Use the Rust SDK

Use this guide when a Rust application needs AIP 1.0 types, validation, or an embedded protocol service. You will pin the reviewed SDK source, construct and validate one native action envelope, and then select optional modules by role. The

SECTION	Build with AIP
TYPE	Guide
PROTOCOL	AIP 1.0
SOURCE	docs/guides/use-rust-sdk.md

At source revision `97be86e9efedf07ecf1783b03800f683f107fb04`, every AIP workspace package has `publish = false`. Consume that revision from source or from an approved mirror; do not assume that `aip = "1"` resolves to an AIP-owned registry release. The examples and commands on this page were checked against source without building the workspace or running an AIP service.

Prerequisites

You need:

- Rust `1.88` or newer, matching the reviewed workspace minimum;
- a Rust toolchain that supports edition 2024 packages;
- access to the reviewed source revision or an approved immutable mirror of it;
- permission to update and retain the project's `Cargo.toml` and `Cargo.lock`;
- a decision about whether the project needs only protocol semantics or also a

runtime, transport, profile, or connector role.

No provider credential is required for the semantic example. Do not place credentials, tenant identifiers, or deployment secrets in a Cargo feature or source URL.

1. Choose the dependency boundary

The `aip` facade is the single entry crate that re-exports the semantic core and feature-gated modules from role-specific crates. Start with the smallest boundary that owns the API you need.

Boundary	Use it when	Initial choice
`aip` facade with defaults	The application creates, parses, serializes, or validates native AIP values	Prefer for most semantic consumers
`aip` facade with selected features	The application embeds one or more re-exported service roles	Add only the named role features
`aip` facade with `full-core`	Integration or conformance work needs nearly every product-neutral facade module	Use deliberately, then inspect the dependency tree
A role-specific crate	A component needs a narrow compile-time boundary or a workspace role not re-exported by `aip`	Depend on that crate directly

The facade always re-exports `aip-core`. Its default features are the `std` and `json` compatibility markers; they do not enable the runtime, gateway, transport, profile, schema, storage, conformance, or connector modules.

The facade is not an inventory of every workspace crate. Fleet admission, orchestration, control-plane, host-bootstrap, and shared MCP-session roles are direct crates at the reviewed revision.

2. Pin the reviewed source

Add the facade and JSON support to your application. A Git dependency records the immutable revision in `Cargo.lock`:

```
TOML
[dependencies]
aip = { git = "https://github.com/getaip/core", rev = "97be86e9efedf07ecf1783b03800f683f107fb04" }
serde_json = "1"
```

If policy requires a vendored checkout, replace the Git dependency with the approved local path:

```
TOML
[dependencies]
aip = { path = "../aip-core/crates/aip" }
serde_json = "1"
```

A path dependency does not record the checkout's Git revision in the consumer lockfile. Record and verify the vendored source identity through the deployment or software bill of materials (SBOM) process responsible for that checkout.

3. Construct and validate an envelope

Create `src/main.rs` with a semantic-only example:

```
RUST
use aip::{
    Action, CapabilityId, Envelope, MessageBody, Principal, PrincipalId, PrincipalKind,
    validate_envelope,
};
use serde_json::json;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let action = Action::new(
        CapabilityId::parse("cap:example:echo")?,
        json!({ "message": "hello from Rust" }),
    );

    let mut envelope = Envelope::new(MessageBody::Action(Box::new(action)));
    envelope.from = Some(Principal::new(
        PrincipalId::parse("agent:sdk-example")?,
        PrincipalKind::Agent,
    ));

    validate_envelope(&envelope)?;
    println!("{}", serde_json::to_string_pretty(&envelope)?);
    Ok(())
}
```

`Action::new` generates an action identifier. `Envelope::new` derives the message type, generates a message identifier, sets `aip_version` to `1.0`, and records the current UTC time. The printed JSON therefore changes on every run.

`validate_envelope` checks semantic invariants such as the protocol version, the message-type/body match, identifiers, and body-specific requirements. It does not authenticate `from`, authorize the capability, admit a manifest, or dispatch the action. A gateway or protocol endpoint owns those boundaries.

