/                      traceability and independent evaluation protocols
artifacts/evaluation/      frozen result bundles and integrity hashes
```

The public library re-exports `AmosRuntime`, `RuntimeConfig`, `AmosError`, and `Result`, while exposing each module for embedding applications. Domain code imports serialization, hashing, time, and ID libraries but no web framework, database driver, model SDK, or connector implementation. The runtime composes typed services; transactional state changes and their outbox events commit together. The current `Store` is the only table-writing component. Production architecture tests must additionally enforce dependency direction, table ownership, tenant-scoped repositories, and credential absence from model-visible values.

18.2 First Vertical Slice

Use a recurring executive business review selected from a design partner's actual analyst queue as the first commercial workflow. It must be economically meaningful, bounded, repeatable, and reviewable without defining AMOS around one department or dataset. Limit the alpha to one metric family, one warehouse, three query shapes, a small approved chart set, one report template, one presentation template, and one reviewer role. Shadow the existing analyst process before replacing any deliverable.

The vertical slice is complete when a user request or schedule produces: an admitted transaction; local model plan; versioned context manifest; verified SQL executions; result and chart hashes; structured claims; review-aware report; editable presentation deck; claim dependencies; replay package; audit trail; and explicit terminal outcome. A fluent answer without these records and completed artifacts is not an AMOS product slice.

18.3 Team and Ownership

A five-person initial team is sufficient: founding architect/product lead; memory/connectors engineer; runtime/tools engineer; product/review engineer; and design-partner/data lead. Assign one named owner for security and operations even if the role is fractional. Every architectural component in Table 3 has a code owner and operational owner before paid pilot.

19 Current Rust Implementation to Production

As completed on 2026-07-19, the repository is a Rust 2024 crate at version 0.2.0. The original audited inventory of 236 production declarations remains present or is intentionally superseded by cleaner typed recovery, migration, publication, retention, dispatch, observability, and bounded-execution APIs. The control schema is at checksummed version 6 and the executable suite contains 61 tests plus a reproducible capacity benchmark. Formatting, Clippy with warnings denied, debug and release tests, optimized build, and Rustdoc are release gates. Appendix I records exact local evidence without treating external infrastructure as implemented.

The implementation completes a legacy local payment-health control-layer fixture: indexed permission-first memory, exact role compilation, durable source observation events, deterministic planning, a frozen SQL subset, constant-time verified capabilities, cancellable bounded execution, independently verified claims and charts, atomic evidence, review, transitive invalidation, fenced workers and dispatch, state-driven recovery, persisted level-3 replay comparisons, hash-checked object promotion, metrics, and privacy erasure. It is not the complete analyst product and does not yet include a local model, domain-neutral workflow, editable slides, production connectors, managed PostgreSQL/RLS, enterprise identity/KMS, isolated worker infrastructure, or cloud publication topology.

20 Low-Level Execution and Performance Engineering

20.1 Hot-Path Cost Model

Let M be active tenant memory objects, T the number of normalized request terms, K the context-item budget, S the plan-step count, N_q the SQL AST size, R returned rows, C returned columns, B_o serialized output bytes, E dependency edges, and P the invalidation page size. The current implementation has the following dominant costs:

Path	Current bound	Engineering consequence
Memory retrieval	Indexed FTS candidate generation plus $O(C \log K)$ bounded ranking for $C \leq 2000$	Tenant, lifecycle, interval, type, and all-label permission predicates execute in SQLite before objects enter Rust; a bounded heap retains only top-K. The capacity benchmark reports p50/p95/p99 and enforces a release p95 threshold.

Path	Current bound	Engineering consequence
Reconciliation	$O(M \log M)$ time, $O(M)$ grouping	Objects group by type and logical key, then sort by authority and record time. A preordered database candidate stream or per-key best-candidate reduction removes full-group sorts.
Context compilation	Retrieval plus $O(KR_d)$ role scans	Each required/optional role scans reconciled candidates; R_d is role count. Index candidates by `(MemoryType, role)` inside the compile call when role or candidate counts grow.
SQL verification	Approximately $O(N_q \log C_s)$	`sqlparser` walks the AST and inserts references into ordered sets; schema/filter checks follow. This is bounded by query length and selected schema size, not warehouse rows.
SQL execution	$O(RC + B_o)$ time and $O(B_o)$ memory	Each row is encoded and charged against the byte budget before it enters the result vector; the progress handler enforces time and explicit cancellation. Production remote workers should stream large outputs directly to content-addressed object storage.
Evidence commit	$O(C_l + E)$ statements	Claims and edges are inserted individually inside one immediate transaction. PostgreSQL should use prepared/batched inserts while retaining a single atomic commit.
Invalidation	$O(\log E + P)$ indexed lookup and updates	Reverse-edge indexes bound a page to at most 250 claims; each changed claim creates a job and outbox event. A leased worker consumes the durable cursor, retains the root visited set, and idempotently schedules another page until traversal completes or its quota fails closed.
Job acquisition	Indexed ready/expired-job lookup plus one CAS update	SQLite serializes the immediate write transaction. PostgreSQL should use `FOR UPDATE SKIP LOCKED`, retain monotonically increasing fences, and keep lease mutation atomic with events.

Table 11: Asymptotic and allocation model of the current Rust implementation.

20.2 Concurrency, Allocation, and I/O

`Store` wraps one `rusqlite::Connection` in `Arc<Mutex<...>>`; cloned services share it and therefore serialize database access within a runtime. WAL permits useful concurrency across separately opened stores. Immediate write transactions, state sequences, and fencing tokens prevent lost updates. All asynchronous-facing database and warehouse work enters an eight-permit `spawn_blocking` lane, so the Tokio executor is not blocked and admission is bounded. Production PostgreSQL adapters still require a measured connection pool, lock/queue telemetry, and the same CAS predicates.

The SQL worker opens a connection per execution, enables `query_only`, disables trusted schema, installs a SQLite progress callback, and materializes only rows that fit the incremental row and byte budgets. Deadline and explicit cancellation both interrupt the statement. Capabilities expire after 60 seconds; constant-time HMAC verification binds issuer, audience, tenant, A-TXN, plan, step, subject, tool, source, operations, relations, limits, policy epoch, fence, and token ID at both connector and worker boundaries. Production remote workers must additionally obtain short-lived source credentials without ambient process access.

Hashes are SHA-256 over fallibly serialized values; serialization failures propagate and cannot alias empty content. Runtime entity IDs are typed UUIDv7 values, while durable checkpoint and event identities use typed prefixes plus 128 bits of the serialized-value digest. JSON payloads keep the local adapter simple; indexed scalar columns remain authoritative for

Area	Implemented Rust MVP	Production completion work
State store	SQLite WAL; typed/indexed rows; atomic evidence, review, validity, retention, erasure, publication, outbox, and job effects; checksummed schema ledger.	PostgreSQL forced RLS, HA/PITR, online DDL, regional backup and restore.
Transactions	Normative A-TXN guard, CAS, immutable admission, version vectors, fences, deterministic checkpoint identities, and state-driven recovery tested across fourteen boundaries.	Distributed controller election and remote worker placement.
Connectors	Typed port and SQLite adapter with stable versions, capability-bound reads, validation, durable deduplicated cursor events, restart recovery, and health.	Customer warehouse/IAM/catalog/document credentials, quotas, retry certification, and C1/C2 fixtures.
Runtime	Deterministic legacy reference workflow, bounded repair, statistics/chart, claims, review/revalidation, durable replay evidence, object promotion, and a bounded blocking lane.	Local Gemma 4 provider, domain-neutral task configuration, report/slide planning, artifact compilation, and isolated worker services where measured.
Verification	Frozen AST subset; schema/metric/time/function rules; cancellation; incremental limits; referential, numeric, statistical, and chart-hash recomputation.	Calibrated task-specific rule packs and driver conformance beyond SQLite.
Provenance	Typed claims/edges, six validity dimensions, transitive quota traversal, consumed continuation cursors, replay comparisons, retention and erasure receipts.	Signed customer export formats and long-term data-platform version leases.
API/UI	Authenticated `/v1`, explicit mutation keys, stable errors, request/correlation IDs, 1 MiB cap, opaque artifact cursors, OpenAPI 3.1 endpoint, CSP and security headers.	Enterprise SSO/session adapter, SDK distribution, tenant quota configuration, and streaming progress.
Operations	Leased job and outbox loops, fence/owner/expiry checks, retry/dead letter, shutdown, metrics, structured traces, audit, connector and migration health.	Telemetry backend, SLO alerts, on-call, broker, backup/DR and regional deployment.
Security	Fail-closed binary/provider port, tenant/epoch/label enforcement, constant-time full-bound capabilities, query-only SQL, cancellation, HTML escaping and browser headers.	Enterprise OIDC, database RLS, KMS keys, container/VM sandbox, egress and credential broker.

Table 10: The production plan extends verified Rust behavior without presenting local adapters as managed infrastructure.

concurrency, filtering, and lifecycle state.

20.3 Performance Gates

Optimization is accepted only with a workload, correctness oracle, and before/after evidence. The paid-pilot gate records p50/p95/p99 for admission, context compilation, verification, execution, evidence commit, review commit, job acquisition, invalidation page, and replay; peak resident memory; allocations per selected context item and output row; database lock/pool wait; rows and bytes scanned/returned; and outbox/job age. Benchmarks cover 10^4, 10^5, and 10^6 memory objects, skewed permission labels, conflicting authority versions, large SQL outputs, 100–250-claim invalidation pages, expired lease storms, duplicate admissions/reviews, and concurrent tenants. No optimization may weaken permission-before-ranking, source-version immutability, compare-and-swap, fencing, atomic evidence, or claim-level provenance.

21 Architecture Decisions and Non-Goals

Key decisions are explicit:

- build a modular monolith first, with internal service contracts and one authoritative transaction database;
- keep raw data in source systems and move bounded results or retained snapshots only under policy;
- enforce permissions before retrieval ranking and again at execution and commit;
- treat models as untrusted, replaceable processors with no ambient credentials;
- use optimistic cross-source validation rather than claim nonexistent global ACID guarantees;
- attach provenance at claim granularity and define replay levels precisely;
- construct typed claims before prose and persist validity dimensions separately;
- separate evidence commit, object finalization, and external dissemination;
- gate causal and high-impact conclusions on human or approved methodological review;
- add autonomy per task class, never as a platform-wide switch.

DEFERRED: General Python, free-form claim extraction, vector retrieval, and automatic external publication. **NON-GOAL:** Universal enterprise knowledge management, unrestricted natural-language access to all data, arbitrary production writes, perfect model reproducibility, automatic causal discovery, and replacing warehouses, semantic layers, catalogs, lineage systems, or BI tools.

22 Harsh Qualification Checklist

Before claiming that AMOS is ready for a paid pilot, answer yes to every item:

1. Can an operator reconstruct why every selected memory object entered the context and why every omitted mandatory role caused a visible outcome?
2. Can permission revocation between retrieval and publication prevent commit without relying on the model?
3. Can every numeric statement in a final artifact resolve to query, result hash, data version, metric, schema, and verification?
4. Can a reviewer correction affect the next applicable task while the original artifact remains immutable?
5. Can a metric or schema change identify and reclassify every dependent claim?
6. Can duplicate requests, events, or retries occur without duplicate publication?
7. Can a worker be killed or compromised without exposing long-lived credentials or corrupting transaction history?
8. Can the team restore database, objects, queue state, keys, and connector configuration within the declared RTO?
9. Can a tenant export its audit package and enforce retention or erasure without another tenant being observable?
10. Can a lost publication acknowledgment be retried without creating a duplicate external artifact?
11. Do composite tenant keys, forced RLS, repositories, caches, object paths, and capabilities all reject cross-tenant access?
12. Can model provider, retrieval backend, or worker implementation change without changing the memory and A-TXN contracts?
13. Do load, security, connector, failure-injection, and human-review tests pass at the declared supported scale?
14. Does the product save measurable analyst/reviewer time while maintaining zero confirmed critical permission leaks?

A no answer is an architectural gap, not documentation debt. The corresponding phase remains incomplete until the gap has an owner, contract, test, metric, and runbook.

23 Conclusion

AMOS becomes a product when persistent company knowledge, model reasoning, tool execution, verification, human judgment, and artifacts are joined by one enforceable operating contract. Typed analytical memory provides durable state. The bounded context compiler supplies only valid and permitted working memory. A-TXN makes admission, version observation, capabilities, validation, commit, dissemination, replay, and invalidation explicit. Claim dependencies make conclusions inspectable and change-aware. External enforcement keeps models useful without making them trusted authorities.

The implementation path is deliberately practical: build one governed read-only workflow in a modular monolith; make every outcome and dependency observable; add durable review, scheduling, tenancy, operations, and connector breadth; then introduce platform APIs and autonomy only when phase-specific gates pass. If the team follows the contracts and qualification checklist in this guide, it can build AMOS end to end without leaving the hardest correctness, security, and operational decisions implicit.

A Specification A: A-TXN, Concurrency, and Publication

This specification freezes the internal state machine and the boundary between evidence commit, object finalization, and external dissemination. Generated enums and transition tests must match this section.

