Capabilities and contracts

An AIP capability describes a stable operation that a caller can discover and request. Its schemas describe data shape; its contract describes side effects, execution, retry, data, credential, approval, and recovery expectations. This page

SECTION	Core Concepts
TYPE	Concept
PROTOCOL	AIP 1.0
SOURCE	docs/concepts/capabilities.md

The types and admission behavior on this page apply to AIP 1.0 and the Rust implementation at source revision `97be86e9efedf07ecf1783b03800f683f107fb04`. A capability declaration is not by itself proof of provider behavior, conformance, qualification, or production readiness.

An endpoint is not a capability contract

An API endpoint identifies where and how one system accepts a request. A capability gives an AIP caller a stable semantic operation and enough declared context to decide whether that operation can be attempted safely.

For example, `POST /bookings` does not answer all of these questions:

- Does the operation create state, send a notification, or call another system?
- Which JSON input is valid, and what structured result can be expected?
- Can the caller retry after a lost response?
- Is an idempotency key required, and in which namespace is it unique?
- Can the operation run asynchronously, stream, or be cancelled?
- Which data and credential boundaries apply?
- Does policy require human approval?
- Can the operation be planned, committed, reconciled, or compensated?

A connector can combine several provider requests behind one capability. One provider endpoint can also become several capabilities when distinct use cases need different inputs, risk, authorization, or recovery contracts. The AIP capability remains the caller-facing unit of governance.

The capability mental model

Read a capability as five linked declarations:

1. **Identity:** `id`, `name`, and `kind` say which operation is being offered.
2. **Shape:** `input_schema` and optional `output_schema` define the accepted and

completed data shapes.

3. **Discovery:** description, risk, stability, cost, auth hints, and bindings

help a caller select and project the operation.

4. **Behavior:** `CapabilityContract` describes safety and lifecycle semantics.
5. **Evidence:** manifest admission and implementation claims establish what the

reviewed runtime is willing to publish; separate tests establish what an artifact actually does.

The first four are data. The fifth determines how much confidence a deployment can place in that data.

Capability fields

The `Capability` type has the following top-level fields at the reviewed revision:

Field	Required	Meaning and boundary
`id`	Yes	Operation identifier carried by `Action.capability_id`; external constructors should use the checked parser
`name`	Yes	Human-readable operation name; manifest admission rejects an empty value
`kind`	Yes	`agent`, `tool`, `workflow`, `channel`, `resource`, or `human_task`
`input_schema`	Yes	Draft 2020-12 JSON Schema value for `Action.input`
`output_schema`	No	Draft 2020-12 JSON Schema value checked for a completed result
`description`	No	Human-readable purpose and limits
`risk`	No	`low`, `medium`, `high`, or `critical` operational classification
`stability`	No	`stable`, `experimental`, or `deprecated` lifecycle label
`cost`	No	Profile- or deployment-defined pricing and metering metadata
`auth`	No	Flexible discovery metadata about authentication or scopes
`bindings`	No	Profile-specific projection metadata keyed by `ProfileId`
`requires_human_approval`	No	Top-level approval flag still evaluated by runtime policy
`contract`	No	Structured `CapabilityContract` for execution and governance semantics

The Rust `CapabilityId` parser currently requires only a non-empty string. IDs such as `cap:calendar:booking:create` are a useful ownership convention, not an additional parser-enforced wire rule. Once published, an ID should keep one semantic meaning; changing an operation while reusing its ID makes action, idempotency, policy, and evidence records ambiguous.

A capability whose `kind` is `resource` is discovery-only in the production handler-admission path. Readable objects use the separate `Resource` model and resource query APIs. Other capability kinds are callable and require an implementation claim when strict admission is enabled.

The flexible `auth` and `cost` fields are discovery hints. They do not replace trusted transport authentication, resolved credential handles, capability policy, or the structured credential contract.

Read the contract in safety order

When `contract` is present, `idempotency`, `execution`, and `data` are required objects. Side effects default to an empty list; credentials, approval, service level, transactions, and compensation are optional. Absence of a contract or optional section means that guarantee is not declared. It does not mean the operation is harmless.

Side effects and risk

`side_effects` lists every consequence a caller should assume:

Value	Consequence
`read`	Reads data without an intended mutation
`write`	Creates or changes state
`delete`	Deletes or destroys state
`send_message`	Produces a user-visible or external message
`financial`	Moves money or changes a financial obligation or billing state
`identity`	Changes identity, account, access, or credential state
`medical`	Touches medical or health-related records
`legal`	Touches legal, contractual, or compliance-sensitive records
`external_network`	Calls an external network service
`code_execution`	Executes code or scripts supplied at runtime

Side effects are cumulative. A booking creation can be both `write` and `external_network`; a refund can be `write` and `financial`. The separate `risk` field expresses operational severity. Runtime policy can require human approval when risk exceeds its automatic-approval threshold even if the legacy approval flag is absent.

