Runtime

Use this page to understand what the reviewed AIP runtime owns between trusted gateway dispatch and connector execution. It explains how capability contracts, policy, idempotency, approvals, transactions, queues, callbacks, and recovery for

SECTION	Architecture and Security
TYPE	Architecture
PROTOCOL	AIP 1.0
SOURCE	docs/architecture/runtime.md

The page describes version `1.0.0` of the Rust workspace at source revision `97be86e9efedf07ecf1783b03800f683f107fb04`. `aip-runtime` is a reusable in-process service composed by gateways and SDKs. Its service layout and storage choices are implementation architecture, not a topology required by AIP 1.0.

Place the runtime after trusted ingress

The gateway establishes the authenticated actor, verified tenant, credential handle, identity context, session, and correlation coordinates. The runtime uses that trusted context to govern work, select a capability implementation, record lifecycle evidence, and return a native AIP result or queued acknowledgement.

Boundary	Runtime owns	Runtime does not own
Admission	Capability lookup, schema and contract checks, policy, idempotency, approval, and transaction gates	Listener authentication, signature verification, or payload-to-identity trust
Execution	Bounded handler invocation with trusted context, cancellation, streaming, and checkpoints	Product data-transfer objects, provider credentials, or provider API semantics
Lifecycle	Results, events, approvals, transactions, delegations, queue state, callbacks, and query views	External observability storage or deployment-specific reporting
Recovery	Lease-aware queued work, callback retry, delegation replay, and a structured restart report	Process supervision or proof that an uncertain provider effect did not occur

The runtime can call a local handler or a catalog-resolved remote handler. Both paths enter the same action governance. Moving a connector into a fleet does not create a second authorization or lifecycle model.

Compose one coherent state bundle

`RuntimeStores` supplies 13 services as one deployment contract. The runtime records its durability class for the complete bundle; an embedder must not mix ephemeral and durable implementations when replay or side-effect fencing depends on restart safety.

Service	State or decision it owns
Storage health	A bounded read-and-write readiness check for the selected backend
Maintenance	Retention of events, replay claims, dead letters, callback records, and published outbox records
Replay	Atomic claims for signed envelope message IDs

Service	State or decision it owns
Profile state	Versioned, compare-and-set projections and bindings owned by protocol profiles
Sessions	Session creation, state, and resume data
Idempotency	Action intent reservations, ownership, input binding, and resolved results
Events	Append-only lifecycle and operational events with cursor reads
Lifecycle	Results, chunks, cancellations, escalations, conversations, receipts, audits, and settlements
Delegations	Parent-child graph records and durable delegation outcomes
Approvals	Requests, decisions, leases, and resumable approval state
Transactions	Plans, state transitions, provider checkpoints, and recoverable records
Action queue	Queued and running actions, leases, retries, terminal records, and dead letters
Callback delivery	Durable callback attempts, lease state, retry windows, and terminal views

The discovery service, admitted handler map, active-action map, policy object, routers, dispatchers, and work semaphores are runtime composition rather than additional entries in `RuntimeStores`.

Three durability classes make the deployment promise explicit:

- `ephemeral` is process-local and loses state on restart;
- `durable_single_process` survives restart but coordinates one process at a

time; and

- `durable_shared` survives restart and relies on backend-enforced coordination

between processes.

The built-in local file bundle is `durable_single_process`. The reviewed PostgreSQL bundle is `durable_shared` and supplies all 13 services from one pool. Its queue uses transactions, row locks, skip-locked selection, and lease-identity checks. Merely serializing queue records does not provide the shared coordination guarantee.

The default runtime uses in-memory services. That is useful for embedding and tests, but its always-ready health implementation and no-op retention service are not evidence of durable production readiness.

Give handlers bounded trusted context

An action payload is not the handler's complete authority. Before invocation, the runtime builds a non-serializable `ExecutionContext` from trusted and validated state.

