What AIP is

Agent Interoperability Protocol (AIP) gives agents, applications, tools, and workflows a shared contract for discovering capabilities and managing work that extends beyond a one-shot call. This page is for evaluators deciding whether AIP fi

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

The normative wire protocol is AIP 1.0. Implementation details on this page apply to version `1.0.0` of the Rust workspace at the source revision recorded in page metadata. An implemented feature or a passing deterministic test does not by itself qualify a deployment for production.

The interoperability problem

Applications already expose APIs, tools, webhooks, queues, and agent-specific protocols. Those interfaces can carry a request. They do not necessarily give every participant the same model for capability discovery, identity, approval, retry safety, cancellation, delegated authority, lifecycle state, or an external outcome that is temporarily unknown.

AIP defines a semantic contract around that work. A participant can describe what it offers, a caller can submit a typed action, and both sides can refer to the same lifecycle and evidence objects. The application or external product continues to own its domain logic and system of record.

The AIP mental model

AIP answers five questions with explicit protocol objects:

Question	Main AIP objects	What they establish
What can this participant do?	`Manifest`, `Capability`, `Resource`	Provider identity, available operations, input and output schemas, supported profiles, resources, and declared limits or security metadata
Who is acting, and for whom?	`Principal`, `Envelope`, `IdentityContext`, delegation chain	Claimed sender and recipient, tenant and external-account context, human or service actors, credential references, and delegated scope
What may the operation do?	`CapabilityContract`	Declared side effects, idempotency, execution and retry support, data handling, credential requirements, approval, transactions, and compensation
What state is the work in?	`Session`, `Action`, `ActionStatus`, `Cancel`, `Event`, `StreamChunk`	Stable identifiers, lifecycle state, cancellation and streaming support, and terminal status
What was recorded?	`ActionResult`, `ReceiptChain`, `AuditEvent`, transaction views	Result data and bounded execution, receipt, audit, and transaction records

A capability contract is a declaration that callers and runtimes can validate; its presence is not proof that every deployment enforces the declaration. The capabilities and contracts documentation owns the detailed contract model.

How work moves through AIP

A typical AIP exchange has five stages:

1. **Discover.** A participant returns a manifest containing its identity,

capabilities, profiles, resources, and declared boundaries.

2. **Establish trust.** Before privileged work, a trusted edge authenticates the

caller and binds protocol identity and tenant context to deployment-owned evidence.

3. **Submit.** The caller sends an `Action` inside an AIP `Envelope`, using

stable identifiers for capability, message, session, and correlation where applicable.

4. **Govern and execute.** The receiving implementation validates the request,

applies authorization and approval policy, and dispatches the action to a native handler or product connector.

5. **Observe.** The caller receives an acknowledgement, can follow lifecycle or

stream events, and can query the final result and associated records.

The protocol does not require one physical topology. In the reviewed Rust implementation, the gateway, runtime, storage, profiles, and product connectors remain separate components with explicit responsibilities. Read How AIP works for the detailed lifecycle.

Trust and data boundaries

The protocol carries identity, policy, and evidence data, but those values do not create trust by themselves.

- **Identity:** `Principal`, `Envelope.from`, and `IdentityContext` contain

claims and references. A deployment can treat them as authority only after a trusted edge authenticates the caller and resolves tenant membership.

- **Integrity and authorization:** the Rust implementation keeps signature and

canonicalization support in `aip-crypto` and policy primitives in `aip-auth`. A valid signature protects message integrity; it does not grant permission to invoke a capability.

- **Credentials:** capability contracts describe credential requirements, and

actions can carry credential references. Raw connector secrets remain at the connector or deployment boundary rather than in manifests, action metadata, protocol errors, or audit records.

- **Provider state:** an external product remains the source of truth for its

side effects. Idempotency, retry, transaction, compensation, and reconciliation contracts describe how uncertainty is handled; they do not turn a non-idempotent provider into an exactly-once system.

- **Evidence:** lifecycle views, receipts, and audit events are scoped records.

Conformance and qualification require separate procedures and exact artifact identities.

Read Identity and trust for the full trust model and Conformance and qualification for the evidence boundary.

Integration choices and trade-offs

AIP keeps one native semantic core while allowing different integration surfaces:

Surface	What it preserves	Boundary or trade-off
Native AIP	Native envelopes, capabilities, actions, lifecycle, and evidence objects without translation	The participant adopts the AIP model directly

Surface	What it preserves	Boundary or trade-off
Transport binding	Native AIP messages over HTTP, NATS, SSE, or WebSocket	The binding changes delivery behavior, not native message meaning
Compatibility profile	Existing MCP or A2A client behavior mapped to AIP	Only the correspondence defined by the selected profile is available; foreign DTOs remain outside `aip-core`
Product connector	An existing product API and event model exposed as AIP capabilities	The product remains the domain authority; declared capabilities require separate implementation and test evidence

Compatibility profiles improve access from existing protocol clients. Product connectors preserve the external product boundary. Native AIP avoids a profile mapping when an integration needs the native lifecycle model.

AIP is a fit when independently operated participants need shared semantics for identity, governed mutations, lifecycle, recovery, or evidence. A direct API or tool call is usually the smaller contract when one owner controls both sides and none of those cross-system semantics is required.

What AIP does not provide

AIP does not choose an agent model, planner, business workflow, or provider outcome. It does not replace an application's database, authorization source, policy engine, or secret manager. It also does not make a self-asserted identity trustworthy, guarantee exactly-once external effects, or qualify an untested build or deployment.

Profiles and connectors are integration boundaries, not alternative branches of the native wire protocol. They translate external systems without adding their product-specific objects to the AIP semantic core.

Related pages

- Quickstart
- How AIP works
- Profiles, transports, and connectors
- Identity and trust
- AIP 1.0 specification