Idempotency and retry

The idempotency contract has four decisions:

Field	Values	Question answered
`requirement`	`required`, `optional`, `unsupported`	May the runtime accept the action without a key?
`collision_behavior`	`return_original_result`, `reject_conflict`, `revalidate_input_hash`	What happens when the scoped key already exists?
`key_scope`	`action`, `capability`, `principal`, `tenant`, `external_account`	Within which trusted partition is the key unique?
`ttl_ms`	Positive integer or absent	When may the retained key expire?

`ExecutionContract.retry_safety` is a separate classification: `safe`, `safe_with_idempotency_key`, `unsafe`, or `unknown`. The `supports_retry` flag says that retry behavior is implemented; it does not make an unsafe retry safe. A caller needs both the classification and the required idempotency context.

The reviewed runtime hashes the capability ID, business input, and transaction intent when it validates idempotent replay. A runtime-assigned transaction ID is excluded from that fingerprint so the same business intent does not change identity merely because correlation state was allocated.

Execution

`ExecutionContract` declares five support flags:

- `supports_sync`;
- `supports_async`;
- `supports_streaming`;
- `supports_cancel`;
- `supports_retry`.

It also declares `expected_completion` as `sync`, `async`, `streaming`, or `any`, plus the retry-safety classification above. Manifest admission rejects a contract that supports none of the three completion modes. Runtime invocation

defaults an omitted action mode to `sync` and rejects a mode whose flag is false.

Cancellation support means the runtime can reach a handler cancellation path. It does not establish that an external provider can reverse or stop work after commit. Streaming support means the implementation can publish incremental chunks; it does not make every transport binding lossless or resumable without its own replay support.

Data and credentials

`DataContract` requires a sensitivity value: `public`, `internal`, `confidential`, `restricted`, `regulated`, or `unknown`. It also declares whether personally identifiable information may be present and whether redaction is required. Optional residency rules list allowed and prohibited regions. Optional retention rules set minimum retention, maximum time before deletion, and whether legal hold can override deletion.

`CredentialPolicy` declares whether a resolved external credential is required, which issuers are accepted, which downstream scopes are required, and whether OAuth refresh may be used. It carries no raw secret. In the reviewed runtime, the credential handle comes from trusted identity resolution; the runtime checks presence, expiry, issuer, scopes, and tenant consistency before handler execution.

Approval

`ApprovalPolicy` can identify a principal, role, group, tenant policy, external system, or delegated authority. A composed `rule` supports `all`, `any`, and quorum expressions. The policy can also require distinct principals, separation of requester, operator, and approver duties, evidence, expiration, and a stable policy version.

These values identify required authority. A role name or approval object inside an action does not prove membership. The runtime evaluates approval decisions against a trusted authority resolver and the immutable policy snapshot for the governed action.

Service level, transactions, and compensation

`ServiceLevelContract` can declare expected latency, runtime timeout, whether asynchronous execution is expected, maximum queue delay, and a human-readable availability target. These are scheduling and operational expectations, not an availability guarantee.

`TransactionContract` declares supported modes from `execute`, `dry_run`, `plan`, `commit`, `compensate`, `reconcile`, and `rollback_not_supported`. It also says whether commit requires a prior plan and labels dry-run fidelity as `schema_only`, `policy_and_schema`, `downstream_validation`, or `full_simulation`.

`CompensationContract` classifies reversal as `not_required`, `supported`, `best_effort`, or `rollback_not_supported`. Supported compensation identifies the compensating capability; an optional window limits when it can be used, and the contract states whether compensation itself needs approval.

Compensation is a new governed action, not time travel. `best_effort` and `rollback_not_supported` must remain visible to callers deciding how to recover from a partially completed workflow.

