Delegation

Use this page to understand how one AIP participant asks another participant to execute a child action. It is for application, runtime, and gateway developers who need to preserve a parent-child graph, choose local or remote execution, and

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

Delegation is a first-class AIP graph edge. The request names a parent action, contains a complete child action, and identifies both the requester and the delegate. The child still passes through the ordinary capability, policy, approval, transaction, idempotency, and result lifecycle. A delegation does not transfer unlimited authority, make remote delivery exactly once, or turn a human-readable scope into an authenticated grant.

This page describes the reviewed Rust implementation of AIP 1.0 at source revision `97be86e9efedf07ecf1783b03800f683f107fb04`. Durability and remote reach depend on the stores, routes, peer keys, and transports configured by a deployment.

Start with the graph edge

Keep the parent action, child action, and delegation edge separate. Each has a stable identity and a different job.

Identity	What it names	What it does not prove
Parent action ID	The action named as the parent of this edge	That the record exists, is owned by the requester, or can cancel the child
Child action ID	The independently executed action	That the action was accepted by the intended delegate
Delegation ID	The durable edge between parent and child	Exactly-once delivery or successful execution
Requester principal	The participant creating the edge	Authority merely because the value appears in the payload
Delegate principal	The participant expected to execute the child	Possession of a peer key or permission for the capability
Correlation ID	The surrounding request relationship	Ownership of either action

The runtime persists a `DelegationRecord` containing the effective request, current delegation status, optional result, and creation and update times. It can also index records by parent action and child action. These indexes make the graph observable inside the runtime; they do not create a public delegation query message in the reviewed protocol surface.

```
MERMAID
flowchart LR
    P["Parent action"] --> Q["DelegationRequest"]
    Q --> G["Durable graph edge"]
    G --> D{"Registered peer route?"}
    D -->|"No"| L["Execute child locally as delegate"]
    D -->|"Yes"| O["Durable remote outbox"]
    O --> R["Authenticated AIP peer"]
    L --> A["Child ActionResult"]
    R --> B["Correlated DelegationResult"]
    A --> S["Settle delegation record"]
    B --> S
```

```
    S --> E["Event, receipt chain, and optional callback"]
```

The text equivalent has five steps. Admit one request, bind it to a durable graph edge, choose local execution or a registered peer route, and execute the child. Settle the same delegation ID with a correlated result.

The wire contract carries a complete child action

A `DelegationRequest` contains:

Field	Role in the graph
`delegation_id`	Stable identity for this parent-child edge
`parent_action_id`	Action named as the parent of the edge
`child_action`	Complete action to execute, including its own ID and capability
`requested_by`	Principal creating the delegation
`delegate`	Principal expected to execute the child
`scope`	Non-empty description retained on the request and delegation hop
`callback`	Optional destination for a final delegation result
`metadata`	Optional routing, scheduling, or product-specific data

The core envelope validator rejects an empty `scope`, equal parent and child action IDs, or an invalid child action. The child action's existing `delegation_chain` may contain at most ten entries when that action passes core envelope validation.

The reviewed generic runtime does not look up the named parent action or prove that `requested_by` owns it while admitting this request. An ingress or workflow that requires an existing, requester-owned parent must enforce that relationship before calling the delegation path.

A `DelegationResult` repeats the delegation, parent, and child IDs. It carries one of six statuses:

Status	Meaning in the model	Reviewed runtime behavior
`accepted`	The delegation was admitted but has not started	Available on the wire; a new local record is stored as `running`
`running`	Child work is still in progress	Returned immediately for a locally scheduled asynchronous child and allowed from a remote peer
`completed`	Child work completed	Terminal
`failed`	Scheduling or child execution failed	Terminal
`cancelled`	The child reached a cancelled result	Terminal; it does not imply parent-driven cascade
`requires_human`	The child needs approval or human input	Terminal for the delegation lifecycle at this revision

A terminal delegation result must include at least one of `ActionResult` or `ProtocolError`; it may contain both. When local execution reaches the child action lifecycle, the runtime maps the child's result status to the delegation status and includes the child result.