A.1 Persistent State Machine

Current state	Allowed next state	Required guard and durable effect
admitted	observing, rejected	Identity, tenant, task definition, risk, budgets, policy epoch, and request idempotency key are fixed.
observing	selecting, needs_review, aborted	Required connector observations and attainable consistency classes are stored.
selecting	planning, needs_review, rejected	Manifest covers every required role; omissions and conflicts are explicit.
planning	executing, needs_review, rejected	Typed plan validates against the task definition and budget.
executing	composing, repairing, aborted	Every execution uses the current lease fence and a valid capability; results are hash-addressed.
repairing	executing, needs_review, rejected	Failure is in an allowed repair class and the attempt budget remains.
composing	verifying, aborted	Typed claims and artifact inputs are stored before prose render.
verifying	revalidating, repairing, needs_review, rejected	Versioned verifier results cover every required rule and claim.
revalidating	evidence_committed, repairing, needs_review, rejected, aborted	Policy epoch, C1 observations, C2 leases, task version, and publication obligations still hold.
evidence_committed	object_finalizing, publication_pending, needs_review	Claims, dependencies, replay descriptor, audit event, and outbox records commit atomically.
object_finalizing	publication_pending, object_failed	Temporary objects promote by content hash; mismatch never mutates committed evidence.
publication_pending	published, publication_failed	Adapter receives stable publication ID and stores external acknowledgment or classified failure.
published	revocation_pending	Only policy, review, or invalidation may request revocation; evidence remains immutable.
revocation_pending	revoked, publication_failed	Adapter records target acknowledgment; retries retain the same publication ID.
needs_review	verifying, revalidating, rejected	A current authorized review supplies the exact missing obligation.
rejected, aborted, revoked	none	Terminal outcome and evidence are immutable; a retry creates or resumes only through an explicit retry record.

Table 12: Allowed A-TXN and publication transitions. A generated transition matrix is the implementation source of truth.

A.2 Isolation, Compare-and-Swap, and Fencing

MUST: Each database interaction is a short PostgreSQL `READ COMMITTED` transaction. Transition code performs compare-and-swap on tenant, A-TXN ID, current state, and `state_seq`; success increments the sequence. A zero-row update means a concurrency conflict, not success.

```
UPDATE atxn_transactions
SET state = :next_state,
    state_seq = state_seq + 1,
    updated_at = clock_timestamp()
WHERE tenant_id = :tenant_id
  AND atxn_id = :atxn_id
  AND state = :expected_state
  AND state_seq = :expected_seq
  AND terminal = false;
```

Job acquisition uses `SELECT ... FOR UPDATE SKIP LOCKED`, increments `fencing_token`, and commits before work begins. Every checkpoint, execution, object promotion, outbox completion, and publication attempt compares that fence. A renewed lease never reuses an old fence. Terminal records reject updates at both repository and database layers. Review, invalidation, and publication commands lock the affected artifact row and compare its validity sequence before appending a new state.

MUST: Every mutation and its outbox event share one database commit. Event consumers deduplicate by tenant and event ID. Side effects use the stable key `tenant/atxn/step/effect`; publication uses `tenant/publication_id`. Duplicate requests return the original resource when the request hash matches and return `IDEMPOTENCY_CONFLICT` when it differs.

A.3 Normal Task and Commit-Time Revocation

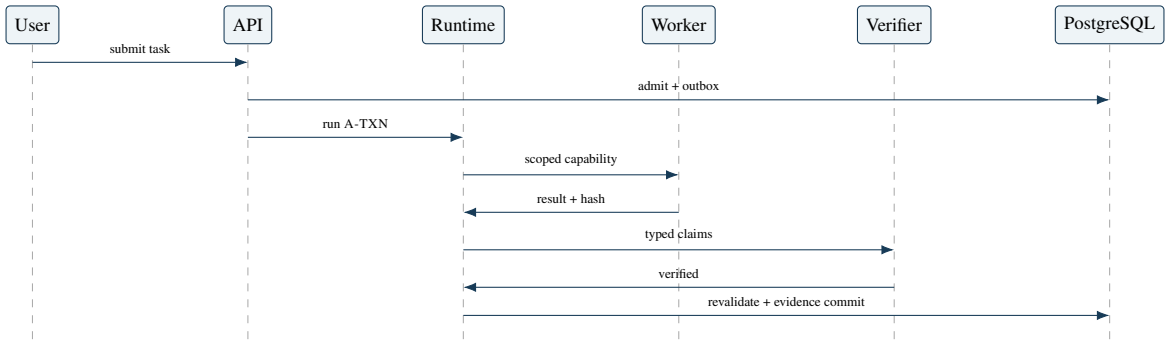

Figure 2: Normal task lifecycle. Durable state, not streamed model text, advances the product.

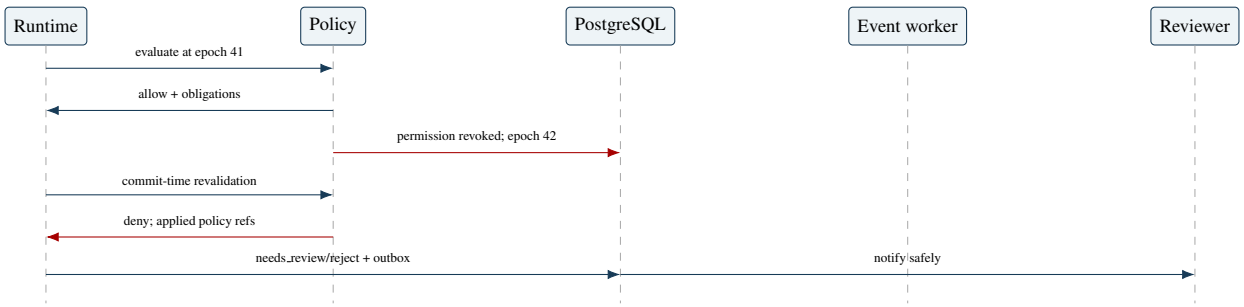

Figure 3: Commit-time policy revocation prevents publication without relying on the model.

A.4 External Publication Protocol

The publication adapter interface is `prepare`, `publish`, `status`, and `revoke`. `prepare` validates destination, audience, current policy, and payload hash without a side effect. `publish` accepts a stable publication ID. `status` reconciles lost acknowledgment. `revoke` is idempotent and records whether the target can truly retract or only append a correction. Permanent failures move to dead letter with operator ownership; retryable failures use capped exponential backoff and preserve the same ID.

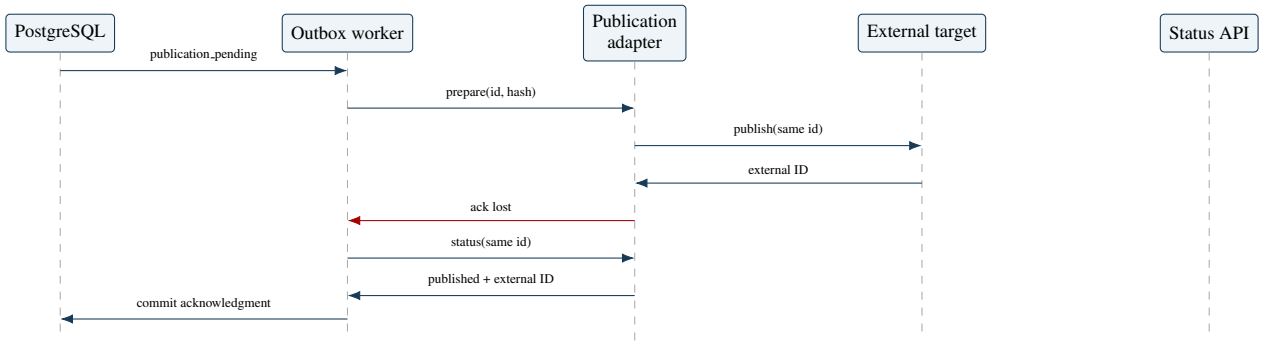

Figure 4: External publication recovers a lost acknowledgment without duplicating the target artifact.

B Specification B: Persistent Model and Database Invariants

B.1 Keys, Constraints, and Roles

Invariant	Implementation rule
Tenant binding	Every primary key and foreign key includes `tenant_id`. Lookups require tenant plus local ID. Composite foreign keys prevent cross-tenant references.
Forced RLS	Runtime tables enable and force row-level security. Runtime roles neither own tables nor hold `BYPASSRLS`; migration roles are separate and unavailable to the application.
Immutability	Memory versions, executions, evidence commits, claims, dependencies, reviews, and audit events are append-only. Correction uses supersession or a new validity record.
State checks	Database checks constrain enum values, nonnegative budgets, effective intervals, terminal flags, and publication transition pairs. Repository CAS supplies transition legality.
Version identity	Unique constraints cover tenant, source, logical key, source version, and content hash where appropriate. Equal version with unequal content is a connector incident.
Idempotency	Unique tenant-scoped keys exist for requests, events, executions, object promotions, jobs, reviews, and publication attempts.
Ownership	Each package owns its tables; other packages use typed ports. Direct cross-package writes fail architecture tests and database grants.
Audit	Sensitive changes append actor, action, target, decision, policy epoch, request, A-TXN, correlation, and time. Audit payloads are redacted but tamper-evident.

Table 13: Database invariants are repeated at schema, database-role, repository, and test layers.

Required indexes begin with tenant and cover active logical version, effective interval, authority, permission label, source version, A-TXN state sequence, job readiness/lease, publication state, and reverse dependency target. A migration that changes a state or contract ships with forward DDL, bounded backfill, compatibility window, verification query, rollback or roll-forward plan, and observed capacity cost. Destructive column removal waits until all running application versions stop reading it.

B.2 Example A-TXN Record

```
{
  "tenant_id": "ten_01",
  "atxn_id": "atxn_01J2",
  "task_definition": {"type": "payment_health_review", "version": 3},
  "request_hash": "sha256:...",
  "subject_id": "usr_42",
  "risk": "material_internal",
  "state": "revalidating",
  "state_seq": 12,
  "policy_epoch": 41,
  "source_versions": {
    "metric:payment_failure_rate": "v3",
    "schema:payments": "etag:82ab",
    "snapshot:warehouse": "snap_2026_07_12T1900Z"
  },
  "manifest_id": "ctx_91",
  "plan_id": "plan_17",
  "fencing_token": 8,
  "outcome": null,
  "created_at": "2026-07-12T19:02:11Z",
  "updated_at": "2026-07-12T19:03:27Z"
}
```

B.3 Object Finalization and Retention

Temporary objects use a tenant prefix and a random upload ID. The evidence commit stores expected hash, size, media type, retention class, and final content-addressed key. Promotion copies or renames idempotently, verifies hash and encryption context, then appends finalization state. A mismatch quarantines the temporary object and leaves evidence inspectable but non-publishable. Garbage collection deletes unreferenced temporary objects only after the maximum retry window.

Retention evaluates each evidence class independently. Erasure creates a tombstone, removes primary and derived bytes, invalidates dependent summaries, and retains only the minimum policy-permitted proof. C2 leases record expiry and renewal; replay availability changes when a lease expires even if historical publication validity remains true.

C Specification C: Task Definitions and Context Manifests

C.1 Versioned Task Definition

```
task_type: payment_health_review
version: 3
status: approved
risk_class: material_internal
required_roles:
  - metric_definition
  - active_schema
  - data_snapshot
  - user_policy
  - review_policy
optional_roles:
  - prior_incident
  - deployment_event
  - reviewer_feedback
minimum_consistency:
  metric_definition: C1
  active_schema: C1
  data_snapshot: C2
  user_policy: C1
allowed_tools:
  - sql.readonly.v1
  - stats.rate_comparison.v1
  - chart.timeseries.v1
claim_types:
  - metric_value
  - metric_comparison
  - operational_recommendation
verifier_profile: payment_health.v2
publication_policy: human_review_required
budgets: {max_steps: 8, max_repairs: 2, max_rows: 50000}
artifact_schema: payment_health_report.v3
```

Task definitions are immutable after approval. A new version declares compatibility, migration behavior for schedules, fixture corpus, owner, reviewer, effective interval, and superseded version. Admission binds one version for the entire A-TXN. Dynamic policy may add obligations or deny execution but cannot silently remove required roles or verifier rules.

C.2 Example Context Manifest

```
{
  "manifest_id": "ctx_91",
  "tenant_id": "ten_01",
  "atxn_id": "atxn_01J2",
  "task_definition": "payment_health_review:v3",
  "policy_epoch": 41,
  "required_role_coverage": {
    "metric_definition": ["mem_metric_pfr_v3"],
    "active_schema": ["mem_schema_payments_82ab"],
    "data_snapshot": ["mem_snapshot_1900"],
    "user_policy": ["mem_policy_access_41"],
    "review_policy": ["mem_policy_review_9"]
  },
  "optional_selected": ["mem_incident_217"],
  "omissions": [{"role": "deployment_event", "reason": "no_authorized_candidate"}],
  "conflicts": [],
  "token_count": 6840,
  "source_versions": ["metric:v3", "schema:82ab", "snapshot:1900"],
  "ranking_trace_ref": "obj://ten_01/traces/ctx_91"
}
```

C.3 Selection and Compaction Rules

Candidate SQL always applies tenant, active status, effective-time overlap, coarse permission scope, and non-superseded filters before text ranking. Object authorization then evaluates the exact subject and policy epoch. The response reveals neither pre-authorization count nor restricted candidate identity. A cache entry is keyed by tenant, task version, subject-attribute hash, policy epoch, query hash, and source-version vector.

Required roles select valid governing objects before optional relevance ranking. Equal-authority conflicts block. Budget pressure removes optional evidence first. A compacted object inherits the strictest source permission and shortest retention, is non-governing, links all source versions, is invalidated on any source change, and cannot satisfy metric, schema, policy, or data-state roles.

D Specification D: Connector SDK and Certification

D.1 Typed Connector Interface

```
discover(scope, cursor?) -> Page<EntityRef>
observe(ref) -> Observation {
  source_version, observed_at, effective_interval,
  freshness, sensitivity, access_descriptor,
  consistency_class, retention_deadline?
