How AIP works

AIP turns a capability request into an identified, governed, and observable unit of work. This page is for developers who need to understand what happens between discovery and a terminal result before choosing an API, profile, connector, or

SECTION	Start Here
TYPE	Getting started
PROTOCOL	AIP 1.0
SOURCE	docs/getting-started/how-aip-works.md

The semantic objects described here belong to AIP 1.0. Component behavior describes the Rust implementation at source revision `d7cce13d1d555644d04a4d73c66c95b113737635`. AIP does not require every implementation to use the same process layout or storage backend.

The problem is larger than request delivery

HTTP, a queue, or a tool protocol can deliver a payload. Delivery alone does not settle identity, tenancy, approval, or whether a retry represents the same intent. It also does not define recovery when the provider may have committed but its response was lost.

AIP separates those concerns into durable semantic objects:

Concern	Main objects	Stable question answered
Discovery	`Manifest`, `Capability`, `CapabilityContract`	What can be requested, in which mode, and under which declared safety conditions?
Identity	`Principal`, `Envelope`, `IdentityContext`	Who is acting, through which trusted boundary, and for which tenant or external account?
Work	`Action`, `Ack`, `StreamChunk`, `ActionResult`	What logical operation was submitted, accepted, streamed, or completed?
Governance	approval, idempotency, transaction, and delegation objects	Which authority, replay rule, or recovery contract governs the operation?
Observation	`ActionStatus`, action events, receipts, and audit views	What does the receiving implementation durably know now?

The envelope carries one protocol message. The action identifies the logical work. The operational read model survives beyond one transport exchange when a durable backend is configured.

The end-to-end action flow

The following sequence shows the reviewed Rust implementation. Boxes represent responsibilities, not a required AIP deployment layout.

```
MERMAID
sequenceDiagram
    participant C as Client or profile adapter
    participant E as Authenticated edge
    participant G as AIP gateway
    participant R as Durable runtime
    participant H as Local or remote handler
    participant P as External product

    C->>G: Discover manifest or tenant catalog
```

```
    C->>E: Submit Action in an Envelope
    E->>G: Verified actor and transport context
    G->>G: Freshness, replay, identity, and query policy
    G->>R: Action plus trusted MessageContext
    R->>R: Contract, policy, idempotency, approval, transaction
    alt asynchronous action
        R-->>C: Ack queued
        R->>H: Worker dispatches admitted action
    else synchronous or streaming action
        R->>H: Dispatch admitted action
    end
    opt remote product capability
        H->>P: Provider-shaped request
        P-->>H: Provider response or uncertain outcome
    end
    H-->>R: Chunks and terminal result
    R-->>C: Result, stream, or callback
    C->>G: Authorized status, event, or receipt query
    G->>R: Read scoped durable state
    R-->>C: Current operational view
```

In text, the flow is:

1. The caller discovers a callable contract from a manifest or, for a remote

fleet, a tenant-scoped capability catalog.

2. A transport edge authenticates the peer and passes verified identity data to

the gateway. A profile adapter may first translate an MCP tool call or A2A task operation into the same native action model.

3. The gateway validates the envelope boundary, replaces payload identity with

the authenticated actor, resolves trusted tenant and credential context, and applies replay and query policy.

4. The runtime resolves the capability, validates the action against its

contract, applies authorization, acquires any idempotency reservation, and handles approval or transaction preconditions.

5. The runtime either returns a governed intermediate state, queues the action,

or dispatches it to an admitted local or remote handler.

6. The handler emits ordered chunks when applicable and returns a terminal

result. A connector may call an external product behind its own credential and network boundary.

7. The runtime records the lifecycle data supported by its backend. Authorized

readers can query that state independently of the original response.

A callable surface exists before work is accepted

Discovery and execution are coupled deliberately. For local capabilities, the runtime's production admission path validates the manifest, schemas, profile identifiers, and implementation claims. It then requires the callable capability IDs to match the supplied handler IDs before publishing discovery and handler state together. A declared operation cannot become callable through this path without an implementation owner.

