Actions and sessions

Use this page to understand how an AIP client identifies, observes, cancels, and resumes work. It is for application and connector developers who need to track an invocation beyond one request or transport connection.

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

An action is one requested capability invocation. A session is an optional relationship between two principals that can group related messages and work. They have different identities, lifecycles, and recovery rules.

This distinction matters when a request is queued, a stream reconnects, a worker restarts, or a provider responds after the original transport has gone away. The transport connection is not the identity of the work.

State-backed does not always mean process-durable: the reviewed runtime has pluggable stores, including in-memory implementations. Deployment configuration determines which state survives a process or host loss. The types and runtime behavior on this page apply to AIP 1.0 at source revision `97be86e9efedf07ecf1783b03800f683f107fb04`.

Keep four identifiers separate

Identifier	Scope	Use
`act_...`	One capability invocation	Submit, cancel, and query one unit of work
`sess_...`	A relationship between an initiator and responder	Group messages and resume session context
`msg_...`	One protocol envelope	Detect or correlate message delivery and replay
`corr_...`	A related request or trace flow	Associate messages without merging their identities

These are the forms produced by the typed constructors and required by their checked parsers. The identifier types use transparent serialization, so a deserialization boundary must invoke a checked parser or an equivalent prefix check; deserialization alone does not enforce the form.

An action can exist without a session. When a session is used, its ID is carried by the enclosing `Envelope.session_id`; it is not a field of `Action`. The runtime copies that trusted message context into its queued action record and exposes it later in `ActionStatus`.

Do not substitute one identifier for another. Reusing an action ID for a different capability, input, or principal is an identity conflict. Retrying the same business intent is governed by the capability's idempotency contract and key scope, not by inventing a new meaning for an existing action ID.

What an action carries

An `Action` always contains an `id`, `capability_id`, and structured JSON `input`. Every other field is optional or defaults to an empty collection:

Field	Purpose	Important boundary
`mode`	Selects `sync`, `async`, or `streaming`	Omission is treated as `sync`; the capability contract must support the selected mode
`idempotency_key`	Identifies repeat business intent	Its required presence, collision behavior, scope, and retention come from the capability contract
`timeout_ms`	Supplies a caller timeout	The runtime combines it with the capability service-level contract and deployment behavior
`conversation`	Carries structured conversation context	It is not the AIP session lifecycle
`memory_context`	Carries application state or context	Runtime transport metadata is kept outside the caller-owned durable action contract
`delegation_chain`	Records delegated authorization hops	The path is checked for continuity and cycles; authority still comes from trusted context
`federation`	Describes cross-domain routing context	It does not authenticate a remote domain by itself
`callback`	Names an asynchronous delivery target and profile	Delivery state is tracked separately from action completion
`observability`	Carries trace, span, and diagnostic fields	It is correlation metadata, not authorization
`compliance`	Carries requested regimes and classification	Runtime policy remains authoritative
`identity`	Projects tenant, account, user, and credential references	The runtime replaces caller claims with resolved identity at the trusted ingress path
`approval`	References a decision authorizing this invocation	The decision must match the durable approval record and policy snapshot
`transaction`	Selects execute, plan, commit, reconcile, or recovery intent	Transaction admission and result state are separate from the action lifecycle

After trusted identity and transaction normalization, the runtime retains the durable action as the caller-facing work contract. Connector-facing enrichment, queue leases, retry metadata, approval records, transaction records, receipts, and stream chunks live in separate runtime state.

One action has several status surfaces

AIP deliberately separates acceptance, execution observation, and result data. Reading one surface as if it were another creates false completion claims.

Acknowledgement

`Ack.status` has five wire values:

Value	Meaning
`accepted`	Accepted for processing
`rejected`	Rejected before processing
`queued`	Accepted into a queue
`streaming`	Accepted and a stream will follow
`cached`	A prior idempotent result was selected

An acknowledgement is not a successful business result. In the reviewed local submission path, an asynchronous action that passes preflight is queued and returns `queued`. A contract or credential violation, approval pause, transaction preview, or resolved idempotent replay can instead return an `ActionResult` immediately; other runtime failures can return a protocol error.

Result

`ActionResult.status` is one of `completed`, `failed`, `cancelled`, `pending_approval`, or `requires_human`. A result can also contain structured output, user-facing message parts, memory updates, usage, a receipt reference, or a typed protocol error.

Only `completed`, `failed`, and `cancelled` settle ordinary action execution. `pending_approval` and `requires_human` preserve a governed pause; the runtime projects both as `pending_approval` in the lifecycle view. A failed result must carry a structured error.