}
read(ref, source_version, capability) -> BoundedHandle
validate(ref, source_version) -> Validation {same, current_version, reason}
subscribe(cursor) -> Page<SourceEvent>
health() -> Health {status, lag, rate_limit, degraded_capabilities}
```

Version identity must represent content or source transaction state strongly enough for the advertised class. Connector configuration declares whether updates overwrite identities, how deletions and permission changes surface, cache behavior, clock assumptions, pagination consistency, cursor expiry, and source outage semantics. Equal source version with different normalized content is a hard incident.

D.2 Certification Matrix

Behavior	C0	C1	C2	Oracle
Stable observation identity	required	required	required	Repeated observation without change returns equal version.
Delayed revalidation	—	required	required	Change after delay returns unequal version or unavailable.
Overwrite and deletion	observed	required	required	Fixture overwrite/deletion cannot validate old state.
Permission change	observed	required	required	Revocation becomes observable within declared bound.
Pagination/cursor recovery	required	required	required	No silent duplicate, omission, or cross-scope item.
Cache staleness bound	declared	enforced	enforced	Bypass/expiry test meets declared maximum.
Exact version read	—	optional	required	Old version reads exact bytes and checksum.
Retention lease	—	—	required	Deadline, renewal, expiry, and missing snapshot are deterministic.
Source unavailable	required	required	required	Returns classified unavailable, never a guessed validation.

Table 14: A connector earns its consistency class through conformance tests, not self-description.

D.3 Example Source-Change Event

```
{
  "event_id": "evt_src_882",
  "schema_version": 1,
  "type": "source.version.changed",
  "tenant_id": "ten_01",
  "source_id": "warehouse_primary",
  "subject": "schema:payments",
  "previous_version": "etag:82ab",
  "current_version": "etag:91cc",
  "change_kind": "schema",
  "occurred_at": "2026-07-12T20:00:00Z",
  "observed_at": "2026-07-12T20:00:09Z",
  "cursor": "cur_10921",
  "deduplication_key": "warehouse_primary/schema:payments/etag:91cc"
}
```

E Specification E: Workers, Capabilities, and Verification

E.1 Capability Claims

```
{
  "iss": "amos-runtime",
  "aud": "sql-worker",
  "tenant_id": "ten_01",
  "atxn_id": "atxn_01J2",
  "plan_id": "plan_17",
  "step_id": "step_03",
  "subject_id": "usr_42",
  "tool": "sql.readonly.v1",
  "source_id": "warehouse_primary",
  "operations": ["query"],
  "relations": ["analytics.payments"],
  "limits": {"seconds": 30, "rows": 50000, "bytes": 5000000},
  "policy_epoch": 41,
  "fencing_token": 8,
  "jti": "cap_7701",
  "nbf": 1783908200,
  "exp": 1783908260
}
```

Capabilities are signed by a dedicated issuer, audience-bound, short-lived, single-use for side-effecting operations, and exchanged only over authenticated worker channels. The worker fetches source credentials from the secret broker after capability validation; the model and runtime plan never receive them. Audit stores token ID and claims hash, not the bearer token.

E.2 Typed Plan and Verifier Result

```
{
  "plan_id": "plan_17",
  "task_definition": "payment_health_review:v3",
  "manifest_id": "ctx_91",
  "steps": [{
    "step_id": "step_03",
    "purpose": "compute current and baseline failure rates",
    "tool": "sql.readonly.v1",
    "input_object_ids": ["mem_metric_pfr_v3", "mem_schema_payments_82ab"],
    "parameter_schema": "payment_rate_query.v2",
    "expected_output_schema": "rate_comparison.v1",
    "timeout_seconds": 30,
    "limits": {"rows": 50000, "bytes": 5000000},
    "retry": {"max_attempts": 2, "repair_classes": ["COLUMN_SUPERSEDED"]},
    "verifier_profile": "payment_health.v2"
  }]
}
```

```
{
  "verification_id": "ver_551",
  "execution_id": "exec_123",
  "verifier_profile": "payment_health.v2",
  "profile_version": 2,
  "outcome": "pass",
  "checks": [
    {"rule_id": "SQL_READ_ONLY", "outcome": "pass"},
    {"rule_id": "METRIC_FILTERS", "outcome": "pass"},
    {"rule_id": "SCHEMA_VERSION", "outcome": "pass"},
    {"rule_id": "RESULT_DENOMINATOR", "outcome": "pass"}
  ],
  "warnings": [],
  "permitted_repair": null,
  "input_hash": "sha256:...",
  "created_at": "2026-07-12T19:03:11Z"
}
```

E.3 Worker Protocol and Deferred Python

The worker validates capability signature, audience, time, tenant, source, operation, policy epoch, and fence; reserves the execution idempotency key; obtains a short-lived source credential; validates the operation; executes within limits; writes result bytes under a temporary tenant key; commits output hash and classified status; destroys credentials; and acknowledges only after durable state. Cancellation and lease loss stop execution and prohibit commit.

DEFERRED: A general Python worker is not an alpha component. Its later safety case must define immutable runtime images, allowlisted packages and hashes, no ambient network or filesystem, secret absence, syscall and resource isolation, data-egress controls, deterministic seed/time behavior, result capture, vulnerability response, and replay claims.

E.4 Verifier Repair and Worker Crash Sequences

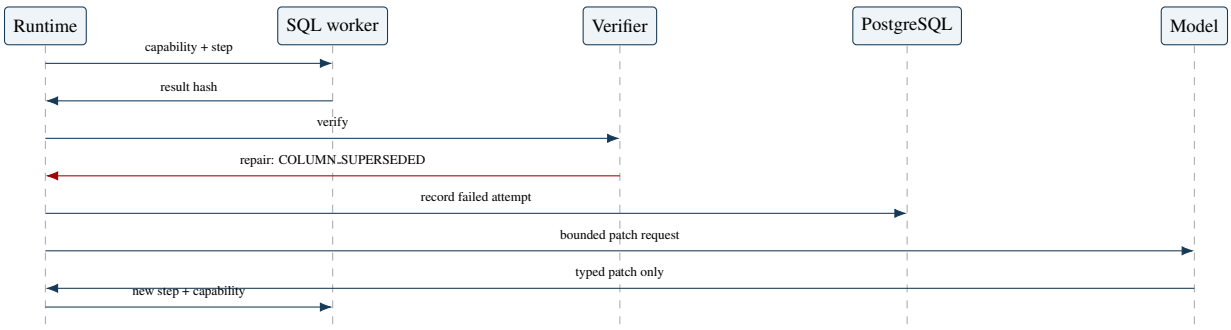

Figure 5: Verifier repair is bounded, typed, and preserves the failed attempt.

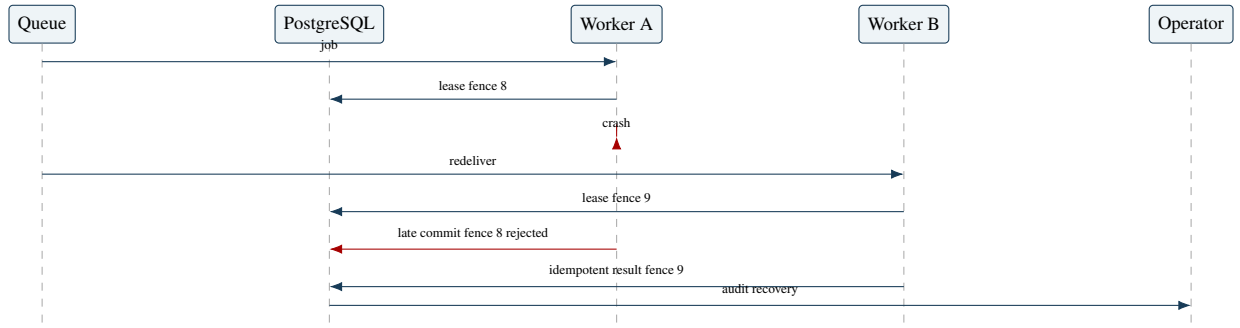

Figure 6: A worker crash cannot let a stale lease corrupt transaction history.

F Specification F: Claims, Review, Invalidation, and Replay

F.1 Structured Claim and Dependency Edge

```
{
  "tenant_id": "ten_01",
  "claim_id": "clm_501",
  "artifact_id": "art_700",
  "claim_type": "metric_comparison",
  "payload": {
    "metric_id": "payment_failure_rate:v3",
    "current_value": 0.074,
    "baseline_value": 0.022,
    "unit": "ratio",
    "window": {"start": "2026-07-05", "end": "2026-07-12"}
  },
  "support_execution_ids": ["exec_123"],
  "verification_ids": ["ver_551"],
  "publication_validity": "valid_at_publication",
  "semantic_validity": "current",
  "policy_visibility": "allowed",
  "replay_availability": "level_2",
  "review_state": "verified",
  "supersession_state": "active"
}
```

Dependency edge

```
{
  "edge_id": "edge_991",
  "tenant_id": "ten_01",
  "from": {"type": "claim", "id": "clm_501"},
  "relation": "computed_by",
  "to": {"type": "execution", "id": "exec_123"},
  "source_version": "snapshot:1900",
  "created_by_atxn": "atxn_01J2",
  "content_hash": "sha256:..."
}
```

F.2 Review Correction, Replay Package, and API Error

Review correction

```
{
  "review_id": "rev_881",
  "tenant_id": "ten_01",
  "artifact_id": "art_700",
  "claim_ids": ["clm_501"],
  "reviewer_id": "usr_finance_owner",
  "decision": "correct",
  "correction": {
    "type": "metric_filter",
    "value": "exclude_internal_accounts",
    "scope": "task:payment_health_review",
    "effective_from": "2026-07-13T00:00:00Z",
    "authority": "owner_approved"
  },
  "original_artifact_mutated": false
}
```

Replay package

```
{
  "package_id": "rpl_440",
  "tenant_id": "ten_01",
  "artifact_id": "art_700",
  "replay_level": 2,
  "manifest_id": "ctx_91",
  "plan_id": "plan_17",
  "execution_ids": ["exec_123"],
  "template": "payment_health_report:v3",
  "render_config_hash": "sha256:...",
  "retained_until": "2027-07-12T00:00:00Z",
  "expected_artifact_hash": "sha256:..."
}
```

API error response

```
{
  "request_id": "req_909",
  "atxn_id": "atxn_01J2",
  "error": {
    "code": "CONTEXT_REQUIRED_ROLE_MISSING",
    "message": "A required context role is unavailable.",
    "retryable": false,
    "safe_details": {"role": "active_schema"},
    "review_required": true
  }
}
```

F.3 Validity Transition Rules

Dimension	Representative states	Transition cause
Publication validity	draft, valid_at_publication, publication_failed, revoked	Evidence and review state at publication; external adapter acknowledgment; later revocation.
Semantic validity	current, pending_revalidation, stale, invalid	Metric/schema/data/document change and verifier outcome.
Policy visibility	allowed, redacted, denied	Current subject, audience, policy epoch, retention, or legal hold.
Replay availability	level_0–level_4, degraded, expired	Retained evidence, C2 lease, tool environment, template, and policy package.
Review state	unreviewed, needs_review, approved, corrected, rejected	Authorized review append; correction never overwrites prior evidence.
Supersession state	active, superseded, tombstoned	New artifact/claim version, owner action, or erasure policy.

Table 15: Validity dimensions remain independent so historical truth is not confused with current usability.

F.4 Review and Source-Change Sequences

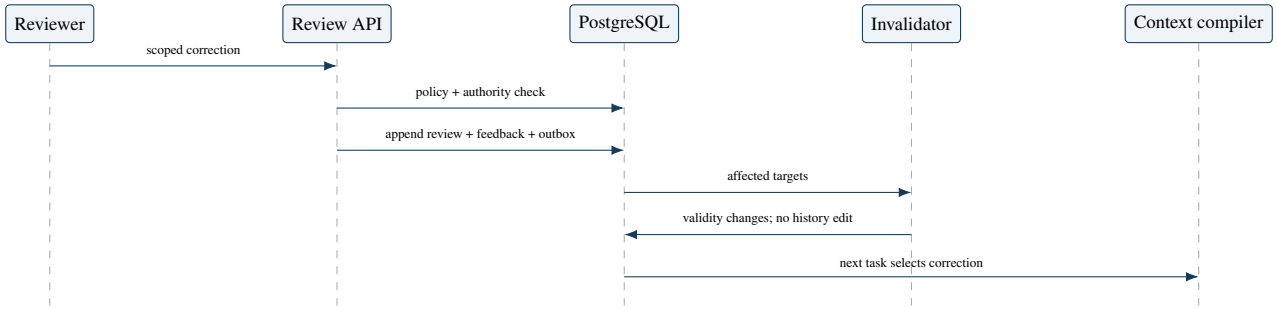

Figure 7: Human review creates governed feedback while preserving the original artifact.