Remote fleet capabilities use a different boundary. The product-neutral daemon installs the tenant-scoped catalog and one shared remote handler as an inseparable pair. The normalized connector registry supplies admitted capability bindings and ready replicas; product implementations remain in standalone connector hosts. The daemon therefore does not load product crates or product credentials to expose their capabilities.

Discovery is still a snapshot, not proof that an external product will remain reachable. Admission, current replica readiness, caller authorization, and provider availability are separate decisions.

Trusted ingress replaces payload claims

An AIP envelope can contain `from`, `to`, session, correlation, idempotency, and security metadata. Those fields describe the message, but a self-asserted `from` value does not authenticate a caller.

The reviewed gateway has separate entry points for a directly verified envelope, an authenticated edge, and a complete deployment-owned identity resolution result. Each trusted path replaces `Envelope.from` with the verified principal. For actions, the gateway can resolve tenant membership and a credential handle before dispatch; the runtime then replaces the action's payload-supplied identity context with that resolved context.

Signed native envelopes add integrity and peer verification. The gateway also uses a state-backed message-ID replay claim and a freshness window when replay rejection is enabled. Neither a valid signature nor a fresh message grants permission by itself. Capability policy, scopes, tenant membership, object ownership, delegated authority, and approval remain distinct checks.

Raw provider credentials do not belong in manifests, action inputs, or identity claims. Capability contracts describe credential requirements, while the deployment resolves a protected credential handle and the connector owns the provider-specific secret boundary.

Governance happens before handler execution

For a normal action, the reviewed runtime performs the following decisions before calling a handler:

1. It rejects reuse of an existing action ID for different work when durable

action state already exists.

2. It resolves the applicable capability for the trusted context.
3. It validates the input schema, invocation mode, capability contract, and

resolved credential context.

4. It evaluates authorization and any transaction-specific policy.
5. It derives the declared idempotency scope and acquires a durable reservation

when the contract requires one.

6. It validates an attached approval decision or creates a durable pending

approval when policy requires human authority.

7. It applies supported transaction preconditions, prepares a frozen execution

context, and selects the admitted handler.

An action ID and an idempotency key solve different problems. The action ID names one logical action and its lifecycle. An idempotency key fences replay of a provider-sensitive intent within the scope declared by the capability. A new action ID or key can represent new work; it is not a recovery mechanism for an ambiguous mutation.

Execution follows the declared mode and contracts

The action mode changes the immediate protocol response, while the capability contract limits which modes and recovery behaviors are valid.

Path	Immediate behavior in the reviewed runtime	Durable continuation
Synchronous	The runtime executes the handler and returns an `ActionResult`	The terminal result and lifecycle record are persisted when the selected backend supports them
Asynchronous	The runtime reserves governance state, enqueues the action, and returns an `Ack` with status `queued`	A worker leases the same action record, executes it, and settles its result

Path	Immediate behavior in the reviewed runtime	Durable continuation
Streaming	The handler can emit monotonic `StreamChunk` records before returning an `ActionResult`	Chunks and the terminal result can be read independently of the live stream
Pending approval	The runtime returns an action result with status `pending_approval` and retains the approval request	A verified decision advances the same governed action rather than creating unrelated work
Transactional	`dry_run`, `plan`, `commit`, `compensate`, `reconcile`, or `rollback_not_supported` behavior is checked against the capability contract	Plans, provider operation checkpoints, transaction state, and reconciliation data remain correlated with the action

Cancellation is cooperative. The runtime signals the active handler and calls its capability-specific cancellation implementation only when the admitted support and contract permit cancellation. A cancellation request cannot prove that an external provider stopped unless the connector can establish that outcome.

Local and fleet handlers share the runtime contract

A trusted local module executes inside the daemon's startup composition. Its handler receives a frozen execution context containing the authenticated actor, verified tenant and credential state, cancellation token, stream publisher, approval evidence, timeout, and transaction checkpoint support relevant to the capability.