Lifecycle view

`ActionStatus` combines queue, result, transaction, approval, receipt, and stream state into an operational read model. Its stable enum contains:

State	Interpretation at the reviewed runtime
`unknown`	No visible queue record, result, or requested retained chunk was found
`accepted`	Reserved wire state; the current projection does not produce it
`queued`	A queue record is waiting for execution
`running`	A worker owns an executing queue record
`streaming`	No result exists and at least one retained chunk was requested and found
`pending_approval`	The result or queue state requires approval or human input
`cancelling`	Reserved wire state; the current projection does not produce it
`cancelled`	Cancellation is recorded as the action result or queue state
`completed`	A completed result or completed queue state exists
`failed`	A failed result or failed queue state exists
`expired`	Queue state expired under retention policy
`dead_lettered`	Retry processing parked the record with a dead-letter reason

The runtime treats `cancelled`, `completed`, `failed`, `expired`, and `dead_lettered` as terminal lifecycle states. `unknown` is a query outcome, not an execution stage. The projection can change as separate records arrive, so a client should retain the action ID and read the latest authorized view.

Mode changes the response pattern

Mode	Reviewed runtime path	Caller responsibility
Omitted or `sync`	Validates, authorizes, executes inline, and returns `ActionResult`	Bound the request and retain the action ID when the outcome is uncertain
`async`	Performs preflight, stores a queued record, and normally returns `Ack(queued)`	Persist the action ID and query status, result, or events
`streaming`	Executes through the non-async path, records chunks, and returns a final `ActionResult`	Consume ordered chunks and still use the result or lifecycle view as settlement authority

The capability contract, not caller preference, decides which modes are admitted. A long-running or approval-prone operation is normally easier to recover when submitted asynchronously, but the provider's actual contract is authoritative.

At the reviewed revision, asynchronous submission checks the contract, credential context, authorization, approval, transaction preconditions, idempotency, and handler presence before queueing. Capability input-schema validation occurs when the worker enters the normal processing path, so a queued acknowledgement is not proof that the input will execute.

Stream chunks are ordered action records

Every `StreamChunk` contains an action ID, a monotonically increasing `sequence`, a `kind`, and optional structured data or rich message content. Kinds are `data`, `progress`, `tool`, `thought`, `preview`, `pending_approval`, `error`, and `done`.

The reviewed lifecycle backend applies these rules atomically per action:

- the exact same sequence and content is classified as a replay and is not

inserted twice;

- the same sequence with different content is rejected;
- a new sequence must be greater than every retained earlier sequence;
- no chunk may follow a retained `done` or `error` chunk;
- protocol-edge ingestion rejects chunks for an unknown or already settled

action.

Action-event reads can return events and retained chunks behind one cursor. Clients should checkpoint that cursor and chunk sequence, handle reconnects idempotently, and treat the terminal action state as authoritative. These storage checks do not create an exactly-once guarantee for every transport, callback receiver, or external consumer.

Cancellation stops runtime work, not committed history

`Cancel` can target either an action or a session. For an action, the reviewed runtime authorizes write access, signals the active cancellation token, calls the handler's cancellation path when the action is active, and records cancellation in the lifecycle and queue stores. For a session, it performs an authorized transition to `cancelled` when that transition is legal.

A cancelled AIP result proves what the runtime recorded. It does not prove that an external provider reversed a side effect that had already committed. Provider uncertainty belongs in transaction reconciliation and compensation; blindly retrying a mutating action after a cancellation timeout can duplicate work.

The `cancelling` enum value leaves room for an intermediate read state, but the reviewed lifecycle projection does not currently emit it. Do not wait for that state before reading the actual result and transaction evidence.

Operational reads are authorization boundaries

The native read model supports four action operations:

Operation	Selectors and optional data
Read status	Action ID, tenant boundary, result, receipt chain, and retained chunks
Read result	Action ID, tenant boundary, receipt metadata, and terminal event hints
List actions	Lifecycle, capability, session, principal, approval, transaction, tenant, cursor, and result or receipt inclusion
Read events	Action ID, tenant, cursor, kinds, retained chunks, and follow preference

List and event limits are validated in the range 1–1000. Status and result requests accept `wait_ms` only up to 30,000 milliseconds. At the reviewed revision, the in-process runtime lookup is immediate and does not consume `wait_ms`; callers should not infer long-poll behavior from the field alone. The HTTP events binding implements follow mode separately through SSE.

These queries require transport-established identity and apply owner, delegated scope, tenant, and sensitive-field policy. A caller that cannot see an action must not use `unknown` or `not_found` to infer whether another tenant owns it.

A session is optional negotiated context

The protocol `Session` stores five facts: `id`, lifecycle `state`, `created_at`, `initiator`, and `responder`. `SessionView` adds the latest known update time, an optional expiration, a count of non-terminal bound actions, and operational links.

