AIP JSON Schemas

Use this reference to select an AIP 1.0 schema, reproduce the committed 52-file catalog, validate data at the correct layer, and record enough provenance to identify the exact bytes you used. It is for protocol implementers and release revi

SECTION	Protocol Standard
TYPE	Specification reference
PROTOCOL	AIP 1.0
SOURCE	docs/spec/schemas.md

Scope and authority

The JSON Schemas are the machine-readable half of the AIP 1.0 normative contract. The prose specification owns behavior, state transitions, security requirements, and conformance. The schemas own JSON shape, required members, enum values, body tagging, closed typed objects, and constraints expressible in JSON Schema.

Property	Reviewed value
Protocol	AIP `1.0`
Schema dialect	JSON Schema Draft 2020-12
Normative files	52
Source directory	`schemas/aip/`
Stable `$id` prefix	`https://getaip.org/schemas/aip/`
Generated registry	52 fixed `SchemaName` entries
Reviewed implementation release	Rust workspace `2.0.0`

A release binds the specification and schema files to an exact source revision. If prose and schema disagree, treat that as a specification defect; do not silently choose a third wire shape. Report which revision, file, and rule your implementation followed.

The committed files are generated artifacts. Change the semantic type or registry owner, regenerate the complete set, review the diff, and commit the result together. Do not hand-edit one exported file as an isolated fix.

Browse and obtain a schema

Choose the source that matches the claim you need to make.

Need	Use
Reproduce the reviewed implementation	`schemas/aip/<file>.schema.json` at the full reviewed commit
Validate with the Rust implementation	The built-in `SchemaRegistry` entry
Inspect every normative filename	Appendix A of AIP 1.0
Prepare a release	Regenerate all 52 files and require a clean source diff
Validate offline	Map each stable `$id` to the matching local committed file

The `$id` is a logical schema identifier. It does not by itself guarantee that an HTTP server exists at that URL. A release or documentation bundle that offers downloadable schemas must publish the committed bytes without changing formatting, identifiers, or content.

Use the full source revision when reproducibility matters. A mutable branch name does not identify a fixed schema set.

Schema families

The registry contains ten families. The filenames below are exhaustive for the reviewed AIP 1.0 set.

Family	Count	Files
Envelope, discovery, and identity	5	`envelope`, `manifest`, `manifest_filter`, `capability_contract`, `identity_context`
Actions and streaming	11	`action`, `ack`, `stream_chunk`, `action_result`, `action_status_request`, `action_status`, `action_result_request`, `action_list_request`, `action_list`, `action_events_request`, `action_events`
Approvals	6	`approval_request`, `approval_decision`, `approval_query_request`, `approval_record_view`, `approval_list_request`, `approval_list`
Sessions	7	`session_request`, `session_view`, `session_list_request`, `session_list`, `session_close_request`, `session_resume_request`, `session_resume`
Transactions	5	`transaction_plan`, `transaction_request`, `transaction_result`, `transaction_query_request`, `transaction_view`
Resources	4	`resource_list_request`, `resource_list`, `resource_read_request`, `resource_read_result`
Callback delivery	5	`callback_delivery_policy`, `callback_delivery_query_request`, `callback_delivery_list_request`, `callback_delivery_record`, `callback_delivery_list`
Delegation	2	`delegation_request`, `delegation_result`
Receipts and audit	3	`receipt_query_request`, `audit_query_request`, `audit_query_result`
Events and channels	4	`escalation`, `event`, `event_stream`, `channel_message`
Total	**52**	Each base name has the suffix `.schema.json`

The envelope schema embeds definitions for all 58 native message variants. That does not turn every message body into a separate top-level schema file. The registry exposes top-level files only for the 52 entries above.

How schemas are generated

The reviewed generator follows one deterministic ownership path:

1. `SchemaName` defines the fixed registry and canonical filename for each

top-level schema.

2. `SchemaRegistry::schema` derives a document from the corresponding semantic

Rust type with `schemars`.

3. `annotate_schema` adds the stable `$id` and fixes the envelope

`aip_version` member to `1.0`.

4. `harden_object_schemas` recursively closes typed objects that do not declare

an explicit `additionalProperties` policy.

5. `SchemaRegistry::all` exports the 52 entries in a fixed order.
6. The CLI serializes each document as pretty JSON into its canonical file.