A failure that prevents a retryable remote dispatch from ever settling remains an outbox concern until delivery succeeds, becomes a permanent failure result, or reaches dead letter.

Path validation protects graph shape

For a new edge, the runtime appends a `DelegationEntry` to the child action unless the exact current hop is already the last entry. Each entry records `from`, `to`, `scope`, and `delegated_at`.

Before accepting the path, the reviewed runtime checks that:

- the requester does not delegate to itself;
- every earlier scope is non-empty;
- no hop delegates a principal to itself;
- each hop starts where the previous hop ended;
- the chain ends at the current requester;
- a principal does not appear twice in a cycle; and
- the new target is absent from the earlier path.

The runtime accepts an exact replay of a stored request. It also normalizes the two fields that it may add itself—the current hop and a copied callback—before deciding whether a retry is the same request.

Reusing a delegation ID for a different parent, child, requester, delegate, scope, callback, or metadata is an authorization conflict. A previously stored terminal result is returned for a matching replay.

The numeric hop limit and the path checks are related but distinct. The core wire validator limits the incoming child action's existing chain to ten entries.

The direct runtime path validator has no separate numeric limit, and the reviewed code does not repeat the length check after appending a missing current hop. Embedders that call the runtime without validated AIP envelopes need to enforce their intended bound at that boundary.

Scope records intent; policy grants authority

The `scope` field is a non-empty string. The runtime copies it into the delegation chain, event data, and `DelegationMade` receipt. It compares the string when recognizing an already-recorded current hop.

The reviewed generic delegation path does not parse that string, compare it with the requester's authenticated grants, prove that each hop narrows authority, or add it to the delegate's authenticated scopes. Treat it as an auditable statement of delegated purpose, not as the authorization decision.

The child action still enters ordinary action processing. The selected capability contract, resolved identity, policy decision, required approval, transaction mode, credential context, and connector handler govern what can actually execute. Deployments that require scope narrowing need an explicit policy vocabulary and enforcement point in addition to the delegation string.

`FederationContext` similarly carries a current `trust_domain` and a list of domain `hops` on an action. The reviewed generic delegation validator does not update or evaluate those fields. A label can preserve routing context for deployment policy, but it does not establish transitive trust or cross-domain authorization by itself.

Local delegation changes execution identity deliberately

When no registered router owns the request, the runtime executes the child locally. Before doing so, it requires transport-established authentication for the requester and matches both the authenticated principal ID and kind to `requested_by`. It also requires the runtime-verified current hop to end at the declared delegate.

The runtime then constructs a child execution context with:

- the delegate as the current actor and service account;
- the requester as `acted_on_behalf_of`;
- the existing resolved identity context otherwise preserved;
- an internal authenticated-principal projection for the delegate;
- an issuer derived from the requester's authenticated issuer;

- the requester's expiry and credential fingerprint; and
- an empty authenticated scope set.

This projection gives ordinary action processing a precise principal and on-behalf-of relationship. It is not evidence that the delegate presented a new external credential at this local boundary, and it does not copy the free-form delegation scope into authenticated grants.

Synchronous and streaming child actions execute on the request path. A local asynchronous child returns a `running` delegation result, records a receipt chain, and continues in a spawned task. A configured persistent backend can retain the running record for recovery; the default in-memory stores do not turn process memory into durable storage.

Remote delegation uses an explicit peer route

A gateway delegation route can select by delegate principal ID, child capability ID, or both. A route with both selectors matches only when both match; a route with neither selector is a catch-all. The gateway uses the first matching explicit route before consulting extension routers, so overlapping routes require deliberate registration order.

The selected binding sends the first-class `DelegationRequest` over native HTTP or native NATS request/reply. If no peer route or extension router owns the request, the gateway accepts local execution only when the delegate matches its own manifest agent; otherwise it returns `delegation.target_mismatch`.