Context value	Why it is present
Authenticated actor and verified tenant	Bind execution to transport-established identity and membership
Opaque credential handle	Let the deployment credential provider resolve secret material without placing it in protocol state
Deadline and cancellation token	Bound execution and support cooperative cancellation
Idempotency reservation	Identify the one owned action intent for this attempt
Verified approval set	Carry durable decisions and policy hashes that passed runtime verification
Transaction context	Carry the stable transaction ID and recovered provider-operation coordinates
Transaction checkpoint publisher	Persist a provider operation and reconciliation cursor at the uncertainty boundary
Execution checkpoint publisher	Expose ordered, bounded crash-qualification observations without trusting payload telemetry
Stream publisher	Persist and publish incremental chunks through runtime lifecycle state

Context value	Why it is present
Trace and redaction policy	Carry trusted correlation and deny sensitive diagnostic fields

The context is deliberately not an AIP schema and is not serialized into an action. Durable stores retain the verified source records needed to reconstruct it for another attempt. A connector cannot obtain more authority by adding look-alike fields to its action input.

Admit capability and implementation together

A local manifest becomes callable only when every non-resource capability has a handler whose declared implementation support satisfies the capability contract. Runtime admission validates the complete manifest-handler set under one admission lock and publishes discovery and handler state together.

A compatibility registration method revalidates the owning manifest before it replaces one handler. A local capability present in discovery without its admitted handler fails as `HandlerNotFound`; the runtime does not silently send it to a remote process.

For fleet capabilities, the runtime looks in bounded local discovery first and then asks the tenant-scoped capability catalog. Only a capability absent from local discovery can use the configured catch-all remote handler. This ordering prevents a remote catalog entry from shadowing a local contract.

Follow one synchronous action

The synchronous path is ordered so an external handler sees only an action that has passed all applicable gates.

```
MERMAID
flowchart LR
    T["Trusted message context"] --> C["Resolve and validate capability"]
    C --> P["Authorize policy"]
    P --> I["Reserve idempotent intent"]
    I --> A["Validate or request approval"]
    A --> X["Apply transaction gate"]
    X --> H["Select admitted handler"]
    H --> E["Execute with bounded context"]
    E --> V["Validate output"]
    V --> D["Persist lifecycle and transaction state"]
    D --> S["Settle idempotency result"]
```

The text equivalent is:

1. The runtime replaces action identity with the gateway's resolved identity.

It normalizes transaction identifiers and restores reconciliation coordinates when applicable.

2. It rejects an action ID that aliases a different durable queued action.
3. It resolves the capability from local discovery or the tenant-scoped remote

catalog, then validates input schema, mode, contract, and credential context.

4. It applies capability policy and any transaction-specific policy

restrictions.

5. It derives the idempotency claim and either owns a bounded reservation,

returns the already resolved result, or rejects a conflicting intent.

6. It runs any transaction preview that must precede approval, validates

supplied approval evidence, and creates a pending approval when policy requires a human decision.

7. It applies the remaining pre-execution transaction operation. A dry run,

plan, or other non-executing branch can return here with durable evidence.

8. It enriches only the connector-facing action, resolves compensation routing

when needed, and selects the admitted local or remote handler.

9. It constructs trusted execution context, marks the action active, invokes

the handler under the effective timeout, and removes the active entry.

10. It validates the output contract, persists the result and terminal action

record, records transaction outcome, and settles the idempotency reservation.

The runtime records ordered execution checkpoints around intent persistence, provider invocation, response receipt, result persistence, and reconciliation. These observations help qualify crash boundaries; they do not make a provider effect atomic with local storage.

Queue asynchronous work before execution

An asynchronous submission performs capability, contract, credential, policy, idempotency, approval, transaction, and handler-availability checks before it returns `queued`. It then stores the action, trusted principal and message context, retry policy, timestamps, and owned idempotency reservation in the action queue.

The acknowledgement means the reviewed queue accepted the action. It does not mean that a worker started, the provider accepted the operation, or a callback was delivered.

A worker atomically leases either the next eligible action or a named action. While execution is active, a heartbeat renews both the queue lease and the transferred idempotency reservation. Loss of the lease prevents that worker from claiming terminal ownership.

The worker re-enters the normal execution path with the transferred reservation instead of acquiring a new intent. It then completes the attempt against the same lease identity.