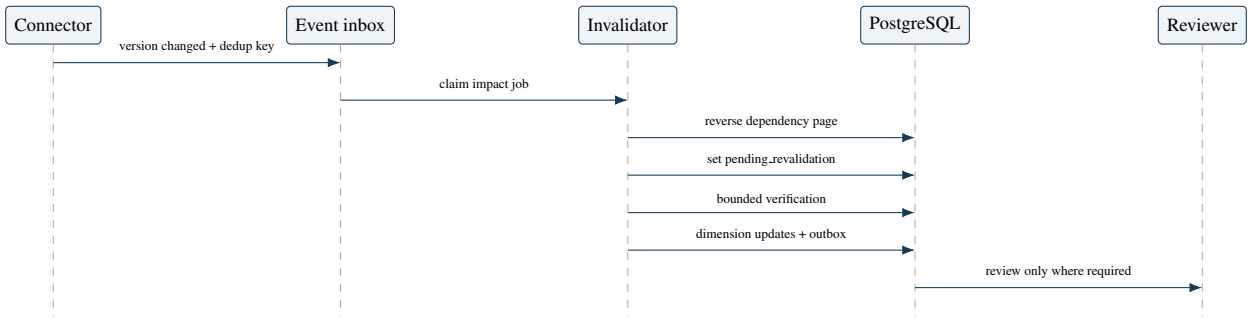

Figure 8: Source-change invalidation is deduplicated, paged, bounded, and dimension-specific.

F.5 Replay Sequence

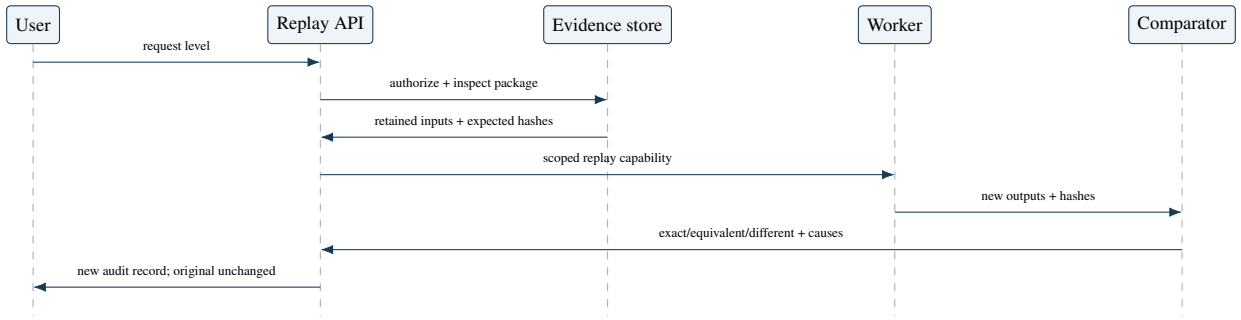

Figure 9: Replay creates new evidence and a comparison; it never rewrites the original run.

G Current Rust Implementation Model

G.1 Audit Baseline and Scope

This appendix is descriptive of the Rust working tree completed on 2026-07-19; Specifications A–F remain normative. The target product is an internally deployed analyst system that connects to company data and tools, answers business questions, performs verified analysis, and produces graphs, reports, and presentation slides. The implementation inventoried here is the control-layer reference fixture, not that complete product. It does not yet contain the local Gemma 4 analyst agent, domain-neutral configuration, or deterministic PPTX/PDF/spreadsheet compiler. The crate uses Rust 2024 and exposes library and binary targets at version 0.2.0. All 236 declarations in the original audited inventory remain present or are intentionally superseded by the recovery, migration, dispatch, retention, publication, observability, and bounded-execution APIs described below; new declarations were added without deleting a documented production contract. Generated derives, dependency code, constants, benchmarks, and test-only helpers are not counted as production functions.

The executable boundary is deliberately small. `lib.rs` declares 17 modules and re-exports the runtime/configuration pair and the error/result pair. No model SDK is linked. Axum owns HTTP transport; Rusqlite owns local control and warehouse adapters; Sqlparser supplies a dialect-neutral AST; Tokio supplies the asynchronous server shell and bounded blocking lane; HMAC-SHA-256 signs capabilities; SHA-256 hashes content; UUIDv7 supplies time-ordered identifiers. Filesystem object promotion and operational counters are isolated in `publication` and `observability`.

G.2 Complete Public Type Inventory

Type	Purpose and contract
`api::AppState`	Shared Axum state containing an atomic runtime reference and an identity-provider port; fields are private so routes cannot replace authorization state.
`auth::IdentityProvider`	Thread-safe bearer-authentication port. Production embeds an enterprise implementation; transport code depends only on this trait.
`auth::StaticIdentityProvider`	Immutable token-to-identity map used only by explicit demo mode.
`connectors::EntityRef`	Source-qualified discovery result: source ID, reference, and entity type.
`connectors::Page<T>`	Cursor page containing typed items and an optional next cursor.
`connectors::Validation`	Revalidation result with equality flag, current version, and optional reason.
`connectors::BoundedHandle`	Capability-scoped, version-bound source content and hash.
`connectors::Connector`	Async source port for discovery, observation, bounded reads, revalidation, subscriptions, and health.
`connectors::SqliteWarehouseConnector`	Reproducible local warehouse adapter with fixed tenant/source scope and a durable, deduplicated, 250-item cursor event inbox in the warehouse database.
`context::ContextCompiler`	Service that compiles permitted memory into required-role coverage, optional evidence, versions, omissions, conflicts, and token accounting.
`domain::MemoryType`	Fourteen memory categories: active context, data/stream/schema/semantic/document state, prior work, feedback, policy, execution, artifact, claim, and provenance.
`domain::Authority`	Ordered trust tier from untrusted external through owner approved. Authority resolves conflicts but never grants access.
`domain::MemoryStatus`	Staging, active, superseded, revoked, rejected, pending-review, and tombstoned lifecycle states.
`domain::MemoryObject`	Full governed-memory record: tenant/object/logical identity; type; summary/content/reference; source/version; authority; effective and recorded time; permissions/sensitivity; lifecycle; supersession; provenance; hash; governing flag.

Type	Purpose and contract
`domain::Identity`	Tenant-bound subject with roles, groups, permission labels, policy attributes, and immutable policy epoch for a request.
`domain::ConsistencyClass`	C0 best effort, C1 revalidated observation, and C2 reconstructible snapshot contract.
`domain::RiskClass`	Exploratory, internal, material-internal, external, and regulated task risk.
`domain::Budgets`	Context items/tokens, steps/repairs, and worker row/byte/second limits.
`domain::TaskDefinition`	Versioned task policy: roles, consistency, tools, claim types, verifier/publication profiles, budgets, artifact schema, and effective interval.
`domain::AtxnState`	Twenty explicit analytical-transaction states from admission through evidence, object finalization, publication, review, rejection, abort, and revocation.
`domain::Outcome`	Pass, warning, repair, needs-review, reject, or abort.
`domain::AnalyticalTransaction`	Durable request identity, task/risk/budget/policy snapshot, source version vector, CAS state sequence, terminal flag, outcome, warnings/errors, and timestamps.
`domain::ContextOmission`	Optional role and safe omission reason.
`domain::ContextConflict`	Logical key, conflicting object IDs, and conflict reason.
`domain::ContextManifest`	Immutable compiler output containing role coverage, selected objects, omissions/conflicts, token count, source versions, and warnings.
`domain::OperationLimits`	Per-step seconds, row, and byte bounds.
`domain::PlanStep`	Tool step with purpose, source, input object IDs, parameter/output schemas, parameters, limits, attempts, repair classes, and verifier profile.
`domain::TypedPlan`	Tenant/A-TXN/task/manifest-bound list of steps and planner identity.
`domain::CapabilityLimits`	Signed seconds, row, and byte bounds copied from a plan step.
`domain::CapabilityClaims`	Signed issuer/audience, tenant/A-TXN/plan/step/subject/tool/source scope, operations/relations, limits, policy epoch, fence, token ID, and validity interval.
`domain::CapabilityEnvelope`	Capability claims plus URL-safe HMAC signature.
`domain::ExecutionRecord`	Tool/version/capability, parameters and hash, input versions, output and hash, resource counts, latency, fence, status, and timestamp.
`domain::VerificationCheck`	Stable verifier rule ID, outcome, and optional safe message.
`domain::VerificationRecord`	Versioned profile result with checks, warnings/errors, optional bounded repair, input hash, and creation time.
`domain::SqlPreflight`	Ephemeral context manifest ID, referenced source versions, and proposed-step verification.
`domain::PublicationValidity`	Draft, valid-at-publication, publication-failed, or revoked.
`domain::SemanticValidity`	Current, pending-revalidation, stale, or invalid.
`domain::PolicyVisibility`	Allowed, redacted, or denied.
`domain::ReplayAvailability`	Levels 0–4 plus degraded or expired.
`domain::ReviewState`	Unreviewed, needs-review, verified, approved, corrected, or rejected.
`domain::SupersessionState`	Active, superseded, or tombstoned.
`domain::Claim`	Typed text/payload with risk, execution/verification support, and six independent validity dimensions.
`domain::EdgeEndpoint`	Typed dependency-graph endpoint ID.

Type	Purpose and contract
`domain::DependencyEdge`	Claim support edge with relation, optional source version, creating A-TXN, and integrity hash.
`domain::Artifact`	Tenant/A-TXN-bound rendered object with type/title/content/hash, audience/risk, object state, and publication validity.
`domain::ReplayPackage`	Replay level, manifest/plan/execution IDs, template/config hash, retention deadline, expected hashes, and source versions.
`domain::ReviewDecision`	Approve, reject, or correct.
`domain::Review`	Idempotent reviewer command, scoped claims, decision/comment/correction, authority, effective time, mutation assertion, and request hash.
`domain::ReviewResult`	Review plus resulting transaction, artifact, and claim states.
`domain::AuditEvent`	Append-only actor/action/target, request/A-TXN correlation, outcome, policy epoch, details, and time.
`domain::OutboxEvent`	Tenant-scoped event identity/type/aggregate/idempotency key, payload, and delivery timestamps.
`domain::JobState`	Ready, running, retry-wait, complete, dead-letter, or cancelled.
`domain::Job`	Durable work identity/payload/key, attempt budget, monotonically increasing fence, lease owner/expiry, schedule, and dead-letter reason.
`domain::SourceObservation`	Versioned source observation with effective time, freshness, classification, permissions, consistency, and retention deadline.
`domain::SourceEvent`	Source change with before/after version, kind, occurrence/observation time, cursor, and deduplication key.
`domain::ConnectorHealth`	Source status, lag, rate-limit remainder, and degraded capabilities.
`domain::RunResult`	Complete vertical-slice result: transaction, manifest, plan, executions, verifications, artifact, claims, dependencies, and replay package.
`domain::ReplayResult`	Artifact, outcome, matching/changed execution IDs, warnings, and errors.
`error::Result<T>`	Crate-wide result alias using `AmosError`.
`error::AmosError`	Stable internal taxonomy for not-found, authentication/authorization, conflict/validation/context/state/idempotency/capability, connector/execution, storage, and serialization failures.
`evidence::EvidenceService`	Policy-aware edge construction, atomic human feedback, and invalidation entry service.
`memory::RetrieveQuery`	Request text, required types, effective-time window, and candidate cap.
`memory::RetrievalResult`	Ordered visible objects plus safe warnings.
`memory::MemoryService`	Policy-aware memory write, list, supersession, retrieval, reconciliation, and compaction service.
`policy::PolicyEngine`	Stateless typed authorization rules for current MVP resources and actions.
`runtime::RuntimeConfig`	Control/warehouse paths plus private capability key; custom debug formatting redacts the key.
`runtime::AmosRuntime`	Composition root containing store, policy, memory, context, connector, verifier, workers, evidence, scheduler, and orchestration.
`scheduler::Scheduler`	High-level durable job API over the store.
`store::Store`	Cloneable local persistence adapter sharing a mutex-protected SQLite connection.
`verification::Verifier`	Policy, SQL-AST, schema, metric, freshness, repair, and claim-support verifier.
`workers::CapabilityIssuer`	HMAC capability issuer/validator with a minimum 256-bit secret.
`workers::SqlWorker`	Read-only SQLite query worker bound to capabilities and plan fences.
`workers::StatisticsWorker`	Deterministic built-in statistical transformations.
`workers::ChartWorker`	Deterministic SVG chart generator with content hashing.

Type	Purpose and contract
`domain::ContextRankingEntry`	Per-candidate role, score, consistency, selection, and omission reason frozen in the manifest.
`domain::ReplayComparisonKind`	Exact, semantically equivalent, or different replay classification.
`domain::ReplayExecutionComparison`	Original/replay execution IDs, expected/actual hashes, classification, and explanation for one step.
`domain::OutboxState`	Ready, running, retry-wait, delivered, or dead-letter durable delivery lifecycle.
`domain::RetentionRecord`	Tenant/target deadline, legal-hold flag, reason, actor, and update time.
`domain::RetentionCommand`	Transport-independent retention mutation with explicit idempotency key.
`domain::ErasureReceipt`	Content-hash proof, affected claim IDs, actor, command key, and completion time without erased content.
`workers::ExecutionCancellation`	Cloneable atomic cancellation handle consumed by the SQLite progress callback.
`scheduler::OutboxDestination`	Synchronous idempotent destination port for one leased event.
`scheduler::OutboxDispatcher`	Bounded leased delivery loop with retry/dead-letter completion and shutdown support.
`observability::OperationalMetrics`	Lock-free tenant-safe task/recovery counters and fixed latency buckets.
`observability::MetricsSnapshot`	Serializable administrator-facing point-in-time metric values and bucket bounds.
`publication::PromotedObject`	Stable key, local path, verified hash, and byte count for a finalized object.
`publication::ObjectStore`	Stage, promote, and read-back port with expected-hash enforcement.
`publication::LocalFilesystemObjectStore`	Fsyncing, mode-restricted, path-safe, atomic local object adapter with lost-ack idempotency.

Table 16: Public type, trait, and alias declarations, including completion-pass additions.