Example capability

This illustrative capability is not part of a maintained connector catalog. It shows the minimum structured contract needed to describe a tenant-scoped, retry-sensitive write:

```
JSON
{
  "id": "cap:calendar:booking:create",
  "name": "Create booking",
  "kind": "tool",
  "input_schema": {
    "type": "object",
    "required": ["starts_at"],
    "properties": {
      "starts_at": { "type": "string", "format": "date-time" }
    },
    "additionalProperties": false
```

```
  },
  "output_schema": {
    "type": "object",
    "required": ["booking_id"],
    "properties": {
      "booking_id": { "type": "string" }
    },
    "additionalProperties": false
  },
  "risk": "medium",
  "stability": "stable",
  "contract": {
    "side_effects": ["write", "external_network"],
    "idempotency": {
      "requirement": "required",
      "collision_behavior": "revalidate_input_hash",
      "key_scope": "tenant",
      "ttl_ms": 86400000
    },
    "execution": {
      "supports_sync": true,
      "supports_async": false,
      "supports_streaming": false,
      "supports_cancel": false,
      "supports_retry": true,
      "expected_completion": "sync",
      "retry_safety": "safe_with_idempotency_key"
    },
    "data": {
      "sensitivity": "internal",
      "contains_pii": true,
      "redaction_required": true
    }
  }
}
```

A caller reading this contract can determine that the operation mutates an external system, requires a tenant-scoped idempotency key, supports only synchronous execution, and permits retry only with that key. It cannot conclude that the provider actually deduplicates correctly without implementation and test evidence.

How an action uses the capability

`Action.capability_id` selects the declaration and `Action.input` is validated against `input_schema`. The action may add a mode, idempotency key, trusted identity projection, approval decision, or transaction context. The runtime then applies the declared contract before selecting a handler.

At the reviewed revision, the runtime returns typed, non-retryable errors when required idempotency is absent or a mode is unsupported. It applies the same error class when a plan is required, a transaction or compensation mode is not declared, or the resolved credential does not satisfy the contract.

It validates an `output_schema` only for a result whose status is `completed`; a failure carries its own typed error instead of a successful output object.

Schema validation bounds shape, not meaning. A schema can require an amount to be a number; policy or connector logic must still decide whether that amount is authorized and whether the provider accepted the intended currency and account.

Admission connects declaration to implementation

The reviewed discovery service checks a manifest before publication:

- profile and capability IDs are unique within a manifest, and resource and

channel identifiers satisfy their own non-empty uniqueness checks;

- each input and output schema compiles as Draft 2020-12;

- non-fragment schema references are rejected by default;
- a binding names a profile declared by the manifest and does not duplicate a

profile for the same capability;

- projected profile names do not collide;
- contract invariants such as positive TTLs, non-empty transaction modes,

valid approval rules, and compensation targets hold.

With strict implementation checking, every callable capability must also have a `CapabilityImplementationSupport` record. Admission compares declared cancellation, streaming, retry, transaction, reconciliation, compensation, approval, and credential behavior with the handler's support flags. The local runtime then publishes the admitted manifest and exact handler set atomically.

This check prevents a handler from advertising a feature it does not claim to implement. It still does not prove that the downstream provider behaves as declared. Connector tests, conformance checks, qualification runs, and live external evidence answer different questions and must retain exact artifact identities.

Trust and data boundaries

- The caller can select a capability, but the gateway supplies the authenticated

actor and trusted tenant context.

- The manifest can declare credential requirements, but the deployment resolves

a protected credential handle and the connector owns raw provider secrets.

- The contract can label side effects and retry safety, but the provider remains

the source of truth for external commits.

- The runtime can validate schemas and declared support, but it cannot infer an

undeclared business invariant from JSON shape.

- A compatibility binding can rename or project a capability, but the native ID

and contract remain the canonical operation identity.

Design trade-offs

A small, coarse capability catalog is easier to discover but forces unrelated risk and policy into one contract. A highly granular catalog gives policy and retry logic a more accurate unit, but it increases versioning, navigation, and qualification work. The right boundary is one stable business intent with one coherent input, side-effect, authorization, and recovery model.

Rich contracts make conservative automation possible. They also create an obligation to track provider changes. When an upstream endpoint changes its side effects, idempotency, or result shape, the connector must update and requalify the capability rather than preserving a stale declaration for compatibility.

What a capability does not mean

- `stable` does not mean a deployment is qualified or a provider is available.
- `read` does not mean the result is non-sensitive or safe to disclose.
- `supports_retry` does not mean retry is safe without checking

`retry_safety` and idempotency.

- `supports_cancel` does not guarantee reversal after a provider commit.
- `full_simulation` is a declared dry-run fidelity, not proof of provider parity.
- An absent contract does not imply zero side effects or unrestricted use.
- A profile binding does not replace the native capability ID or contract.

Related pages

- Actions and sessions
- Approvals and policy
- Transactions and compensation
- Profiles, transports, and connectors
- JSON Schemas