A fleet capability reaches the same runtime through the shared remote handler. That handler requires a verified tenant, applies bounded fair admission, asks the registry for a deterministic route assignment, and dispatches the action to the assigned connector host. The assignment is reused for retry and cancellation. The host executes exactly its compiled product adapter and returns a signed, correlated native result.

If remote dispatch fails after the provider may have changed state, the route settlement records an unknown outcome rather than declaring the operation safe to repeat. A transactional connector can persist a provider operation reference and reconcile it. The external product remains the source of truth for its own state.

Durable observation is separate from the response

The original exchange can return an acknowledgement, chunks, a final result, or an error. It cannot tell a disconnected caller what happened afterward. The runtime therefore exposes action status, result, event, receipt, approval, transaction, callback-delivery, session, audit, and resource read models as separate protocol families.

The daemon's HTTP read routes convert those queries into native message bodies and send them through the same gateway policy boundary. Query authorization can enforce actor scopes, tenant selection, session ownership, and broader `read:any` authority. Knowing an action ID is not sufficient authorization to read it.

Durability depends on deployment composition. The implementation offers in-memory, file-backed, and PostgreSQL-backed paths. In-memory state does not survive process loss, file-backed state is local to one durable directory, and the reference clustered topology uses PostgreSQL so gateway replicas can share operational state. A protocol lifecycle object does not by itself select or qualify one of those backends.

Trust and data boundaries

Boundary	Trusted input created there	Data that remains untrusted or externally owned
Client to edge	Authenticated principal, transport credential status, and request limits	Payload sender claims and requested tenant selectors
Edge to gateway	Verified actor and, when resolved, tenant and credential handles	Permission to invoke a specific capability until policy evaluation
Gateway to runtime	Frozen message context, selected capability contract, and governed identities	Provider outcome and any undeclared handler behavior
Runtime to connector host	Assigned action, bounded execution context, approval and idempotency state	Raw product secret material, which the host resolves locally
Connector to product	Provider-specific authenticated request	Provider availability, commit semantics, and system-of-record state
Runtime to observer	Authorized lifecycle and evidence view	Conclusions beyond the retained artifact, topology, and time window

These boundaries let an implementation fail closed without pretending that one signature, database, or receipt establishes every kind of trust.

Design choices and trade-offs

- **One semantic core, several integration surfaces.** Native HTTP, NATS, SSE,

and WebSocket bindings coexist with MCP and A2A compatibility profiles. Each surface still has its own authentication, delivery, and projection limits.

- **Explicit lifecycle over one-shot inference.** Durable status and event

queries make disconnect recovery possible. They require storage, retention, authorization, and operational maintenance.

- **Declared contracts plus admitted support.** Contracts expose risk and

recovery behavior before execution. They remain declarations until an implementation path validates and enforces them.

- **Product-neutral gateway plus standalone hosts.** The fleet can add or scale

an admitted connector without recompiling `getaip-server`. It adds registry, lease, routing, host identity, and remote-failure concerns.

- **Idempotent intent rather than an exactly-once promise.** Durable reservations

and provider keys reduce duplicate effects. Ambiguous provider commits still require reconciliation or a conservative terminal state.

What this flow does not mean

- AIP 1.0 does not require one Rust daemon, one database, or one transport.
- A declared capability does not prove that a particular build or deployment is

conformant, qualified, or ready for production.

- A successful response does not authorize another reader to inspect the action.
- A queued or pending-approval response is not a terminal provider result.
- A cancellation request, retry flag, receipt, or idempotency key does not turn

an uncertain external mutation into a guaranteed rollback or exactly-once effect.

- Compatibility profiles project supported semantics; they do not make every

MCP or A2A feature identical to native AIP.

Related pages

- Capabilities and contracts
- Actions and sessions
- Identity and trust
- Architecture overview
- AIP 1.0 specification