Approvals and policy

Use this page to understand how an AIP capability can pause an action for approval and later resume the same queued action from durable, verified authorization. It is for application, runtime, and connector developers who need to preserve a

SECTION	Core Concepts
TYPE	Concept
PROTOCOL	AIP 1.0
SOURCE	docs/concepts/approvals-and-policy.md

An approval is not a comment attached to an action. The runtime freezes an approval request, admits decisions only from transport-authenticated actors, resolves their authority from deployment-owned data, and computes a terminal state from stored policy and verified votes. An approved action still has to execute and may still fail at the provider.

This page describes the reviewed Rust implementation of AIP 1.0 at source revision `97be86e9efedf07ecf1783b03800f683f107fb04`. It explains the implemented lifecycle and trust boundaries, not a claim that a particular deployment has configured an external policy system or qualified every approval path.

Keep policy, request, decision, and authorization separate

Four objects participate in an approval workflow:

Object	Created by	What it establishes
`ApprovalPolicy`	Capability owner	The required authority, evidence, time limit, vote rule, and separation-of-duties settings
`ApprovalRequest`	Runtime	The immutable action, identity, risk, policy snapshot, and hash being governed
`ApprovalDecision`	Approver-facing client or system	A claimed outcome, reason, constraints, evidence, stable decision ID, and policy hash
`ApprovalRecord`	Approval backend	The request, every resolver-verified decision, computed status, terminal decision, and timestamps

The decision object is not authorization by itself. Before storing it, the runtime binds its `approver` to a transport-established identity and attaches an `AuthorityMembership` returned by a trusted resolver. The resulting `VerifiedApprovalDecision` also carries a hash over the canonical decision, the authority membership, and the resolver revision.

The complete approved `ApprovalRecord` is the durable authorization. Runtime execution uses it to construct a non-serializable `VerifiedApprovalSet` for a connector attempt. A remote dispatcher transports the record only as signed gateway-to-host route metadata.

```
MERMAID
flowchart LR
    P["Capability approval policy"] --> R["Frozen approval request"]
    R --> D["Authenticated decision"]
    D --> A["Trusted authority resolution"]
    A --> V["Verified decision journal"]
    V -->|"rule satisfied"| X["Approved authorization"]
    V -->|"deny, expire, or valid revoke"| T["Terminal rejection"]
    X --> E["Resume queued action"]
    X --> H["Signed remote-host import"]
```

A capability declares the approval policy

An `ApprovalPolicy` first states whether approval is required. When it is, the policy can provide a reason, a request TTL, required evidence, a stable policy version, a legacy approver selector, an optional composable rule, a minimum number of distinct principals, and separation-of-duties controls.

The legacy `approver_selector` selects one of these authority classes:

Selector	Trusted membership used for the match
`principal`	Exact principal ID
`role`	Role name
`group`	Group name
`tenant_policy`	A tenant policy membership, or tenant membership under the legacy selector
`external_system`	External system identifier

The optional `rule` provides a more precise expression:

Rule	Evaluation behavior
`all`	Every child must match; by default, one decision or one principal cannot satisfy multiple children
`any`	At least one child must match
`quorum`	At least `required` matching decisions must exist; distinct principals are required by default
`principal`, `role`, `group`	Matches the corresponding trusted authority membership
`tenant_policy`	Matches one exact tenant policy ID
`external_system`	Matches one exact external system ID
`delegated_authority`	Matches a live, non-revoked grant with the requested scope and applicable risk or value limits

`all.allow_decision_reuse` can explicitly permit reuse across child rules. Independent of the expression, `minimum_distinct_principals` is clamped to at least one and must be met before the record can become approved.

Rule leaves never trust roles, groups, policy names, external systems, or delegated grants carried only in a decision or action payload. Those values come from the authority resolver.

The runtime freezes what is being approved

When authorization requires human approval and the action has no approved decision, the runtime creates a pending request and queues the action as `requires_human`. The request records:

Frozen value	Purpose
Approval, action, and capability IDs	Bind the workflow to one invocation and one capability
Requester, subject, and operator	Support ownership, audit, and separation-of-duties checks
Resolved identity context	Retain the tenant and represented identity used for the decision
Capability risk and governed numeric value	Bound delegated authority when those values are available
Policy snapshot and version	Preserve the rule that terminal evaluation will reproduce
Expiration time	Close a pending request after its policy TTL
Policy hash	Bind the policy and governed subject to later decisions

The policy hash is SHA-256 over a canonical object containing the full policy, action ID, capability ID, requester ID, represented subject ID, and an input hash. The input hash covers capability ID, action input, and transaction

context. A decision submitted for a request with a policy hash has to carry the same hash.

The request can contain evidence references, but it does not copy the raw action input. If the policy requires an input snapshot, the runtime stores a redacted `input_snapshot` artifact with the canonical input hash. If it requires a policy decision, the runtime stores a hash of the action ID, capability ID, and approval policy.

The governed value is a bounded implementation convention, not a general expression evaluator. The reviewed runtime looks for the first unsigned integer at `/amount_minor`, `/amount`, `/value`, or `/total_minor`.