Private transport and persistence types remain deliberately narrow:

- `TaskRequest` and `TaskForm` deserialize task inputs;
- `ReviewRequest` and `ReviewWithArtifactRequest` deserialize review inputs;
- `MemorySearchRequest`, `VerifySqlRequest`, and `CursorQuery` cover search, verification, and cursor paging;
- `ReplayRequest`, `ErasureRequest`, `Limit`, `JobRequest`, and `SourceCursor` deserialize the remaining API inputs;
- `RequestIds` carries validated request and correlation identifiers; `Cli` and `Command` define the binary;
- `ErrorEnvelope` and `ErrorBody` serialize safe failures; `InvalidationReceipt` provides idempotent impact receipts; and `SchemaReferences` collects SQL AST relations, identifiers, and aliases.

G.3 Exact Local Persistence Schema

Table	Indexed columns and invariant
memory_objects	Composite tenant/object primary key; immutable unique source/logical/version identity; indexed scope, effective interval, source version, and supersession. Full typed body is JSON.
task_definitions	Tenant/task/version primary key; approved definitions are selected by greatest version.
atxn_transactions	Tenant/A-TXN primary key; tenant/idempotency unique key; indexed state/time; scalar state sequence and terminal flag drive CAS.
context_manifests	Tenant/manifest primary key with A-TXN identity and immutable JSON body.
plans	Tenant/plan primary key with A-TXN identity and immutable JSON body.
executions	Tenant/execution primary key; unique tenant/A-TXN/step/fence effect; output hash and fence are scalar columns.
verification_records	Tenant/verification primary key; A-TXN and outcome columns support inspection.
artifacts	Tenant/artifact primary key; A-TXN, content hash, publication validity, and validity sequence.
claims	Tenant/claim primary key; artifact and all six validity dimensions are indexed/queryable; validity sequence supports change tracking.
dependency_edges	Tenant/edge primary key with source and target reverse indexes.
replay_packages	Tenant/package primary key and unique tenant/artifact mapping; retained-until column.
replay_comparisons	Tenant/replay-A-TXN primary key with artifact/status index and immutable comparison body.
reviews	Tenant/review primary key and unique tenant/idempotency key; request hash rejects conflicting reuse.
invalidation_ receipts	Tenant/deduplication primary key; request hash makes source-event processing idempotent.
audit_events	Tenant/event primary key; tenant/created-time index; append-only JSON evidence.
retention_records	Tenant/type/target primary key with deadline/legal-hold due index.
erasure_receipts	Tenant/receipt primary key and unique tenant/idempotency proof.
outbox_events	Tenant/event primary key and unique event idempotency key; pending index over tenant/completion/creation.
jobs	Tenant/job primary key and unique enqueue idempotency key; ready index over tenant/state/schedule/lease expiry.
schema_migrations	Forward-only version primary key, unique name, checksum, and applied timestamp.
memory_fts	FTS5 virtual table keyed to memory row IDs for bounded lexical candidate generation.

Table 17: Current SQLite control-plane tables.

G.4 Transport and CLI Surface

The router registers 30 paths. /health and /v1/openapi.json are public. Protected HTML paths are /, /memory, /reviews, /operations, and /ui/tasks. Protected API paths are task and transaction lookup; artifact list/cursor page/detail/replay/revalidation/review; claim detail; memory list/write/search/supersession; SQL preflight; audit; metrics; retention/erasure; jobs; connector health; and source-event processing. Replay and review each have a canonical artifact-scoped path plus a compatibility alias. Successful v1 calls retain their original typed-resource shapes for compatibility; request/correlation metadata is carried in headers and failures use the stable error envelope.

Path	Method/access	Handler and result
/health	GET, public	health: liveness/version JSON.
/v1/openapi.json	GET, public	openapi: OpenAPI 3.1 contract JSON.

Path	Method/access	Handler and result
/	GET, authenticated	workspace: Analysis Workspace HTML.
/memory	GET, authenticated	memory_studio: visible memory HTML.
/reviews	GET, reviewer	review_queue: visible review work HTML.
/operations	GET, administrator	operations_console: audit HTML.
/ui/tasks	POST, authenticated	run_task_form: run and render a task.
/v1/tasks	POST, authenticated	run_task: typed run result.
/v1/tasks/{id}	GET, owner/reviewer	get_transaction: lifecycle record.
/v1/transactions/{id}	GET, owner/reviewer	Alias of get_transaction.
/v1/artifacts	GET, owner/reviewer	list_artifacts: visible artifact page.
/v1/artifacts/page	GET, owner/reviewer	list_artifacts_page: opaque cursor page.
/v1/artifacts/{id}	GET, owner/reviewer	get_artifact: artifact/claims/edges.
/v1/artifacts/{id}/replay	POST, owner/reviewer	replay: output-hash comparison.
/v1/replay/{id}	POST, owner/reviewer	Compatibility alias of replay.
/v1/artifacts/{id}/revalidate	POST, reviewer	revalidate: validity changes.
/v1/artifacts/{id}/reviews	POST, reviewer	review: artifact-scoped decision.
/v1/reviews	POST, reviewer	review_with_artifact: body-scoped alias.
/v1/claims/{id}	GET, owner/reviewer	get_claim: claim and support.
/v1/memory	GET/POST, authenticated	list_memory/write_memory.
/v1/memory/search	POST, authenticated	search_memory: governed retrieval.
/v1/memory/{id}/supersede	POST, authorized writer	supersede_memory: append replacement.
/v1/verify/sql	POST, authenticated	verify_sql: typed preflight.
/v1/audit	GET, administrator	audit: bounded tenant audit.
/v1/metrics	GET, administrator	metrics: bounded operational snapshot.
/v1/retention	POST, administrator	set_retention: idempotent deadline/legal hold.
/v1/retention/memory/{id}/erase	POST, administrator	erase_memory: due non-held erasure proof.
/v1/jobs	GET/POST, administrator	list_jobs/enqueue_job.
/v1/connectors/health	GET, administrator	connector_health: source status.
/v1/source-events/process	POST, administrator	process_source_events: impact map.

Table 18: Exact route registration, authorization, and handler map.

The binary accepts --root, requires explicit --demo or AMOS_DEMO=true, and implements the seed, serve, run, and replay commands. Serve accepts port and seed-demo flags; run accepts request and identity; replay requires artifact and idempotency-key values and accepts identity. Without demo mode the binary fails before initializing storage because no production identity provider is embedded.

H Foundational Function Reference

Each row names the owning type where Rust permits the same function name in several implementations. "Reads/writes" describes durable or external side effects; pure constructors and helpers have none. Complexity statements are stated where they affect design; all errors use the taxonomy in `AmosError`.

This appendix covers domain, authentication, error, policy, memory, and context functions. Appendix K continues with runtime, HTTP, persistence, connector, worker, and verifier functions. Together the two appendices preserve the original 236-declaration audit baseline and document the completion APIs that intentionally replace or extend it.

H.1 Domain, Authentication, Errors, Policy, Memory, and Context

Function or method	Detailed behavior
`domain::new_id`	Prefixes a simple UUIDv7 value. Time ordering improves locality and debugging; uniqueness is probabilistic and no storage is touched.
`domain::content_hash`	Serializes a value as JSON and returns a fallible `sha256:` hex digest; serialization errors propagate and can never alias the hash of empty bytes.
`domain::stable_id`	Hashes the fallibly serialized value, removes the digest scheme, and prefixes the first 128 digest bits to create deterministic checkpoint and event identities.
`Authority::rank`	Pure constant mapping from the six authority variants to 0–5, used for reconciliation and indexed persistence.
`MemoryObject::new`	Builds an active, internal, governing version with new ID, current record time, empty permissions/supersession, and a hash of content; callers then add policy/time metadata.
`MemoryObject::effective_at`	Returns interval overlap using inclusive endpoints; missing start/end is unbounded.
`Budgets::default`	Returns 16 items, 8,000 tokens, eight steps, two repairs, 50,000 rows, 5 MB, and 30 seconds.
`AtxnState::can_transition`	Pure exhaustive transition guard matching Specification A; any absent edge is forbidden.
`AtxnState::terminal`	Marks rejected, aborted, and revoked as terminal. Published is intentionally nonterminal so revocation can follow.
`Job::ready`	Constructs an unleased ready job with a new ID, attempt/fence zero, immediate eligibility, caller budget, and no failure reason.
`IdentityProvider::authenticate_bearer`	Trait contract: map one bearer credential to a complete tenant-bound identity or return `Unauthenticated`; implementations must be thread safe.
`StaticIdentityProvider::new`	Wraps an immutable credential map in `Arc` for cheap clone and concurrent reads.
`StaticIdentityProvider::demo`	Installs the fixed local identities returned by `demo_identities`.
`StaticIdentityProvider::authenticate_bearer`	Clones the mapped identity; unknown credentials fail as `Unauthenticated` without disclosing valid tokens.
`auth::demo_identities`	Creates analyst, reviewer, and administrator demo principals for `tenant_demo`; it is never selected unless demo mode is explicit.
`AmosError::from(rusqlite)`	Converts all SQLite failures to a storage error while preserving the driver message for internal/API error text.
`AmosError::from(serde_json)`	Converts JSON encode/decode failures to serialization errors.
`AmosError::into_response`	Maps every error variant to stable HTTP status/code/retry/review flags and a new request ID; authentication failures include `WWW-Authenticate: Bearer`; safe details are empty.

Function or method	Detailed behavior
`PolicyEngine::can_read_memory`	Pure fail-closed predicate requiring tenant equality, every object permission label in the subject permission set, and neither revoked nor tombstoned status.
`PolicyEngine::authorize_memory_write`	Rejects cross-tenant writes and maps authority tier to permitted roles: owner/admin, reviewer/owner/admin, connector/system/admin, unrestricted user note, runtime/system/admin hypothesis, or connector/system/admin untrusted source.
`PolicyEngine::authorize_task`	Requires tenant equality and approved definition; external or regulated tasks additionally require reviewer/owner/admin.
`PolicyEngine::authorize_tool`	Requires the tool in the task allowlist and all declared relation labels in subject permissions.
`PolicyEngine::authorize_review`	Requires owner/admin for owner-authority feedback; otherwise reviewer/owner/admin.
`PolicyEngine::authorize_transaction_read`	Allows the owning subject or a reviewer/owner/admin within the tenant; denies cross-tenant access first.
`PolicyEngine::authorize_artifact_read`	Checks artifact-to-transaction tenant and A-TXN provenance, then delegates transaction ownership/reviewer authorization.
`PolicyEngine::authorize_claim_read`	Authorizes the transaction, tenant, and `PolicyVisibility::Allowed`; redacted and denied claims are not returned.
`PolicyEngine::authorize_revalidation`	Restricts revalidation to reviewer/owner/admin roles.
`PolicyEngine::authorize_operations`	Restricts operations endpoints to the administrator role.
`PolicyEngine::can_review_resources`	Private shared role predicate for reviewer, owner, or administrator.
`policy::has_any`	Pure set-intersection helper used by role checks.
`MemoryService::new`	Composes a store and policy engine without I/O.
`MemoryService::write`	Authorizes authority/tenant, validates nonempty logical key/source and exact content hash, then persists an immutable source version.
`MemoryService::list_visible`	Loads active tenant memory and filters every object with policy before returning it. Current cost is linear in active tenant memory.
`MemoryService::supersede`	Loads the old object, checks visibility and replacement write authority, requires the same logical key, appends the old ID to `supersedes`, and delegates one atomic old/new commit.
`MemoryService::retrieve`	Builds a sanitized FTS query; pushes tenant, active/non-superseded, interval, type, and all-label permission predicates into SQLite; reauthorizes returned objects; and retains bounded top-K results from at most 2,000 candidates.
`MemoryService::reconcile`	Groups by memory type and logical key, orders by authority then record time, reports divergent equal-highest-authority hashes as conflicts, otherwise selects one winner.
`MemoryService::compact`	Rejects empty or partly invisible sources; creates a non-governing active-context summary with union permissions, interval intersection, all source IDs in provenance, and a fresh hash. It does not persist automatically.
`memory::terms`	Splits on non-alphanumeric characters, lowercases, drops terms shorter than three characters, and deduplicates into an ordered set.
`memory::retrieval_score`	Counts term substring matches across key/summary/type/content, weighted by 100, then adds authority and governing bonuses. It receives only already-authorized candidates.
`MemoryTypeString::memory_type_string`	Trait declaration for rendering a memory type into the retrieval search text.

Function or method	Detailed behavior
`MemoryObject::memory_type_string`	Renders the closed memory-type enum deterministically for lexical scoring without a fallible serialization fallback.
`ContextCompiler::new`	Captures a memory service.
`ContextCompiler::compile`	Validates tenant/status/effective task interval; retrieves bounded authorized candidates; enforces per-role consistency minima; rejects governing ambiguity; selects required roles before optional diversity; accounts exact tokens; and freezes omissions, ranking trace, tokenizer, source versions, and selected objects.
`context::role_candidates`	Filters reconciled objects by type and optional payload `role`; orders candidates by authority descending.
`context::exact_token_count`	Serializes canonical content and counts the declared `amos_lexical_v1` tokens exactly; checked addition and required-first selection enforce the frozen token budget.

I Executable Verification Matrix

The following inventory began with 42 named tests and now contains 61 unit, API, end-to-end, security, concurrency, migration, failure-injection, privacy, and recovery tests. "Pass" means `cargo test --all-targets` completed with zero failures; the release invocation runs the same contracts with optimizations.

I.1 Original Library Unit and Storage Contracts (17 of 31)

Test	Control boundary proved
`state_machine_matches_normative_contract`	Representative valid/invalid A-TXN edges, including repair, post-review revalidation, and post-publication revocation.