4. Verify the semantic project

Generate and retain a lockfile before using `--locked`:

```sh
cargo generate-lockfile
cargo check --locked
cargo run --locked
```

Expected result: the project compiles and prints a JSON envelope containing `"aip_version": "1.0"`, `"message_type": "aip.core.v1.action"`, an `action` body, and the `agent:sdk-example` sender. Generated identifiers and `sent_at` will differ between runs.

Inspect the enabled facade features:

```sh
cargo tree --locked -e features -p aip
```

For the initial example, the facade should not show optional runtime, gateway, transport, profile, schema, storage, conformance, or connector features. A successful compile proves only that this consumer and dependency graph compile; it is not protocol conformance, deployment readiness, or provider qualification.

5. Add features by task

Enable a feature only when its public module belongs in the current component. Feature implications in this table are the explicit facade relationships at the reviewed revision.

Task	Facade feature	Important implication
Generate or validate against the schema registry	`schema`	Exposes `aip::schema`
Compose authentication and authorization primitives	`auth`	Exposes `aip::auth`
Sign, verify, or canonicalize native values	`crypto`	Exposes `aip::crypto`; it does not define authorization policy
Admit and query manifests	`discovery`	Exposes `aip::discovery`
Run durable action and session services	`runtime`	Also enables `auth` and `discovery`
Embed the gateway composition layer	`gateway`	Also enables `runtime`
Persist runtime state in PostgreSQL	`storage-postgres`	Also enables `runtime`
Use the shared transport abstraction	`transport`	Exposes `aip::transport` without selecting a wire binding
Use a native wire binding	`transport-http`, `transport-nats`, `transport-sse`, or `transport-websocket`	Each also enables `transport`
Translate a compatibility protocol	`profile-mcp`, `profile-a2a`, or `profile-webhook`	Exposes only the selected profile mapping
Embed an outbound MCP client	`mcp-client`	Also enables `profile-mcp`
Embed an AIP-backed MCP server	`mcp-server`	Also enables `gateway` and `profile-mcp`
Check the MCP compatibility mapping	`mcp-conformance`	Also enables `profile-mcp`
Use an MCP transport	`transport-mcp-stdio` or `transport-mcp-streamable-http`	Also enables `transport` and `profile-mcp`
Use the product-neutral connector contract	`connector`	Exposes `aip::connector`
Host one immutable connector artifact	`connector-host`	Also enables `connector` and `connector-registry`
Read or implement fleet registry contracts	`connector-registry`	Also enables `connector`
Persist the fleet registry in PostgreSQL	`connector-registry-postgres`	Also enables `connector-registry`
Dispatch to a remote connector host	`connector-remote`	Also enables `connector-registry`
Add metrics and tracing conventions	`observability`	Exposes `aip::observability`
Build deterministic tests	`testkit`	Exposes `aip::testkit`
Run implementation conformance checks	`conformance`	Exposes `aip::conformance`
Enable every product-neutral facade role	`full-core`	Excludes product connector features by an enforced source gate

For example, a component that embeds a gateway and exposes an HTTP transport can select those roles explicitly:

```
TOML
[dependencies]
aip = { git = "https://github.com/getaip/core", rev = "97be86e9efedf07ecf1783b03800f683f107fb04",
features = ["gateway", "transport-http"] }
```

Selecting a feature makes its module available. It does not configure an identity resolver, policy, storage backend, transport listener, manifest, handler, connector registry, or credential provider.

6. Use role-specific crates when ownership matters

Choose a direct crate when a component should expose only one role or when the facade does not re-export that role.

Role	Direct dependency	Boundary owned by the crate
Protocol semantics	`aip-core`	Native types, identifiers, serialization, and pure validation
Schema tooling	`aip-schema`	Schema registry, generation, compilation, and validation helpers
Identity and cryptography	`aip-auth`, `aip-crypto`	Policy primitives and cryptographic primitives remain separate
Discovery	`aip-discovery`	Manifest registry, admission policy, cache, and profile negotiation
Execution	`aip-runtime`, `aip-gateway`	Durable lifecycle services and gateway composition
Transports	`aip-transport` and `aip-transport-*`	Shared transport contract and individual bindings
Compatibility profiles	`aip-profile-*`, `aip-mcp-*`	Wire translation and MCP lifecycle roles
Connector implementation	`aip-connector`	Product-neutral connector traits and error contracts
Fleet data plane	`aip-connector-registry`, `aip-connector-host`, `aip-connector-remote`	Catalog, isolated host, route selection, and remote dispatch
Fleet lifecycle	`aip-connector-admission`, `aip-connector-orchestration`, `aip-connector-control-plane`, `aip-connector-host-bootstrap`	Verified admission, platform-neutral rollout, restricted lifecycle service, and host process shell
PostgreSQL persistence	`aip-storage-postgres`, `aip-connector-registry-postgres`	Runtime state and normalized fleet registry state
Testing	`aip-testkit`, `aip-conformance`	Deterministic fixtures and conformance checks

