Dependency graph

Use this page to decide where a change belongs in the reviewed Rust workspace and which lower-level contracts it may depend on. The graph groups crates by architectural responsibility so native AIP semantics do not drift into a transport, c

SECTION	Architecture and Security
TYPE	Architecture
PROTOCOL	AIP 1.0
SOURCE	docs/architecture/dependency-graph.md

The page describes version `2.0.0` of the workspace at source revision `d7cce13d1d555644d04a4d73c66c95b113737635`. Cargo manifests are the source of truth for exact direct edges. The grouped graph is a design map, not a replacement for the manifests.

Read every arrow as “depends on”

An arrow points from a consumer to the lower-level crate or contract it imports. For example, `aip-runtime --> aip-core` means the runtime consumes native AIP types. It does not mean that the semantic core knows how to execute an action.

The pinned direct workspace graph is acyclic. That property matters because a lower layer can define a stable contract without importing the process that eventually composes it.

Compile-time dependency is not the same as runtime traffic:

- a binary can depend on a transport crate without enabling every listener;
- a storage adapter depends on the traits it implements even though the runtime

receives the adapter through composition;

- the public facade depends on crates selected by features, but those crates do

not depend back on the facade; and

- a product host depends on one connector adapter without making that adapter a

dependency of the central daemon.

Follow the primary ownership layers

```
MERMAID
flowchart TB
    P["Central process: getaip-server"]
    H["Six connector-host binaries"]
    L["Client and control CLI: getaip"]
    X["Shared connector-host bootstrap"]
    C["Composition: gateway, MCP edges, remote dispatch"]
    F["Fleet control: registry, admission, orchestration, lifecycle control"]
    S["Storage adapters: runtime PostgreSQL and registry PostgreSQL"]
    R["Runtime governance: lifecycle, policy, queues, connector contract"]
    B["Bindings: native transports and compatibility profiles"]
    D["Domain services: authentication, cryptography, discovery, schema"]
    K["Semantic foundation: aip-core"]
    A["Feature-gated aip facade"]

    P --> C
    P --> F
    P --> S
    H --> X
    X --> C
    X --> F
    X --> S
    L --> F
    L --> B
    L --> D
    C --> R
    C --> B
    C --> F
    F --> R
    F --> D
    S --> R
    S --> F
    R --> D
    R --> K
    B --> K
    D --> K
    A --> C
    A --> R
    A --> B
    A --> D
    A --> K
```

The text equivalent is:

1. `aip-core` owns native protocol data and validation-independent semantics.
2. Schema, cryptography, authentication, and discovery build on that semantic foundation.
3. Runtime governance and the connector contract consume those domain services to validate and execute native work.
4. Transports and compatibility profiles translate or carry protocol data without becoming owners of the native lifecycle.
5. Gateways, remote dispatch, MCP edges, and connector-host bootstrap compose the runtime with selected bindings.
6. Fleet registry, admission, orchestration, lifecycle control, and their concrete storage adapters remain separate control and persistence paths.
7. The central daemon, six product-host binaries, and the client or control CLI consume different outer combinations instead of sharing one executable dependency tree.
8. The `aip` facade is a feature-gated consumer convenience layer, not a new

owner below those components.

Some direct Cargo edges cross the visual layers. For example, a gateway imports authentication and cryptography directly, while the PostgreSQL runtime adapter imports profile session contracts as well as runtime storage traits. Those edges implement a bounded responsibility; they do not transfer ownership of the whole lower layer.

Keep native semantics at the foundation

`aip-core` has no workspace dependency. It defines the native message model, identifiers, capability and lifecycle values, errors, and validation methods used throughout the workspace. Its optional schema feature adds derive support; it does not import the schema registry, runtime, transport, or a connector.

`aip-schema` consumes core types to generate and validate JSON Schema. `aip-crypto` and `aip-auth` consume core identities and messages for cryptographic and authorization decisions. `aip-discovery` consumes core and schema contracts to admit manifests and expose capabilities.

This direction creates a practical rule: a protocol field required across transports and profiles belongs in the semantic or schema layer. It must not be defined first in an HTTP route, an MCP mapping, or a product adapter and then leak upward through conversions.

Let runtime depend on policy contracts, not products

`aip-runtime` depends on authentication, cryptography, discovery, schema, the product-neutral connector registry contract, and core. It owns action governance, lifecycle state, queues, approvals, transactions, delegations, callbacks, and recovery.

The runtime has no dependency on any of the six maintained product adapters, connector-host binaries, compatibility servers, or deployment processes. Product behavior reaches the runtime only through an admitted action handler or the tenant-scoped remote catalog and remote handler.

`aip-connector` depends on core, discovery, and runtime because it defines the connector-facing implementation contract, credential boundary, and lifecycle integration. Product adapters depend on that contract. Reversing this edge would make generic execution depend on a particular provider.

Keep bindings parallel to lifecycle

`aip-transport` defines native transport abstractions over core messages. Protocol-specific transport crates implement HTTP, NATS, SSE, or WebSocket bindings by depending on that abstraction and the core model.