Attempt outcome	Queue behavior	Lifecycle meaning
Success or terminal non-retryable result	Settle the queue record	Persist terminal result and emit worker evidence
Retryable result within policy	Remove the lease and set `next_attempt_at`	Keep the action non-terminal; do not publish a final callback yet
Exhausted or unretryable failure	Move the attempt to dead-letter state	Persist terminal failure evidence
Cancellation won before completion	Preserve cancelled state	Do not let the stale worker overwrite cancellation
Lease lost	Reject worker settlement	Another worker or recovery cycle may own the record

Retry eligibility considers the capability policy, error category, attempt and elapsed budgets, idempotency requirements, and the future retry window. The finite worker drain stops after bounded idle polling or a claim limit. A daemon or process supervisor owns the long-running loop, shutdown, and backpressure policy.

Keep streams, callbacks, and completion separate

The stream publisher persists chunks through lifecycle state and can publish them without waiting for the terminal handler result. Cancellation reaches the active handler through the same trusted execution context.

After a terminal asynchronous result, the runtime records an event and, when a callback exists, creates or advances durable callback delivery state. Callback work has an independent concurrency budget, lease, retry schedule, receipt chain, and terminal view. A completed action and a delivered callback are therefore distinct facts.

Callback recovery claims only a bounded batch that fits the available callback budget. Reconciliation has a separate semaphore, so a callback storm cannot consume the capacity reserved for uncertain transaction work.

Treat provider uncertainty as recoverable state

The runtime can persist a provider operation ID and optional reconciliation cursor before a connector awaits an uncertain commit response. A later transaction reconciliation reconstructs that context and runs under its own work budget.

This boundary reduces blind retries, but it cannot guarantee exactly-once external effects. A crash can still occur between a provider effect and its durable checkpoint, and a provider may not offer an idempotent lookup. The connector and provider contract determine whether reconciliation can establish the outcome.

Approval, idempotency, replay, and transactions solve different problems:

Mechanism	Question answered
Replay claim	Has this signed envelope message ID already been accepted in its time window?
Idempotency reservation	Does this caller-owned action intent already have an owner or result?
Approval record	Did an authorized decision satisfy the immutable policy and action binding?
Transaction record	What plan, external-operation evidence, outcome, or compensation state survives?

None of these records alone proves provider success.

Recover explicitly after restart

`recover_runtime_state` produces one auditable report rather than hiding restart work inside construction. Its configuration independently controls five categories:

1. include pending approvals in the report;
2. include recoverable transaction or saga records;
3. lease and replay queued or running actions;
4. replay running delegations through the remote router or local fallback; and
5. claim and retry recoverable callback deliveries.

The report separates successful queued and delegation results from typed errors, and includes callback delivery views. Listing a pending approval or recoverable transaction is not the same as deciding or reconciling it. The embedding supervisor remains responsible for sequencing recovery after handlers, catalogs, routers, dispatchers, and durable stores are ready.

The local file store can recover state for one process. Shared workers require a backend whose lease, renewal, completion, compare-and-set, and idempotency operations coordinate every process. The reviewed PostgreSQL implementation is the reference shared backend for that contract.

Operate readiness and retention as narrow controls

Storage health checks only that the configured runtime backend can perform its bounded readiness operation. It does not probe a connector host, provider, callback destination, registry, transport listener, or process supervisor.

Retention maintenance deletes bounded operational families according to the configured event, dead-letter, callback, and outbox windows, and also expires replay claims. Operators must choose windows that preserve their audit, recovery, and replay requirements. The protocol does not prescribe those deployment values.

Failure ownership and non-implications

Failure	Owning runtime decision
Capability absent or local handler missing	Reject before connector execution
Input, mode, credential, or policy mismatch	Return a typed failed result or runtime error before execution
Idempotency conflict	Reject the different intent without reusing its result
Approval required	Persist pending state and stop before execution
Queue lease or reservation lost	Refuse stale worker settlement
Retry budget exhausted	Persist terminal failure and dead-letter evidence
Callback target unavailable	Retain independent callback retry state
Provider outcome uncertain	Preserve transaction coordinates for explicit reconciliation when supported
Durable backend unavailable	Fail the state transition rather than claim durability

The runtime does not authenticate transport connections, hold raw provider secrets, define product payloads, start connector processes, or supervise an infinite worker loop. Its support for a store, handler, transaction mode, callback, or recovery method does not prove that a deployment enabled or qualified that path.

Related pages

- Actions and sessions
- Approvals and policy
- Transactions and compensation
- Gateway
- Observe and recover