A decision becomes a verified vote

Decision admission follows a fail-closed sequence:

1. The message context has to contain a valid transport-established actor.
2. The claimed approver ID and kind have to match that actor; the runtime then

replaces the payload principal with the authenticated principal.

3. The runtime loads the durable approval request and resolves authority for

the approval ID, actor, and request tenant.

4. The returned membership has to belong to the actor, remain unexpired and

non-revoked, and match the approval tenant when the request is tenant-bound.

5. The decision has to identify the same approval, carry a non-empty stable

`decision_id`, and match the request policy hash. System-generated expiry bypasses the ordinary client admission path but receives its own stable decision ID and trusted runtime authority.

6. Authority, separation of duties, evidence, revocation target, and supported

input constraints are checked before the vote is stored.

7. The backend appends the verified decision and evaluates the full record

atomically.

When the registered capability is available during decision admission, its current approval policy supplies the authority, evidence, and separation-of-duties checks. Terminal rule evaluation uses the request's stored `policy_snapshot`. This makes the snapshot reproducible while also making capability-policy changes during an open request an operational concern.

The runtime derives `authority_path` from trusted membership: principal, tenant, roles, groups, tenant policies, external systems, and the authority revision. It does not retain an authority path asserted by the client.

Separation of duties

The policy can prohibit the requester or operator from approving. By default, self-approval is not allowed when the requester and approver are the same; `allow_self_approval` has to permit it and no stricter requester rule may forbid it. `operator_must_differ` independently protects actions operated by a principal other than the represented subject.

Required evidence

The implemented evidence requirements are:

Requirement	Satisfied by
`reason`	A non-empty decision reason
`input_snapshot`	An artifact of kind `input_snapshot` on the request or decision

Requirement	Satisfied by
`policy_decision`	A stored policy decision ID or an artifact of kind `policy_decision`
`external_ticket`	An artifact of kind `external_ticket`
`attachment`	An artifact of kind `attachment`

An evidence artifact can contain an ID, kind, URI, integrity hash, and redaction flag. The generic runtime verifies the required kind or reason. Retrieval, malware scanning, ticket validity, and external-system semantics belong to the deployment integration.

Decision constraints

An approved decision can constrain fields in the queued action input. A field is interpreted as a JSON Pointer; a name without a leading slash is converted to a top-level pointer. The reviewed runtime implements these operators:

Operator	Supported comparison
`eq`, `neq`	JSON value equality or inequality
`lt`, `lte`, `gt`, `gte`	Numeric comparison through JSON numbers
`contains`	String contains string
`max_length`, `min_length`	Unicode character count against an unsigned integer
`in`	Actual JSON value occurs in the supplied array

A missing field, incompatible value type, failed comparison, or unknown operator rejects the decision. The check runs against the durable queued action before an approved vote is admitted. Connectors receive the same resumed action; the generic constraint layer is not an arbitrary policy language and does not define provider-specific conditions.

The record computes one terminal state

An approval starts as `pending`. Approved votes accumulate until both the global distinct-principal minimum and the selected rule are satisfied. A vote that can contribute to one leaf may still leave a quorum or `all` expression pending.

The evaluator gives terminal rejection outcomes precedence over approvals:

1. any admitted `revoked` decision yields `revoked`;
2. otherwise any `expired` decision yields `expired`;
3. otherwise any `denied` decision yields `denied`;
4. otherwise the approval rule determines `approved` or leaves the record

`pending`.

The backend changes status and selects the terminal decision in the same locked record update. Once terminal, it rejects a new decision. Reusing a stable decision ID with different verified content is also rejected.

This terminal rule narrows the meaning of revocation in the reviewed implementation. A `revoked` decision has to name an already admitted approved decision, but it can be admitted only while the overall record remains pending, such as during a multi-vote workflow. It does not revoke an already terminal approved record or undo provider work that has executed.

TTL expiry is a runtime-owned lifecycle transition. The expiry worker selects pending records past `expires_at`, records a system decision with trusted runtime authority, and publishes the same durable terminal transition used by other outcomes.

Terminal publication is recoverable

A terminal record is not considered fully published merely because its status changed. The runtime claims a fenced approval-transition lease, writes the canonical receipt and event, then either resumes or terminates the queued action. It marks the transition complete only while it still owns the fencing token.

If a publisher stops between settlement and publication, a recovery worker can claim the incomplete transition and repeat the publication path. Stable event and receipt IDs make their durable writes idempotent.

Terminal status	Queued-action effect
`approved`	Atomically lease the `requires_human` action, attach the terminal decision, and re-enter normal action processing
`denied`	Persist and settle a failed action result with `policy.approval_denied`
`expired`	Persist a failed result with `policy.approval_expired` and expire the queue record
`revoked`	Persist and settle a failed result with `policy.approval_revoked`

Resuming is authorization to attempt the original action, not evidence that the attempt completed. Normal schema, capability, transaction, idempotency, credential, timeout, connector, and provider failure paths still apply.