If schema generation cannot serialize a derived document, the registry emits a fail-closed schema containing `not: {}` and a diagnostic comment. It does not silently publish a permissive replacement.

The generated registry is the code owner; `schemas/aip/` is the committed release artifact. Both must agree at a release revision.

Export from source

Run export only from the source revision you intend to document or release. For the reviewed revision, the direct command is:

```
SH
cargo run -q -p getaip-cli -- schema export schemas/aip
```

The workspace task runner owns the same operation:

```
SH
cargo run -p xtask -- schema
git diff --exit-code -- schemas/aip
```

The exporter creates the destination directory and writes every registered file. It does not make an arbitrary output directory authoritative, and it does not prove that no unrelated file is present. A release check must also verify:

- exactly 52 regular `.schema.json` files;
- the exact registry filenames, with no missing or extra schema;
- valid JSON and Draft 2020-12 on every file;
- 52 unique `$id` values with the canonical prefix;
- byte-for-byte agreement with a fresh export;
- a clean source diff after generation.

The reviewed continuous-integration and release workflows regenerate the catalog and fail when `schemas/aip/` differs from the committed result.

Validate data

Validate a manifest from the CLI

The reviewed CLI has a manifest-specific structural validator:

```
SH
cargo run -q -p getaip-cli -- manifest validate path/to/manifest.json
```

Success prints `manifest valid`. This command validates the JSON value against the manifest schema. It does not admit capabilities, bind handlers, or prove that a deployment can execute them.

The reviewed CLI exposes `schema export`; it does not expose a generic `schema validate` subcommand. Use a Draft 2020-12 validator or the registry API for other top-level schemas.

Validate a named schema in Rust

Select the registry entry explicitly so a filename typo cannot select a different contract.

```rust
RUST
use aip_schema::{SchemaName, SchemaRegistry};
use serde_json::Value;

let value: Value = serde_json::from_slice(&bytes)?;
SchemaRegistry::new().validate_json(SchemaName::Action, &value)?;
```

`validate_json` compiles the generated Draft 2020-12 schema and returns bounded diagnostics when the instance fails.

Validate a native envelope

For a typed native envelope, use the combined registry entry point:

```rust
RUST
use aip_core::Envelope;
use aip_schema::SchemaRegistry;

let envelope: Envelope = serde_json::from_slice(&bytes)?;
SchemaRegistry::new().validate_envelope(&envelope)?;
```

This runs core envelope invariants and then validates the serialized value against `envelope.schema.json`. A gateway still has additional admission work.

Interpret validation failures

Failure class	Meaning	Next check
JSON decode or typed decode	The input is not usable JSON or cannot map to the selected type	Encoding, member names, tags, and value types
Schema compile	The schema is invalid or exceeds compiler safety limits	Dialect, recursion, node count, and text size
Schema validation	The instance violates machine-readable shape	Required members, enums, closed objects, bounds, and body tags
Core validation	A cross-field AIP invariant failed	Type/body match, ids, limits, transaction links, or result requirements
Admission denial	Structurally valid work is not authorized or executable	Identity, replay, manifest, handler, policy, approval, and ownership

Do not retry an unchanged payload merely because a validator returned several diagnostics. Correct the first owning layer, then validate again.

Structural, semantic, and admission validation

Validation is layered. Passing a lower layer never skips a higher one.

Layer	Owner	Examples of what it proves	What it does not prove
JSON and typed decoding	JSON parser and semantic types	The payload can be decoded into the selected representation	Schema, identity, or behavior
JSON Schema	`SchemaRegistry::validate_json`	Required members, enums, tags, object closure, and declared bounds	Cross-record state, authentication, or authorization
Core semantic validation	`SchemaRegistry::validate_envelope`	AIP version, type/body match, message id, body-specific cross-field invariants	Trusted caller, replay ownership, or capability availability
Gateway and runtime admission	Auth, discovery, policy, and runtime services	Authenticated identity, replay claim, admitted handler, input contract, idempotency, approval, and durable ownership	External provider success or qualification
Connector/provider execution	Admitted connector and provider	Provider-specific result under the declared operation contract	Universal interoperability or production readiness

The following cases require semantic or admission logic:

- transaction ids must agree across the wrapper and action context;
- a failed result must carry an error;
- a decision must refer to the stored governed subject;
- a caller must be authorized to read an otherwise valid resource.