`static_provider_rejects_unknown_credentials_as_unauthenticated`	Unknown demo bearer tokens fail with the authentication error class.
`permission_filter_is_all_labels`	Object permissions are an all-label subset requirement, not an any-label match.
`analyst_cannot_forge_owner_memory`	Analyst role cannot create owner-approved governing memory.
`connector_has_stable_version_and_detects_change`	Unchanged SQLite schema hashes stably and an added column fails revalidation.
`retrieve_filters_permissions_before_results`	Restricted memory is removed before retrieval results are constructed.
`normalize_ignores_sql_spacing`	Verifier normalization treats equivalent spacing around a required predicate consistently.
`signed_capability_rejects_tampering`	Modified signed claims fail capability validation.
`source_version_identity_is_immutable`	Reusing a source/logical/version key with different content is rejected.
`schema_upgrade_backfills_queryable_claim_validity_dimensions`	Additive initialization upgrades legacy rows and backfills indexed validity fields from JSON.
`invalidation_is_bounded_and_enqueues_a_continuation_page`	Page cap is honored and excess impact creates durable continuation work.
`concurrent_admission_returns_one_resource_and_one_outbox_event`	Racing identical admissions converge on one A-TXN and one admission effect.
`admission_rejects_a_reused_key_with_a_different_request_hash`	Idempotency-key reuse for different semantic input returns a stable conflict.
`concurrent_transitions_use_compare_and_swap_and_emit_one_transition_event`	Only one racing state transition wins the expected state/sequence CAS and emits its effect.
`execution_commit_is_fenced_and_idempotent_per_plan_step`	Stale A-TXN fences fail and identical step/fence effects converge without duplicate completion.
`checkpoints_preserve_fixed_fields_source_versions_and_terminal_immutability`	Checkpoints cannot rewrite admission, observed versions, state sequence, or terminal records.
`expired_final_job_attempt_moves_to_dead_letter_instead_of_redelivery`	An expired lease at the final attempt is dead-lettered and not handed out again.

I.2 Original API and Binary Contracts (9 of 12)

Test	Control boundary proved
versioned_api_exposes_the_complete_local_mvp_contract	Authenticated routes cover task, memory, verification, artifacts, claims, review, replay, validity, audit, jobs, connector, and source-event behavior.
protected_api_and_ui_routes_fail_closed_without_valid_bearer_credentials	Every route except public health and OpenAPI rejects missing, malformed, and unknown credentials.
task_admission_returns_a_stable_idempotency_conflict_for_a_changed_request	HTTP task admission exposes semantic idempotency conflict consistently.
review_mutations_are_idempotent_and_commit_one_feedback_job_and_event	Repeated review HTTP commands return one review and do not duplicate feedback, job, audit, or outbox state.
bundled_binary_requires_explicit_demo_mode_before_initializing_storage	CLI fails before creating local databases when demo mode is absent.
bundled_binary_initializes_demo_storage_only_when_demo_mode_is_explicit	Explicit demo mode permits deterministic local initialization.
transactions_artifacts_claims_and_replay_enforce_owner_and_policy_visibility	Non-owner analysts cannot inspect another analyst's transaction/artifact/replay, and non-visible claims remain denied.
ui_uses_authenticated_identity_and_cannot_be_upgraded_by_form_fields	Browser form input cannot select or elevate the authenticated principal.
memory_listing_applies_policy_permissions_before_serialization	The list endpoint never serializes a memory object missing caller labels.

I.3 Original End-to-End, Security, and Concurrency Contracts (16 of 18)

Test	Control boundary proved
runtime_requires_an_explicit_cryptographically_sized_capability_key	Runtime construction rejects a signing key below 32 bytes.
runtime_configuration_redacts_the_capability_key_from_debug_output	Debug formatting cannot print the signing secret.
complete_vertical_slice_is_review_gated_and_replayable	One request persists manifest, plan, three executions, verifications, claims, edges, replay package, and needs-review outcome; replay hash comparison succeeds.
idempotent_request_returns_the_original_committed_resource	Repeated semantic admission returns the original completed result rather than a duplicate resource.
review_appends_feedback_without_mutating_original_artifact	Review creates governed feedback and claim-state changes while artifact content/hash remain unchanged.
concurrent_review_retries_across_connections_commit_once	Separate SQLite connections racing the same review converge atomically.
review_commit_rolls_back_every_record_when_the_job_conflicts	A job conflict aborts review, feedback, claim, audit, and event writes together.
reviewer_feedback_is_selected_on_the_next_relevant_run	Approved feedback becomes optional governed context in the next applicable task.
authorized_approval_completes_the_local_publication_lifecycle	Full reviewer approval revalidates governing state and atomically publishes transaction, artifact, and claims.

Test	Control boundary proved
`verifier_rejects_unsafe_queries_and_permits_only_declared_repairs`	Writes, schema violations, blocked data, and missing metric filters reject; only known renamed-column repair is allowed.
`memory_permissions_and_document_content_cannot_reprogram_the_plan`	Restricted memory is absent and prompt-injected document text cannot change deterministic tool plan or gain capabilities.
`source_invalidation_traverses_reverse_claim_dependencies`	A source change reaches supporting memory, marks dependent claims pending, and creates revalidation work.
`scheduler_rejects_a_stale_fencing_token`	Scheduler completion with an old fence cannot commit.
`job_enqueue_is_idempotent_only_for_the_same_job_request`	Same key/same job semantics converge; changed type, payload, or attempt budget conflicts.
`expired_running_job_is_redelivered_with_a_higher_fence`	Expired leased work is reacquired with increased attempt and fence, invalidating the old handle.
`expired_or_wrong_owner_job_leases_cannot_commit_and_active_leases_can_renew`	Wrong owner and expired lease fail; an active matching owner/fence can only extend the lease and then finish.

I.4 Completion Contracts Added After the Baseline (19)

The completion pass adds explicit tests for fallible hashing; deterministic context consistency, token budgets, ranking traces, and ambiguity; storage-filtered FTS top-K retrieval; connector signature/relation binding and durable cursor restart; exhaustive capability-field mismatch; worker timeout, cancellation, and incremental bytes; outbox leases, delivery retry, dead letter, bounded shutdown, and stale handles; transitive continuation consumption; checksummed future/tampered migration rejection; state-driven recovery with a new runtime at fourteen lifecycle checkpoints; replay persistence/original immutability; object-promotion lost acknowledgment and unsafe keys; legal hold and idempotent erasure; API request limits and browser headers; opaque artifact cursors; and erasure-driven claim visibility revocation. The benchmark is a separate release executable, not counted as a unit test.

J Production Gap Register and Completion Checklist

This register prevents a reader from confusing complete documentation of the MVP with a claim that production infrastructure already exists.

Area	Implemented local contract and evidence	External infrastructure still required
Tenancy and identity	Composite tenant keys, authenticated-tenant source events, owner/reviewer reads, all-label permissions, stale policy epochs, and hidden claims fail closed in executable tests.	Enterprise OIDC/SAML/JWKS and PostgreSQL forced RLS require a deployed identity tenant and database.
Storage concurrency	Tokio-facing storage and review work use an eight-permit blocking lane; independent connections exercise CAS/fences. Schema v6 has a checksummed ledger, legacy backfill, and future/tamper rejection.	Production pool sizing, online PostgreSQL DDL, rollback rehearsal, backup, and point-in-time restore.
Retrieval scale	FTS5 pushes tenant, active, supersession, interval, type, and all-label predicates into storage; a bounded candidate cap and heap produce top-K.	Million-object noisy-neighbor qualification against named production search hardware.
Context semantics	Consistency minima, effective definitions, deterministic ranking trace, required-first selection, exact tokenizer accounting, ambiguity, and budget-pressure tests are implemented.	Organization-specific ranking evaluation corpora.
Connector durability	Events are durable, deduplicated, restart-safe, and cursor paged; unknown cursors fail closed and source reads revalidate signed capabilities.	Customer credentials, auth rotation, vendor quota adapters, and outage certification.
SQL enforcement	The frozen one-query subset rejects joins, CTEs, sets, subqueries, unsupported functions, and unbounded time; progress cancellation and incremental time/row/byte caps are tested.	Driver-specific cancellation conformance for each customer database.
Capability hardening	HMAC verification is constant-time and every tenant, subject, A-TXN, plan, step, tool, source, operation, relation, limit, epoch, fence, issuer, audience, and time field is rebound.	KMS/HSM custody, rotation, isolated worker identity, and egress policy.
Worker isolation	Bounded execution, cancellation, query-only mode, untrusted-schema disablement, and capability fencing are local controls.	Per-tool containers or VMs, CPU/RSS/filesystem/network policy, and credential broker.
Runtime recovery	`recover_task` is state driven over durable manifest, plan, execution, evidence, review, object, and publication checkpoints. A test rebuilds the runtime at fourteen lifecycle boundaries.	Distributed controller election and remote worker placement.
Verification breadth	Referential execution/verifier/memory support, numeric rates, concentration payload, statistical constraints, and deterministic chart SVG/hash binding are recomputed under profile version 2.	Additional task-specific calibrated verifier packs.
Review and policy	Epochs are enforced at execution, commit, recovery, and read; reviews and validity changes are atomic and permission revocation redacts dependent claims.	External policy evaluator, simulation, and organization activation workflow.
Invalidation	General reverse traversal uses a visited set, quotas, durable after-cursors, idempotent continuation consumption, stable audits, and claim-revalidation jobs.	Production storm thresholds and cooldown policy tuned on release infrastructure.

Area	Implemented local contract and evidence	External infrastructure still required
Replay	A separate replay A-TXN persists fenced executions, exact/equivalent/different comparisons, audit, and outbox state while preserving the original.	Long-term source-version leases supplied by customer data platforms.
Publication	Filesystem objects stage, fsync, hash-check, atomically promote, read back, and recover a lost acknowledgment under a deterministic key.	S3/GCS lifecycle/residency plus external destination acknowledgment and revocation adapters.
Outbox and jobs	Leased dispatch/worker batches enforce owner/fence/expiry, exponential retry, dead letter, idempotency, bounded shutdown, and completion timestamps.	Broker adapter and production alert routing.
API contract	Required command keys, stable errors, a 1 MiB cap, opaque artifact cursors, request/correlation IDs, security headers, compatibility tests, and an OpenAPI 3.1 contract endpoint are present.	Generated client SDK distribution and tenant-specific quota configuration.
Browser security	Bearer-only forms have no ambient cookie authority; CSP, frame denial, referrer denial, no-sniff, no-store, strict auth, and escaped output are tested.	Secure-cookie and CSRF design only if an enterprise session adapter replaces bearer-only requests.
Observability	Tenant-safe counters and latency buckets are administrator-only; structured request traces, durable audit, outbox, and failure states are present.	Metrics/trace backend, SLO dashboards, alerts, paging, and game days.
Retention/privacy	Versioned retention, legal holds, due erasure, FTS deletion, dependent-claim invalidation/redaction, idempotent receipts, audit, and outbox proof commit atomically.	Regional placement, cloud key destruction, legal export formats, and external deletion verification.
Performance evidence	`control_paths` measures indexed retrieval, SQL preflight, durable commit, job lease/complete, invalidation, complete task, and persisted replay; each reports p50/p95/p99 and enforces debug/release p95 thresholds.	Final throughput, RSS, allocation, and noisy-neighbor envelope on named release hardware.

Table 23: Local completion evidence and the remaining infrastructure-bound release integrations.

J.1 Build and Release Procedure

A reproducible local release runs, in order:

1. `cargo fmt --all -- --check`;
2. `cargo clippy --all-targets --all-features -- -D warnings`;
3. `cargo test --all-targets`;
4. `cargo build --release`;
5. seed a clean root and exercise authenticated task, review, replay, invalidation, and job flows;
6. export the database/audit/outbox state and verify every final numeric claim resolves to its execution, metric, schema, data state, and verifier record;
7. render this PDF and inspect every page for clipping, broken tables, stale counts, and incorrect implementation claims.

Production release adds migration rehearsal, dependency and license audit, SBOM/signing, secret scan, connector conformance, SQL adversarial suite, browser/API security suite, load and noisy-neighbor tests, backup restore, lost-acknowledgment recovery,

invalidation storm, and disaster-recovery exercises. A completed checklist is evidence attached to a release; prose alone is not a gate.

K Service, Persistence, and Transport Function Reference

K.1 Runtime Orchestration

Function or method	Detailed behavior
`RuntimeConfig::fmt`	Custom debug formatter shows control, warehouse, and object paths but replaces the capability key with `[REDACTED]`.
`RuntimeConfig::new`	Constructor accepting control/warehouse paths and key bytes and deriving a sibling object root; key length is checked by the issuer.
`RuntimeConfig::demo`	Derives local control, warehouse, and object paths under a root and installs an explicit, known demo-only signing key.
`AmosRuntime::open`	Opens schema-v6 store, policy/memory/context/evidence/scheduler/verifier, capability issuer, SQLite connector/worker, bounded semaphore, metrics, and filesystem object store.
`AmosRuntime::get_transaction_for`	Tenant-loads a transaction, fails if absent, then applies owner-or-reviewer authorization.
`AmosRuntime::list_artifacts_for`	Loads a bounded artifact page, resolves each parent transaction and all claims, and returns only artifacts whose artifact and every claim are visible. Filtering after the store limit can yield fewer results than requested.
`AmosRuntime::get_artifact_for`	Loads artifact and parent transaction, authorizes both, authorizes every claim, gathers all outgoing claim dependencies, and returns the complete evidence view.
`AmosRuntime::get_claim_for`	Loads claim, artifact, and transaction; authorizes artifact and claim; returns the claim with outgoing dependencies.