A native handshake requests profiles and optional capabilities. The reviewed gateway requires the claimed client to match the transport-authenticated principal, negotiates a shared profile, creates an `active` session, and returns its ID plus an opaque resume token. If no profile matches, it rejects the handshake and creates no session.

The `SessionState` transition model permits:

Current state	Allowed next states
`new`	`active`
`active`	`queued`, `processing`, `cancelled`
`queued`	`processing`, `cancelled`, `failed`
`processing`	`streaming`, `waiting_for_human`, `completed`, `failed`, `cancelled`
`streaming`	`completed`, `failed`, `cancelled`, `waiting_for_human`
`waiting_for_human`	`processing`, `completed`, `cancelled`, `failed`
`completed`, `failed`, or `cancelled`	`settled`
`settled`	None

This is the legal state graph, not a promise that action execution automatically drives every session transition. The reviewed session manager creates sessions directly as `active`; explicit runtime paths perform later transitions. Closing a session transitions it to `cancelled`, so it can fail when the current state does not permit that edge.

Resume tokens are rotating credentials

The reviewed runtime returns a random resume token when creating a session. It stores a SHA-256 hash, binds the token to the authenticated session owner, gives it an expiration, and rotates it atomically after a resume credential passes the owner, validity, and expiry checks. Reusing the old token is rejected.

A resume request supplies the session ID, current token, and optional last event cursor. On success, the response contains the session view, session-scoped events after the cursor, a next cursor, and the replacement token.

The reviewed runtime bounds one resume replay read to 100 events. Cancelled, failed, and settled sessions cannot resume. Invalid, expired, and replayed credentials that reach token verification collapse to `session.resume_rejected`; owner or scope authorization can fail earlier. A non-resumable lifecycle state returns `session.closed`.

The store rotates a valid token before the runtime rejects a closed lifecycle state, and that error response does not expose the replacement. Do not attempt resume on a session already known to be cancelled, failed, or settled.

Treat a resume token like a bearer credential:

- never put it in a URL, log, trace field, or analytics event;
- persist the replacement before discarding the previous response;
- serialize concurrent resume attempts so only one receives the next valid

token;

- pair the token with transport authentication for the same owner;
- close the session, or use a deployment-owned revocation path when one is

exposed, if continued resume access is no longer safe.

Session resume replays retained session events, not arbitrary application memory or provider history. Retention and cross-process survival depend on the configured event and session stores.

Trust and data boundaries

- An action or session ID selects a record; it does not authorize access to it.

The gateway and runtime use transport-established identity, owner, delegated scope, and verified tenant context for reads and mutations.

- `Action.input`, conversation context, memory, identity projections, results,

chunks, and receipts can contain sensitive data. Inclusion flags do not bypass sensitive-field authorization or retention policy.

- The caller supplies business intent, but the trusted ingress replaces

self-asserted action identity with resolved identity before execution.

- A resume token proves possession of one rotating session credential. It does

not replace transport authentication or grant action access outside that owner's policy.

- AIP records describe runtime state. The external provider remains the source

of truth for side effects whose completion is uncertain.

Design choices and trade-offs

Keeping actions independent from sessions lets a client invoke one operation without negotiating a long-lived relationship and lets many actions share one session when continuity matters. The cost is that clients must preserve action, session, message, and correlation identifiers explicitly instead of relying on one connection-local handle.

Separate acknowledgement, result, and lifecycle types prevent queue admission from masquerading as business completion. They also require clients to choose the correct read surface and reconcile records that can arrive at different times.

Pluggable stores make the runtime usable in tests, single-process deployments, and persistent installations. They also make durability an explicit deployment property. Rotating resume tokens limit replay, but require serialized resume attempts and careful replacement-token persistence.

Stable enums include `accepted` and `cancelling` even though the reviewed projection does not produce them. That preserves a vocabulary for other bindings or later implementations, while requiring documentation to distinguish wire capacity from observed implementation behavior.

What these models do not guarantee

- An action ID does not make a non-idempotent provider call safe to retry.
- An acknowledgement does not prove execution or external commit.
- A `done` stream chunk does not replace the final action result.
- A session does not authenticate its participants without the transport and

gateway trust path.

- Session grouping does not make multiple actions one atomic transaction.
- Cancellation does not compensate an already committed side effect.
- A lifecycle enum value does not prove the reviewed runtime currently emits

that value.

- A state-backed record survives a restart only when its configured backend is persistent and correctly operated.

Related pages

- Capabilities and contracts
- Approvals and policy
- Transactions and compensation
- Identity and trust
- HTTP API