Native peer security is configured per route. It contains a non-empty trust domain, a local request signer, the expected peer principal, an exact expected peer `did:key`, an optional opaque credential handle, a transport retry budget, and an endpoint policy. The request is signed and carries the configured trust domain. The peer response is accepted only after the gateway verifies:

- the exact expected DID and envelope signature;
- the expected peer principal as sender;
- the local request signer as recipient;
- the request correlation ID;
- an `in_response_to` reference to the exact request message; and
- a response timestamp within five minutes of the verifier's current time.

The body must be a `DelegationResult` whose delegation, parent, and child IDs match the request, or a protocol error. Native HTTP also applies the configured URL, DNS, redirect, TLS, timeout, and response-size policy. These checks authenticate one configured peer exchange. They do not make every principal in an earlier delegation chain trusted by the current gateway.

The remote outbox makes retries explicit

The runtime asks `can_route` before creating a remote outbox item, so route ownership must be deterministic for a stable route table. A new outbox record retains the request and trusted message context, but not secret credential material; a `CredentialHandle` is only an opaque reference.

The dispatch states are `pending`, `leased`, `delivered`, and `dead_lettered`. One worker acquires a time-bounded lease with a fencing token and renews it during the peer call. A successful correlated result is stored with the delivered record and returned on replay. Losing the lease blocks stale settlement.

The reviewed runtime creates each outbox item with five total lease attempts. Retry scheduling uses exponential backoff starting at 250 milliseconds and capped at 30 seconds.

A route that declines a request it claimed to own is treated as a retryable dispatch error. An error explicitly marked non-retryable becomes a terminal failed delegation result; other dispatch errors return the record to pending until the attempt budget is exhausted.

Native HTTP has its own route-level retry budget, two retries after the initial attempt by default. Those transport attempts can occur within one durable outbox lease.

The gateway sends the delegation ID as the HTTP idempotency key, but network failure can still leave the sender unable to know whether the peer received a request. Remote peers therefore need idempotent handling of the stable delegation ID and, for side-effecting child work, the child action's own idempotency contract.

Recovery scans delegation records in `accepted` or `running` state. It reuses the remote outbox when a router still owns the request and otherwise re-enters local child execution. Recovery is a replay mechanism, not exactly-once proof. Persistent stores, stable routes, child idempotency, and operator handling of dead letters remain deployment responsibilities.

Results, streams, receipts, callbacks, and cancellation

Remote result ingestion binds an update to the stored graph and to the transport-established delegate principal. If an embedded `ActionResult` is present, its action ID must match the child ID. Replaying the same terminal result is idempotent; a different result cannot replace a terminal record.

Remote stream chunks are accepted only for a child already present in the delegation graph, from the expected delegate, and before terminal delegation settlement. The runtime annotates stream events with the delegation and parent IDs. A stream chunk remains progress evidence; the final delegation result is the settlement authority.

The reviewed runtime attaches a receipt chain if the result does not already carry one. Its locally constructed chain contains a `DelegationMade` receipt with the graph IDs, delegate, scope, actor, time, and correlation. That receipt records creation of the edge; it is not by itself proof of the external effect performed by the child.

An optional request callback can receive a terminal `DelegationResult` on the runtime paths that own terminal callback dispatch. For a new request, the runtime also copies that callback to the child action when the child has none.

Child lifecycle paths can then use it for outputs such as stream chunks or a queued approval continuation. Callback delivery has its own durable retry and security policy; consumers must select behavior by message type and stable IDs rather than assuming that every callback is the final delegation result.

`cancelled` is a valid terminal delegation status, normally derived from a cancelled child result or accepted from the authenticated remote delegate. The reviewed runtime does not walk from a cancelled parent through its delegation children and issue cancellation automatically. Workflow code that needs cascade cancellation must address active child action IDs explicitly, observe their terminal results, and handle remote or provider uncertainty separately.

Peer delegation is not connector-fleet routing

Both paths may cross a process boundary, but they represent different work.