Execution rechecks durable authorization

An action carrying `Action.approval` passes the runtime gate only when the decision is `approved` and exactly matches a durable record whose status is `approved`. The record request must also match the current action ID and capability ID.

The runtime then projects three compact indexes into `VerifiedApprovalSet`: approval IDs, available decision IDs, and policy hashes. It also includes the complete durable authorization. This execution context is trusted, process-local data; it is not part of the public AIP schema and is not accepted from an untrusted action payload.

Remote connector hosts import the verified record

A remote connector host may not share the control-plane approval database. The remote dispatcher therefore serializes the complete approved authorization under `approval_authorization` in the pinned connector route. That route travels inside the gateway-signed AIP envelope.

Before the host invokes its local gateway, it:

1. verifies that the envelope signer and sender match its configured central

gateway and that the recipient and route match the admitted host;

2. requires approval authorization for an action carrying an approval and

rejects unused authorization on an action without one;

3. decodes the record and matches its terminal decision, approval ID, action

ID, capability ID, tenant, and policy hash;

4. reconstructs the record from its resolver-verified decisions and requires

the stored policy to reproduce the approved terminal state;

5. imports the exact record atomically, accepting only an identical existing

record and rejecting conflicting durable state;

6. invokes the local gateway only after the import succeeds.

The record-validation method deliberately does not authenticate its containing transport. The connector host's configured gateway signature and pinned route provide that boundary. A storage failure returns a retryable unavailable error; a conflicting authorization returns a non-retryable authorization error.

This design lets an isolated host verify policy state without direct access to the central approval store. Its trade-off is that the signed envelope carries a larger authorization object and the host explicitly trusts the configured gateway for the routed action bytes.

Read approval state without exposing input by default

The operational read model can return the original request, terminal decision, status, timestamps, and optionally linked action status or the approval receipt chain. Approval reads pass through transport identity, ownership, tenant, scope, delegation, and sensitive-field authorization.

Raw governed input remains in the durable queued action. A caller has to request `include_evidence_payload` explicitly. The export path requires approval visibility plus `approval:export` and `approval:sensitive`; it then derives a payload containing action ID, capability ID, input hash, exact input, transaction context, and resolved identity. Ordinary approval views, receipts, events, and callbacks do not receive that payload.

Trust and data boundaries

Boundary	Trusted input	Rejected assumption
Client to runtime	Authenticated transport actor	Payload `approver`, role, group, tenant, or authority path proves authority
Runtime to authority resolver	Approval ID, authenticated actor, request tenant	Any resolver result is valid without actor, tenant, expiry, or revocation checks
Request to decision	Durable IDs, policy hash, evidence, queued input	A comment or stale approval authorizes changed work
Approval backend	Atomic record update and fenced publication state	Multiple votes can race safely without storage coordination
Gateway to remote host	Configured gateway signature and pinned connector route	A self-consistent approval record authenticates its own transport
Runtime to connector	Non-serializable verified execution context	A raw `Action.approval` is equivalent to verified authorization
Read model to operator	Scoped approval view and explicit sensitive export	Approval visibility automatically grants raw input access

Approval receipts and evidence hashes make later inspection possible, but they do not replace secure storage, key management, resolver governance, or external audit retention. The chosen approval backend also determines whether records and transition claims survive process loss; an in-memory backend does not provide process durability.

Design choices and trade-offs

- Immutable requests and hashes prevent a decision from silently authorizing a

different action, at the cost of creating a new workflow when governed intent changes.

- Resolver-owned authority avoids trusting client-supplied roles and grants, at

the cost of operating a current, fail-closed membership source.

- Composite rules and atomic vote evaluation support independent review, at

the cost of more durable state and careful decision identity management.

- A small constraint vocabulary is deterministic and auditable, but it cannot

express every provider policy.

- Fenced terminal publication supports crash recovery without treating a

status write as completed side effects, at the cost of a separate recovery path.

- Signed full-record import keeps remote hosts isolated from the control-plane

database, at the cost of trusting one configured gateway boundary.

What approvals do not establish

The reviewed implementation does not establish any of the following:

- that an `external_system` membership contacts or validates an external policy

product;

- that arbitrary constraint operators or provider-specific policy expressions

are supported;

- that an approved record proves connector invocation or provider completion;
- that approval replaces transaction planning, idempotency, credential,

tenancy, or capability authorization checks;

- that a terminal approval can later be revoked through the same decision

journal;

- that evidence references have been retrieved or independently validated;
- that every deployment uses a process-durable approval backend;
- that approval behavior has been live-qualified for every connector.

Approval also differs from escalation. Approval decides whether one frozen operation may proceed. Escalation asks another actor to supply input, take over, or resolve a broader exception; it has its own lifecycle and resolution model.

Related documentation

- Identity and trust explains how transport identity,

tenant membership, and deployment-owned enrichment become trusted context.

- Actions and sessions explains the queued action,

idempotency, status, and recovery surfaces that surround approval.

- Transactions and compensation owns plan,

commit, reconciliation, and compensation semantics.

- Observe and recover covers operational

status, events, receipts, and recovery workflows.

- HTTP API owns concrete approval query and decision

transport syntax.