Direct dependencies still use the same exact source revision. Do not combine different AIP source revisions in one dependency graph unless a documented compatibility procedure explicitly permits it.

7. Keep legacy `full` out of new applications

`full-core` is the broad product-neutral feature group in the facade. A source gate follows every feature it enables and rejects the graph if a product connector feature or dependency becomes reachable.

`full` has a different purpose. The facade source labels it a legacy compatibility feature group. It adds a fixed set of product connector features to `full-core`, so it increases coupling and does not represent the current public connector catalog. New core consumers should use `full-core`, role features, or direct role crates. Add only the product dependency owned by the artifact.

To migrate an application away from `full`:

1. replace `full` with `full-core` or a smaller explicit feature list;
2. run `cargo check` once to resolve the changed graph and identify imports that

depended on product code;

3. add only the required product connector dependency through its documented

integration boundary;

4. inspect the manifest and lockfile diff, then run `cargo check --locked` and

`cargo tree --locked -e features` against the reviewed result.

The repository's registered examples still declare `required-features = ["full"]` at the reviewed revision. Running one of those exact examples inside the source workspace may therefore require the legacy flag:

```
SH
cargo run --locked -p aip --example minimal-agent --features full
```

That requirement belongs to the repository example metadata. It is not a recommended dependency choice for a new application, and the authoring pass for this page did not execute the command.

8. Review and narrow the change

Before merging a dependency change, inspect both the declared and resolved surface:

```
SH
cargo check --locked
cargo tree --locked -e features
git diff -- Cargo.toml Cargo.lock
```

Confirm that:

- the dependency resolves from the approved source and revision;
- `full` is absent unless an exact repository compatibility task requires it;
- every enabled facade feature belongs to the component's declared role;
- product connector packages appear only when explicitly owned by that artifact;
- the change contains no credential, internal endpoint, or local absolute path;
- compilation is reported as compilation evidence, not conformance or runtime

evidence.

If a feature is unnecessary, remove it from `Cargo.toml`, regenerate the lockfile, and repeat the checks. No runtime rollback is required for the semantic example because it performs no external operation. Existing Cargo build artifacts do not need to be deleted to narrow the dependency declaration.

Resolve common failures

Symptom	Likely cause	Bounded action
Cargo cannot find `aip` in a registry	The manifest used registry version syntax for a package that is `publish = false` at this revision	Use the pinned Git source or an approved vendored path
Cargo rejects the active toolchain	The compiler is older than the workspace minimum	Select Rust `1.88` or newer and rerun `cargo check --locked`
An import such as `aip::gateway` is missing	Its facade feature is not enabled	Add only the owning feature and inspect the feature tree
Product connector packages appear unexpectedly	`full` or a direct product dependency is enabled	Replace `full`, then trace the remaining product dependency with `cargo tree`
A repository example refuses to build without `full`	Its registered metadata still requires the compatibility feature group	Use `full` only for that exact example; do not copy it into the application manifest
The dependency graph is much larger than expected	`full-core` or a high-level feature selected more roles than the component needs	Replace it with the smallest task-specific features or direct role crates
Validation succeeds but dispatch later fails	Semantic validation does not authenticate, authorize, admit, route, or execute	Inspect the gateway or endpoint boundary that owns the failed stage

Dependency inspection and semantic validation are read-only with respect to provider state. They do not justify retrying an ambiguous provider mutation.

Related documentation

- Start with Install AIP to choose the

correct source artifact and binary boundary.

- Read How AIP works before embedding

protocol roles in one process.

- Use Capabilities and contracts to interpret the

capability placed in an action.

- Read Profiles, transports, and connectors

before selecting compatibility or connector modules.

- Follow Use native AIP when a client should call a running

endpoint instead of embedding the SDK.