Question	Peer delegation	Connector-fleet routing
Protocol unit	`DelegationRequest` containing a child action	Ordinary `Action` execution
Semantic result	New durable parent-child graph edge and `DelegationResult`	`ActionResult` for the same action
Target selection	Delegate principal and/or child capability in an explicit peer route	Registered connector instance and replica for the action's tenant and capability
Trust binding	Expected peer principal, exact peer DID, signed request and correlated response	Registry state, instance and replica identity, active lease, route assignment, and fencing evidence
Retry record	Delegation outbox keyed by delegation ID	Action lifecycle plus an action-scoped connector route assignment

Question	Peer delegation	Connector-fleet routing
Primary purpose	Ask another AIP participant to own a child action	Invoke an external product through a connector host

Registering a connector instance does not automatically create a delegation route. An implementation can explicitly provide both roles, but the runtime contracts remain separate. Do not invent a delegation edge merely because a central runtime dispatched an action to a remote connector replica.

Trust and data boundaries

- Payload requester, delegate, scope, chain, federation labels, metadata, and

callback values are claims until the appropriate ingress and policy establish their authority.

- Local child execution requires a transport-established requester and a

runtime-verified terminal hop. Remote execution additionally requires the configured peer key, principal, response correlation, and graph IDs.

- A delegation scope records intent but does not grant a capability, tenant,

object, provider account, credential, or approval.

- A route trust domain is administrative context, not transitive trust in every

earlier or later domain.

- The complete child action and optional metadata cross the peer boundary.

Deployments must minimize or redact sensitive context before routing it.

- Opaque credential handles may be retained for recovery; secret material must

remain behind the credential provider and transport boundary.

- A receipt chain can prove the data actually included in its receipts. It does

not prove an unstated provider outcome.

- Persistent graph and outbox state can contain identity and business context;

retention, tenant isolation, read authorization, and redaction remain deployment responsibilities.

Design choices and trade-offs

Embedding a complete child action makes delegation transport-independent and lets the child use the normal AIP lifecycle. It also means the delegating side must choose a new action identity, minimize the payload, and preserve both action and delegation idempotency.

Keeping local and remote execution behind one `DelegationRouter` decision gives applications one semantic graph model. Deterministic route ownership becomes a hard requirement: a route-table change during recovery can move a running edge between remote and local paths unless deployment policy prevents it.

Separating the free-form scope from authenticated scopes avoids pretending that one string is a universal authorization language. The cost is that deployments requiring formal attenuation must define and enforce it explicitly.

A leased durable outbox prevents concurrent workers from settling the same dispatch and preserves a terminal replay. It cannot eliminate duplicate peer receipt across network uncertainty, so stable IDs and idempotent child behavior remain necessary.

Treating `requires_human` as terminal gives the parent a clear result at the current delegation edge. The reviewed delegation record does not remain open waiting for a later approval; workflow code must decide explicitly how to

continue.

What delegation does not guarantee

- A delegation ID does not make remote delivery or a provider side effect exactly once.
- A named parent action ID does not prove that the parent exists or belongs to the requester.
- A non-empty `scope` does not prove authority, attenuation, or authenticated scopes.
- A contiguous delegation chain does not authenticate every historical hop.
- A federation trust-domain label does not create transitive trust.
- The ten-entry wire validation limit is not a second limit inside the direct runtime path after it appends a hop.
- A `running` result does not prove that a remote peer or child handler has started work.
- A `DelegationMade` receipt does not prove child completion or an external provider effect.
- Cancelling a parent does not automatically cancel its delegated children.
- Registering a connector instance does not register an AIP peer-delegation route.
- The reviewed message set does not expose a public delegation query or list request.
- This page does not claim live peer connectivity, connector qualification, or product coverage beyond the separately documented Cal.diy, Hermes Agent, Chatwoot, Dify, CrewAI, and Twenty connectors.

Related pages

- Actions and sessions
- Capabilities and contracts
- Identity and trust
- Transactions and compensation
- Profiles and connectors