`AmosRuntime::authorize_operations`	Transport-facing delegation to administrator policy.
`AmosRuntime::authorize_review_queue`	Transport-facing delegation to reviewer policy without owner-authority requirement.
`AmosRuntime::run_task`	Enters the bounded blocking lane and executes the legacy payment-health reference A-TXN: idempotent admission; observe; compile; verify/repair; capability/fence execution; independent claim verification; source/policy revalidation; atomic evidence; review or hash-checked publication. Duplicate in-progress requests enter state-driven recovery. This function must become configuration-driven for the target product.
`AmosRuntime::recover_task`	Reconstructs a nonterminal run from tenant-bound durable checkpoints and resumes automatic work. Only the admitting subject at the frozen policy epoch may resume it.
`AmosRuntime::recover_task_until_checkpoint`	Operational failure-injection hook that runs the same controller until a named durable state; recovery tests reopen the runtime at each boundary.
`AmosRuntime::replay`	Authorizes retained evidence, admits an idempotent replay A-TXN, clones the frozen plan, issues new capabilities, persists fence-checked executions, records exact/equivalent/different comparisons, and atomically commits comparison, audit, and outbox state without mutating the original.
`AmosRuntime::preflight_sql`	Loads/authorizes the payment task, compiles an ephemeral context, rejects context conflicts, wraps proposed SQL in a typed step, verifies it, and returns manifest ID plus referenced versions without execution/persistence.
`AmosRuntime::revalidate_artifact`	Requires reviewer authority and visible artifact; for each claim checks referenced memory active/supersession state and replay retention; computes semantic/replay changes; atomically commits legal dimension updates, audit, and outbox events; returns changes and claims.

Function or method	Detailed behavior
`AmosRuntime::review_artifact`	Delegates atomic review/feedback, reloads resources, rejects a needs-review transaction on reject, or on full approval revalidates source/policy, advances lifecycle, and atomically publishes local artifact/claims. Corrections remain append-only feedback and do not mutate evidence.
`AmosRuntime::artifact_transaction`	Private tenant-bound parent transaction lookup for an artifact; policy is applied by the caller.
`AmosRuntime::connector_health`	Async delegation to the configured connector.
`AmosRuntime::process_source_events`	Reads a bounded durable event page, requires every event tenant to equal the authenticated tenant, scans only that tenant's active memory, invokes deduplicated invalidation, and returns event-to-claim impacts.
`AmosRuntime::advance`	Delegates one expected-state/sequence transition and optional outcome to the store.
`AmosRuntime::build_plan`	Deterministically builds summary, concentration, and hourly-series SQL over the fixed evaluation interval and current context; no retrieved document text can alter the plan.
`AmosRuntime::compose`	Requires all three executions; derives rate comparison, top concentration, and chart; renders the report; constructs four typed claims and mandatory metric/schema/data/execution edges plus optional document support.
`AmosRuntime::replay_package`	Creates a level-3 one-year package containing manifest/plan/execution identities, template/config hash, expected artifact/execution hashes, and source versions.
`AmosRuntime::load_result`	Reconstructs a previously committed idempotent run from transaction, artifact, replay package, manifest, plan, claims, edges, executions, and verifications; in-progress duplicates return conflict.
`runtime::step`	Builds one read-only SQL step with all selected object IDs, relation labels, fixed limits, two attempts, renamed-column repair class, and verifier profile.
`runtime::claim`	Builds a draft/current/allowed/level-3/active claim; causal and operational types need review while other types start verified.
`runtime::edge`	Builds and hashes one claim dependency edge for the current tenant/A-TXN.
`runtime::role_id`	Returns the first object covering a required role or fails as required-role missing.
`runtime::find_execution`	Finds an execution by step ID or fails not-found.
`runtime::parse_time`	Parses fixed RFC3339 scenario constants and returns a validation error; no malformed value can panic the runtime.

K.2 HTTP, UI, Seed, and Binary

Function or method	Detailed behavior
`api::router`	Builds shared state, registers 28 protected paths plus public health/OpenAPI, applies authentication to the protected subtree, a 1 MiB body cap, validated request/correlation IDs, and fail-closed browser security headers, then binds state.
`api::demo_router`	Calls the general router with the explicit static demo identity provider.
`api::authenticate`	Extracts one bearer token, calls the provider, injects the returned identity into request extensions, and runs the next handler; any failure becomes the stable error envelope.

Function or method	Detailed behavior
`api::bearer_token`	Strictly accepts exactly two whitespace fields with case-insensitive Bearer scheme and a nonempty token; missing, invalid-text, malformed, or extra fields fail unauthenticated.
`api::run_task`	Requires task text and an explicit idempotency key, logs validated request/correlation IDs with tenant/subject, awaits runtime orchestration, and returns the typed run result.
`api::run_task_form`	Uses the authenticated identity and a fresh UI key, runs the task, escapes all dynamic artifact/claim text, and renders result/evidence HTML. No form field can choose identity.
`api::get_transaction`	Returns the authorized transaction for either task or transaction alias path.
`api::list_artifacts`	Returns up to 50 artifacts visible under runtime artifact and claim policy.
`api::list_artifacts_page`	Decodes a resource-typed opaque cursor, applies a bounded indexed page, authorization-filters the results, and encodes the next artifact cursor.
`api::get_artifact`	Returns artifact, claims, and dependencies as one JSON object after authorization.
`api::replay`	Reads the optional JSON extractor only so authorization can precede malformed/missing-key disclosure; runtime authorization then requires a nonempty explicit key and returns an idempotent computational replay for either alias.
`api::revalidate`	Requires runtime reviewer authorization and returns atomic claim-validity changes.
`api::get_claim`	Returns one authorized claim and its dependencies.
`api::review`	Deserializes an artifact-scoped idempotent review and awaits review/publication orchestration.
`api::review_with_artifact`	Same review operation with artifact ID in the JSON body for the compatibility route.
`api::list_memory`	Returns only policy-visible active memory.
`api::search_memory`	Applies defaults of now, a one-year interval, and 20 items; caps caller items at 100; delegates permission-first retrieval.
`api::verify_sql`	Returns typed SQL preflight for request text and proposed SQL.
`api::write_memory`	Delegates policy/hash/source validation and returns HTTP 201 with the inserted object.
`api::supersede_memory`	Delegates authorized, same-logical-key atomic supersession and returns the replacement.
`api::audit`	Requires administrator role and returns a bounded tenant audit page with default 100.
`api::metrics`	Requires administrator role and returns the tenant-safe task/recovery counter and latency-bucket snapshot.
`api::set_retention`	Requires administrator role and commits an explicit-key deadline/legal-hold command through the bounded blocking lane.
`api::erase_memory`	Requires administrator role and an explicit key, then atomically erases a due non-held object and returns its content-free proof.
`api::connector_health`	Requires administrator role and returns connector health JSON.
`api::enqueue_job`	Requires administrator role, delegates idempotent enqueue, and returns HTTP 201.
`api::list_jobs`	Requires administrator role and returns a bounded tenant job page with default 100.
`api::process_source_events`	Requires administrator role, passes an optional cursor, and returns event-to-affected-claim mapping.
`api::health`	Public liveness response containing status, Rust runtime marker, and package version; it does not probe storage or connectors.

Function or method	Detailed behavior
`api::openapi`	Public OpenAPI 3.1 JSON describing bearer authentication, stable errors, explicit idempotent commands, and principal v1 operations.
`api::workspace`	Renders the authenticated Analysis Workspace task form and product explanation.
`api::memory_studio`	Loads visible memory, escapes text, and renders type/authority/version cards.
`api::review_queue`	Requires reviewer authority, lists visible artifacts, and links to evidence inspection.
`api::operations_console`	Requires administrator authority, lists 50 recent audit events, and renders escaped action/actor data.
`api::escape`	Escapes ampersand, angle brackets, double/single quotes for HTML text/attribute contexts used by the server renderer.
`api::page`	Wraps a trusted server-generated body in a responsive HTML shell; escapes the title and embeds a compile-time stylesheet.
`seed::seed_demo`	Recreates the warehouse; constructs governed metric, schema, snapshot, user/review policy, deployment document, and restricted prior-analysis memory; writes version 3 payment task definition. Repeated control-store seeding is idempotent only where source-version hashes match.
`seed::memory`	Builds a demo memory object and fills effective interval, permissions, version, and recalculated content hash.
`seed::seed_warehouse`	Recreates `payment_events` and deterministically inserts 14,400 minute-distributed events with a post-14:00 failure spike concentrated in Processor B/Visa.
`seed::lcg`	Wrapping 64-bit linear-congruential generator for deterministic fixture data; it is not cryptographic.
`seed::parse`	Parses fixed repository timestamps into UTC and returns a validation error for an invalid fixture.
`main`	Initializes tracing, parses CLI, fails before storage unless demo is explicit, then seeds, serves with Ctrl-C shutdown, runs a task, or replays an artifact using the selected demo identity.

K.3 Evidence, Scheduling, and Persistence

Function or method	Detailed behavior
`EvidenceService::new`	Composes store and policy engine.
`EvidenceService::cite`	Builds a claim-to-target dependency edge, attaches optional source version and creating A-TXN, then hashes the completed edge. It returns the edge without persisting it.
`EvidenceService::review`	Authorizes review authority; validates key and unique claims; authorizes artifact and all claims; hashes normalized request content; creates review, scoped claim-state updates, append-only feedback memory, audit, and revalidation job; then delegates one idempotent atomic commit.
`EvidenceService::invalidate_memory`	Derives a deterministic manual invalidation key from object and reason, then delegates bounded invalidation.
`EvidenceService::invalidate_memory_with_key`	Traverses reverse claim dependencies through the store with a fixed 100-claim page and caller deduplication key.
`Scheduler::new`	Captures a store.
`Scheduler::enqueue`	Creates a ready job with five attempts and delegates idempotent persistence.
`Scheduler::acquire`	Requires positive lease duration, computes now/expiry, and asks the store to acquire one due or expired job.

Function or method	Detailed behavior
`Scheduler::renew`	Requires positive extension and an active matching local handle; extends expiry and performs owner/fence/time-checked persistence.
`Scheduler::complete`	Requires an active matching handle, changes it to complete, clears lease fields, and commits under the current fence.
`Scheduler::fail`	Requires a nonempty reason and active handle; dead-letters when attempts are exhausted, otherwise schedules exponential retry $2^{\min(attempt,8)}$ seconds; clears the lease and fence-checks commit.
`scheduler::active_lease_owner`	Returns the nonempty owner only when state is running and fence matches; stale or ownerless handles conflict.
`Store::open`	Creates parent directories, opens a file-backed SQLite connection, wraps it in a shared mutex, initializes/upgrades schema, and returns the adapter.
`Store::in_memory`	Opens an in-memory connection and runs the identical schema initializer; used by tests and embedding scenarios.
`Store::connection`	Locks the shared connection; a poisoned mutex becomes a storage error. This serializes operations within one store instance.
`Store::init_schema`	Enables WAL, foreign keys, normal synchronization, and a five-second busy timeout; applies and verifies the checksummed forward-only schema-v6 ledger, creates indexed control tables, and backfills legacy claim, review, outbox, and FTS fields idempotently.
`Store::schema_version`	Returns the maximum verified migration-ledger version and fails if the ledger is empty.
`Store::write_memory`	Enforces immutable `(tenant,source,logical,source-version)` identity: identical hash is idempotent, different hash conflicts; otherwise inserts scalar indexes and full JSON.
`Store::get_memory`	Tenant/object lookup through the generic JSON loader.
`Store::list_active_memory`	Returns tenant rows whose scalar status is active and `superseded_by` is null. Authorization is intentionally performed by the memory service.
`Store::supersede_memory`	In one immediate transaction loads and requires an active old version, updates its JSON/scalar lifecycle, inserts the replacement, and commits.
`Store::put_task_definition`	Inserts a definition once by tenant/task/version; later writes with the same key are ignored to preserve immutability.
`Store::get_task_definition`	Selects the greatest approved version for tenant and task type.
`Store::get_task_definition_version`	Loads the exact immutable task version frozen by a recovering A-TXN.
`Store::create_transaction`	Validates admitted sequence-zero state; starts an immediate transaction; resolves tenant/idempotency duplicates by request hash; inserts the A-TXN and same-commit `atxn.admitted` outbox event.
`Store::get_transaction`	Tenant/A-TXN JSON lookup.
`Store::transition_transaction`	Validates the state edge, loads current snapshot, requires expected state/sequence and nonterminal status, increments sequence, updates with a scalar CAS predicate, emits same-commit transition event, and returns the new snapshot.
`Store::checkpoint_transaction`	Under an immediate transaction, verifies fixed admission/lifecycle fields, monotonic update time, and append-only observed versions; performs same-state/sequence CAS without emitting a transition event.
`Store::save_manifest`	Inserts one immutable context manifest and its A-TXN identity; identical duplicate bodies are idempotent and divergent reuse conflicts.
`Store::get_manifest`	Tenant/manifest JSON lookup.
`Store::get_manifest_by_atxn`	Loads the latest manifest checkpoint for one tenant/A-TXN recovery identity.
`Store::save_plan`	Inserts one immutable plan and its A-TXN identity.

Function or method	Detailed behavior
`Store::get_plan`	Tenant/plan JSON lookup.
`Store::get_plan_by_atxn`	Loads the latest immutable plan checkpoint for one tenant/A-TXN.
`Store::save_execution`	Validates essential IDs/hash, requires the parent A-TXN in executing state with matching sequence fence, makes tenant/A-TXN/step/fence effects idempotent, rejects divergent duplicates, inserts execution and same-commit completion event.