Compatibility profiles map foreign semantics into native AIP. MCP client and server packages depend on the MCP profile and session contracts; the MCP server also composes the gateway. The gateway itself does not depend on the MCP profile. This keeps the native action lifecycle usable without importing an MCP server.

The A2A profile is a gateway dependency for its bounded native integration. That compile-time edge does not mean every gateway deployment exposes an A2A endpoint.

Profiles and transports may depend on native semantics. Native lifecycle code must not depend on a wire listener or a foreign protocol mapping to decide what an action, approval, transaction, or result means.

Compose at the gateway boundary

`aip-gateway` combines trusted ingress, replay protection, identity resolution, query authorization, runtime dispatch, remote delegation, and callbacks. It consumes the runtime, generic connector contract, selected native transport abstractions, and the profile support it directly owns.

The gateway is still a library. Listener construction and deployment configuration belong to an executable or embedding application. A new listener therefore depends on the gateway; the gateway should not import the

process that starts it.

The remote connector package implements the runtime's catalog and handler extension points using the connector registry and gateway peer clients. The central daemon depends on this product-neutral path and does not import the six product implementations.

Separate fleet control from provider execution

Fleet packages divide authority rather than forming one large connector crate:

Package responsibility	Depends on	Must remain independent of
Registry contract	Core, cryptography, and discovery definitions	Product API clients and host processes
Admission	Registry contract, cryptography, discovery, and core	Process startup and provider execution
Orchestration contract	Admission and registry contracts	Deployment-platform implementation details
Registry PostgreSQL adapter	Registry and admission-owned storage contracts	Runtime product execution
Connector host	Gateway, runtime, connector contract, registry views, and bounded transport support	Selection of another tenant or connector instance
Host bootstrap	Host composition, trusted peer setup, and complete runtime storage	Product-specific capability implementation
Lifecycle control process	Host lifecycle and registry control contracts	Catalog admission authority and provider credentials
Product host binary	Shared bootstrap plus exactly one maintained adapter	Central daemon composition

The concrete storage and process crates sit outside the contracts they implement. Their Cargo arrows point inward because dependency inversion lets a runtime or control service accept an implementation without importing a specific database or executable.

Put storage outside the state owner

`aip-storage-postgres` depends on runtime storage traits and constructs a complete shared-durability store bundle. The runtime does not depend on PostgreSQL. It can instead receive the in-memory, local file, or another contract-compatible bundle.

The registry PostgreSQL adapter follows the same pattern for fleet catalog, route, admission-journal, lease, and control state. A database schema can optimize and coordinate those records, but it does not become the owner of their protocol meaning.

This direction allows storage replacement without moving authorization, lifecycle, or routing rules into SQL-specific callers.

Treat binaries and the facade as outer consumers

`getaip-server` is the central product-neutral server composition. It depends on the gateway, runtime, shared storage, remote connector path, profiles, and transports needed for its configured surfaces. It does not depend on the six product adapters.

Each maintained connector-host executable combines shared host bootstrap with one product adapter. That is the intended point where provider-specific code enters an executable dependency tree.

`getaip` is a client, conformance, release-admission, and fleet-operation surface. It imports the contracts and transports needed for those tasks, but does not embed the gateway or become part of server action execution.

The `aip` crate is a feature-gated facade. Enabling one of its features makes the selected lower-level crate available to that consumer. It does not create a reverse edge, prove that a binary enables the feature, or make every optional component part of the core protocol.

Place a change by its owner

Change	Start in	Add outward dependencies only when
Native message field or invariant	`aip-core` and schema coverage	Bindings need serialization or mapping
Signature or principal rule	Cryptography or authentication	Gateway composition must invoke the decision
Manifest or capability admission	Discovery	Runtime or fleet admission must enforce it
Action lifecycle or recovery rule	Runtime	Gateway, storage, or operators must expose it
Transport encoding or listener behavior	Transport implementation or process	It calls the stable gateway boundary
Foreign protocol mapping	Corresponding profile	Client or server adapters need that mapping
Generic connector execution contract	`aip-connector`	Maintained adapters implement it
Product operation	One maintained adapter and its host	Public connector docs expose the mapping
Fleet route or release rule	Registry, admission, or orchestration contract	Adapter and process implement the contract
Database coordination	Concrete storage adapter	The owning service accepts it through a trait
Command or daemon wiring	`getaip`, `getaip-server`, or host binary	No lower layer imports the executable

When a proposed lower-level crate needs a type from a higher-level process, first ask whether the type is actually a missing semantic contract. Moving that small contract inward is usually safer than introducing a reverse dependency.

Interpret the graph narrowly

Cargo metadata proves compile-time edges at the reviewed revision. It does not prove runtime enablement, network reachability, production qualification, feature selection, storage durability, or provider availability.

The graph deliberately omits third-party dependencies, development-only tools, qualification packages, and exact edges that do not change architectural ownership. Add or remove a direct workspace dependency in Cargo first, then update this map when the ownership path changes.

Related pages

- Architecture overview
- Runtime
- Gateway
- Connector fleet architecture
- Transport bindings