Capability `input_schema` and `output_schema` values are dynamic contracts from an admitted manifest. They are validated during action admission and result handling; they are not additional files in the global 52-schema registry.

IDs, versions, and digests

Every committed schema has:

- `$schema: https://json-schema.org/draft/2020-12/schema`;
- one unique `$id` equal to the canonical prefix plus filename;
- a stable filename owned by `SchemaName`;
- `additionalProperties: false` at the top-level typed object.

The envelope schema additionally fixes `aip_version` to `1.0`. Other version axes, such as message-family majors, manifest version, and compatibility profile versions, remain independent as defined by AIP 1.0.

A stable `$id` is not a content digest. To identify exact bytes, record the full source revision and a SHA-256 digest for each file or for a published index whose ordering and encoding are defined. For example:

```
SH
shasum -a 256 schemas/aip/*.schema.json
```

The reviewed registry does not define a canonical aggregate schema-set digest or a documentation asset index. Do not claim one exists. If a publication pipeline adds an index, it must name the source revision and list exactly 52 files. It must also preserve each `$id`, define ordering, and derive digests from the committed bytes rather than regenerating from an unrelated working tree.

An incompatible wire change requires the versioning action defined by the normative specification. Reusing an existing `$id` for incompatible bytes is not a substitute for a protocol or message-family version change.

Closed objects and extension surfaces

The generator recursively adds `additionalProperties: false` to typed objects that expose named properties and do not already declare an extension policy. This makes misspelled members and accidental fields fail structural validation.

Closure is not universal JSON denial. AIP deliberately carries open values in schema-defined places such as `extensions`, trace and security metadata, profile metadata, and selected provider-neutral payload fields. Capability `Binding` also uses flattened profile metadata and therefore remains an explicitly open object. The reviewed envelope and manifest schema graphs each contain that open binding definition.

Use an admitted extension field or versioned profile binding instead of adding an unknown sibling to a closed object. Openness changes only structural shape: extension content is still subject to size, security, redaction, authorization, and profile negotiation rules.

Compiler safety limits

The reviewed Rust validator isolates schema compilation from asynchronous worker stacks and applies the following implementation limits before Draft 2020-12 compilation.

Limit	Reviewed value	Applies to
Maximum nesting depth	128	Schema object and array traversal
Maximum nodes	100,000	Aggregate object, array, and scalar nodes
Maximum text	8 MiB	Aggregate UTF-8 bytes in object keys and string values
Maximum diagnostics	1,024	Returned instance-validation errors
Compiler request queue	64	Pending compilation or validation requests
Dedicated compiler stack	32 MiB	Isolated compiler thread

The diagnostic limit is a cap, not a promise to report every violation. A compiler panic is converted into a compile error, and a stopped or unavailable compiler fails closed.

These values describe the reviewed Rust implementation; they are not payload size limits for every transport and not universal limits for every conforming implementation. Each implementation must bound untrusted schemas and payloads according to its deployment policy.

Formats outside the normative catalog

The following data may use JSON or JSON Schema but is not part of the 52-file AIP 1.0 catalog.

Format	Why it is separate
Connector capability input and output schemas	Versioned with an admitted connector manifest and generated capability catalog
Connector release, admission, evidence, and orchestration records	Control-plane and supply-chain formats, not native AIP messages
Database migrations and storage records	Persistence implementation, not a wire contract
MCP, A2A, and webhook product DTOs	Owned by compatibility profiles at the gateway boundary
Provider request, response, and webhook payloads	Owned by the external product and connector mapping
Documentation page, navigation, and catalog schemas	Authoring and publication controls, not protocol objects

Do not add one of these formats to the normative count because it happens to be machine-readable. Adding a normative top-level schema requires an explicit registry entry, specification ownership, version review, generated artifact, and conformance coverage.

Related references

- AIP 1.0 owns normative behavior, message families, lifecycle,

security, and conformance.

- Compatibility profiles explains where foreign

protocol objects map into the native model.

- Native HTTP API shows how validated envelopes and

operational objects cross the HTTP boundary.

- Conformance and qualification explains which

evidence supports schema, protocol, connector, and deployment claims.