`Store::list_executions`	Returns execution records for one tenant/A-TXN in insertion order.
`Store::save_verification`	Inserts an immutable verifier record with scalar outcome; identical duplicate bodies are idempotent and divergent reuse conflicts.
`Store::list_verifications`	Returns verifier records for one tenant/A-TXN in insertion order.
`Store::commit_evidence`	Requires revalidating state and a consistent bundle; snapshot-checks the A-TXN; CAS-transitions to evidence committed; inserts artifact, claims, edges, replay package, audit, and outbox event in one immediate transaction.
`Store::get_artifact`	Tenant/artifact lookup.
`Store::get_artifact_by_atxn`	Returns the most recently inserted artifact for a tenant/A-TXN.
`Store::list_artifacts`	Returns at most 100 tenant artifacts in reverse insertion order; runtime authorization filters the result afterward.
`Store::list_artifacts_after`	Applies an indexed tenant/row cursor and bounded limit; invalid or cross-tenant cursors fail closed before runtime authorization.
`Store::commit_local_publication`	Requires publication-pending state and a consistent draft bundle; snapshot-checks transaction/artifact/claims; CAS-publishes the A-TXN and changes artifact/claim publication validity with sequence increments; inserts audit/outbox atomically; updates caller-owned values only after commit.
`Store::list_claims`	Returns artifact claims in insertion order.
`Store::get_claim`	Tenant/claim lookup.
`Store::list_edges_from`	Uses the source-edge index to return outgoing dependencies for one typed endpoint.
`Store::list_edges_to`	Uses the target-edge index to return reverse dependencies for one typed endpoint.
`Store::invalidate_claims_page`	Validates request and caps pages at 250; resolves duplicate keys by request hash; indexed-joins target edges to claims; changes current claims to pending revalidation; enqueues per-claim work and validity events; optionally enqueues continuation; writes receipt, audit, and processed event atomically.
`Store::get_replay_package`	Retrieves the unique replay package for a tenant/artifact.
`Store::get_replay_result`	Retrieves an idempotently committed comparison for a replay A-TXN.
`Store::commit_replay_result`	Verifies durable replay executions and atomically commits comparison, audit, outbox, and the revalidating-to-evidence CAS.
`Store::commit_review`	Validates the whole review bundle; resolves idempotent duplicates by request hash; snapshot-checks artifact/claims; inserts review; applies only allowed review-state changes; inserts feedback, audit, revalidation job, per-claim events, and review event atomically.
`Store::get_review_by_idempotency_key`	Retrieves a tenant review by its command key.
`Store::commit_claim_validity_updates`	Validates equal immutable evidence sets and legal dimension transitions; snapshot-checks all claims; applies changed dimensions, emits per-claim events, appends audit, and commits atomically.
`Store::append_audit`	Inserts one append-only audit event.
`Store::list_audit`	Returns at most 250 tenant audit events newest first.
`Store::set_retention`	Idempotently upserts one tenant/target policy and commits its stable audit and outbox event; same-key semantic mismatch conflicts.
`Store::get_retention`	Tenant/type/target lookup for the active legal-hold and deadline record.

Function or method	Detailed behavior
`Store::erase_memory`	Requires a due, non-held memory target; traverses dependent claims under quota; atomically redacts/invalidates claims, deletes FTS and memory rows, and appends an idempotent receipt, audit, and outbox proof.
`Store::list_outbox`	Returns at most 500 tenant events, decodes JSON/timestamps, and orders by creation and event ID.
`Store::acquire_outbox`	Leases one due/expired event with increased attempt and fence; exhausted events dead-letter rather than redeliver.
`Store::finish_outbox`	Requires the unexpired matching owner and fence, then atomically delivers, schedules exponential retry, or dead-letters and records an audit.
`Store::enqueue_job`	Validates a new job and wraps the transactional idempotent enqueue helper in an immediate commit.
`Store::acquire_job`	Validates tenant/worker/time; repeatedly selects one due ready/retry job or expired running lease; dead-letters exhausted expired work; otherwise increments attempt and fence, installs owner/expiry with CAS, emits acquisition event, and returns one job.
`Store::renew_job_lease`	Requires a future extension and matching candidate handle; loads current job, validates fixed fields and active lease, permits only a strictly later expiry, CAS-updates, and emits renewal event.
`Store::finish_job`	Accepts only complete/retry-wait/dead-letter with consistent metadata and cleared lease; validates the current unexpired owner/fence; CAS-updates and emits the state-specific event.
`Store::list_jobs`	Returns at most 250 tenant jobs ordered by next-run time descending.
`Store::get_json`	Private generic optional single-row JSON decoder for a caller-supplied SQL statement and bound parameters.
`Store::list_json`	Private generic multirow JSON decoder.
`store::validate_initial_transaction`	Requires all admission identities and sequence-zero admitted, nonterminal, outcome-free state.
`store::validate_transaction_checkpoint`	Forbids terminal mutation, changes to fixed admission/lifecycle fields, backward time, and replacement/removal of already observed source versions.
`store::ensure_transaction_snapshot`	Reloads a tenant/A-TXN inside the caller transaction and requires structural equality before an atomic multi-resource commit.
`store::ensure_artifact_snapshot`	Reloads and structurally compares an artifact before publication/review.
`store::ensure_claim_snapshot`	Reloads and structurally compares a claim before validity/review mutation.
`store::validate_review_bundle`	Verifies identities, recomputed request hash, unique/existing claim targets, correction semantics, immutable claim evidence/dimensions, exact allowed review-state changes, and matching feedback/audit/job metadata.
`store::calculated_review_request_hash`	Hashes the normalized semantic content of a review command, excluding generated IDs and timestamps.
`store::validate_claim_validity_batch`	Requires equal nonempty claim sets and consistent audit identity; freezes evidence, publication and review dimensions; permits only legal semantic/replay/supersession transitions.
`store::valid_semantic_transition`	Allows current to pending/stale/invalid, pending to current/stale/invalid, stale to pending/invalid, identity transitions, and no transition out of invalid.
`store::valid_replay_transition`	Allows identity or one-way degradation/expiry; nothing leaves expired.
`store::valid_supersession_transition`	Allows active to superseded/tombstoned, superseded to tombstoned, or identity; tombstone is final.
`store::same_claim_evidence`	Compares all immutable claim identity/content/risk/support fields while deliberately excluding validity dimensions.

Function or method	Detailed behavior
store:: update_claim_validity_tx	Updates all six validity columns and JSON body and increments validity sequence for one tenant/claim; caller supplies surrounding transaction and snapshot checks.
store::claim_validity_json	Projects the six independent dimensions for outbox before/after payloads.
store:: validate_evidence_bundle	Requires consistent tenant/A-TXN/artifact/package/audit identities, expected artifact hash, unique claims, and claim-rooted same-transaction dependency edges.
store:: validate_publication_bundle	Requires consistent transaction/artifact/claim/audit identities and a draft artifact.
store::validate_initial_job	Requires nonempty identities, ready/unleased state, zero attempt/fence, positive attempt budget, and no failure reason.
store:: same_execution_effect	Compares semantic execution identity, inputs, output, resource counts, fence, and status while ignoring generated execution/capability IDs, latency, and creation time.
store::same_job_request	Defines enqueue idempotency by tenant, command key, type, payload, and attempt budget.
store::get_job_tx	Loads one tenant/job using a caller-owned SQLite transaction/connection.
store:: validate_active_job_lease	Freezes job identity/payload/attempt/fence and requires running state, expected owner/fence, and unexpired current lease.
store::new_outbox_event	Constructs a pending event with a new ID and current time.
store::insert_outbox_tx	Inserts an event's scalar envelope and serialized payload in the caller's transaction; unique idempotency keys reject duplicate effects.
store::enqueue_job_tx	Validates the job; resolves duplicate enqueue keys by semantic request equality; inserts job and same-transaction enqueue event.
store::parse_timestamp	Strictly parses stored RFC3339 text into UTC or returns a storage-corruption error.
store:: add_column_if_missing	Validates table/column identifiers as alphanumeric/underscore, inspects PRAGMA table_info, and applies one additive column migration when absent.
store::insert_memory_tx	Inserts a complete memory row in a caller-owned transaction; used when memory must commit with review state.
store::insert_audit_tx	Inserts one audit row in a caller-owned transaction.
store::to_json	Strict typed JSON serialization into the crate result type.
store::from_json	Strict typed JSON deserialization into the crate result type.
store::enum_json	Serializes a unit enum and strips JSON quotes for scalar database columns.

K.4 Connector, Workers, and Verification

Function or method	Detailed behavior
Connector::source_id	Trait contract returning a stable connector identifier.
Connector::discover	Trait contract for cursor-paged, scope-filtered source-qualified entities.
Connector::observe	Trait contract for a versioned observation including time, freshness, access, consistency, and retention.
Connector::read	Trait contract for capability-bound exact-version content or a bounded handle.
Connector::validate	Trait contract comparing an observed version with current source state.
Connector::subscribe	Trait contract for cursor-paged source changes.
Connector::health	Trait contract for safe health, lag, rate-limit, and degraded-capability status.

Function or method	Detailed behavior
`SqliteWarehouseConnector::new`	Stores tenant/source/path, installs `analytics` and `payments` permission labels, and uses a durable deduplicated connector-event table created on first event access.
`SqliteWarehouseConnector::path`	Returns the configured warehouse path without I/O.
`SqliteWarehouseConnector::emit_change`	Builds a source event and stable deduplication key, inserts it into the durable connector inbox, and returns the original row for an idempotent duplicate.
`SqliteWarehouseConnector::schema_snapshot`	Opens SQLite, normalizes a `table:` reference, safely quotes the table name for `PRAGMA table_info`, returns column name/type/nullability JSON, and fails when the table is absent.
`SqliteWarehouseConnector::source_id`	Concrete trait implementation returning the stored source identifier.
`SqliteWarehouseConnector::discover`	On the first page, lists non-system SQLite tables ordered by name and filters by wildcard or substring; any supplied cursor yields an empty terminal page.
`SqliteWarehouseConnector::observe`	Hashes the current schema snapshot into a C1 source version and attaches tenant, permissions, classification, freshness zero, and observation time.
`SqliteWarehouseConnector::read`	Independently verifies signature/time/audience/epoch/fence plus tenant, source, tool, operation, relation, limits, and invocation identities; then re-hashes content, rejects a version race, and returns a bounded handle.
`SqliteWarehouseConnector::validate`	Recomputes schema version and returns equality plus current version.
`SqliteWarehouseConnector::subscribe`	Loads at most 250 durable events after a tenant-scoped cursor; restart preserves progress and unknown/expired cursors fail closed.
`SqliteWarehouseConnector::health`	Requires a file and successful SQLite open; otherwise fails as connector unavailable, else returns healthy with zero lag.
`CapabilityIssuer::new`	Rejects secrets shorter than 32 bytes, copies the key, and fixes issuer to `amos-runtime`.
`CapabilityIssuer::issue`	Copies plan/step/identity scope and limits, derives worker audience and declared relations, sets query operation, current policy epoch/fence, unique token ID, one-second clock skew, 60-second expiry, then signs the claims.
`CapabilityIssuer::validate`	Decodes the signature and uses constant-time HMAC verification, then checks issuer, audience, activation/expiry, token identity, policy epoch, and fence.
`CapabilityIssuer::sign`	JSON-serializes claims, computes HMAC-SHA-256, and returns URL-safe base64 without padding.
`SqlWorker::new`	Stores the warehouse path and capability issuer.
`SqlWorker::execute`	Validates every capability binding; opens SQLite query-only with untrusted schema disabled; installs a progress handler for cancellation and wall time; accounts rows and serialized bytes incrementally; and returns a hashed fenced execution record.
`StatisticsWorker::rate_comparison`	Rejects zero denominators and returns current/baseline rates plus absolute change as deterministic floating-point JSON.
`ChartWorker::timeseries_svg`	Rejects empty points, scales an accessible 600-by-260 SVG polyline to the maximum value with a 0.01 floor, and returns SVG plus content hash.
`workers::audience`	Maps `sql.*` to SQL worker, `stats.*` to statistics worker, and all other tools to chart worker.
`workers::sql_value`	Converts SQLite null/integer/real/text/blob values to JSON; blobs are URL-safe base64 and invalid UTF-8 text fails execution instead of being silently changed.

Function or method	Detailed behavior
`Verifier::verify_step`	Authorizes tool/relations; parses exactly one read-only query; visits table, column, and alias references; checks schema allowlists, blocked columns, renamed-column repair, required metric filters, and freshness warnings; then returns pass/warning/repair/reject with stable checks and step hash.
`Verifier::repair_step`	Accepts only `COLUMN_SUPERSEDED:old:new`; clones the step and performs the declared string replacement in its SQL. Unknown/malformed repairs return `None`.
`Verifier::verify_claims`	Resolves tenant-scoped execution, verification, and manifest-memory references; requires typed support relations; recomputes rates and concentration payloads; regenerates the deterministic chart and hash; marks causal/operational obligations for review; and emits a version-2 aggregate record.
`verification::record`	Appends a pass or reject check and mirrors failure text into the aggregate error list.
`verification::normalize`	Removes non-ASCII-alphanumeric characters and lowercases; used for metric/blocked-column textual normalization after AST parsing.
`SchemaReferences:: pre_visit_relation`	SQL visitor hook collecting lowercase relation names.
`SchemaReferences:: pre_visit_select`	SQL visitor hook collecting projection aliases so they are not mistaken for source columns.
`SchemaReferences:: pre_visit_expr`	SQL visitor hook collecting simple identifiers and final components of compound identifiers.