Transactions and compensation

Use this page to understand how AIP represents a consequential operation before, during, and after a connector attempt. It is for application, runtime, and connector developers who need to plan a mutation, own one commit attempt, recover an

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

An AIP transaction is a durable coordination record around one or more action attempts. It is not a database transaction spanning the runtime, connector, and external product. Planning can bind intent, commit claiming can fence competing workers, and reconciliation can stop a blind retry, but none of these controls makes an external side effect atomic or exactly once.

This page describes the reviewed Rust implementation of AIP 1.0 at source revision `97be86e9efedf07ecf1783b03800f683f107fb04`. Concrete product semantics still depend on the selected capability, connector handler, provider operation, and durable backends configured by a deployment.

Keep five identities separate

Identity	Scope	What it binds
Action ID	One AIP invocation	Action lifecycle, idempotent replay checks, cancellation, and connector route assignment
Transaction ID	One durable coordination record	Plan, commit, execution, reconciliation, or compensation state
Plan ID	One immutable plan	A later commit to the planned capability and input
Provider operation ID	One provider-accepted operation	Checkpoint and reconciliation against the external system
Compensation target action ID	Earlier work to offset	The result and transaction evidence that make compensation eligible

These identifiers are correlated, not interchangeable. A transaction ID does not select a connector replica. A plan ID does not prove commit ownership. A provider operation ID does not prove that the provider committed. A compensation target does not guarantee that reversal is possible.

```
MERMAID
flowchart LR
    I["Governed action intent"] --> D["Dry run"]
    I --> P["Durable plan"]
    I --> E["Direct execute"]
    P --> C["Atomic commit claim"]
    E --> R["Connector attempt"]
    C --> R
    R --> K["Committed"]
    R --> F["Failed before known commit"]
    R --> U["Outcome unknown"]
    U --> Q["Reconcile provider operation"]
    Q --> Z["Reconciled"]
    K --> X["New compensating action"]
    X --> Y["Compensated or failed"]
```

Capabilities declare transaction support

`TransactionContract` contains only three declarations:

Field	Meaning
`supported_modes`	Transaction modes admitted for the capability
`requires_plan_before_commit`	Capability declaration that the runtime uses when selecting its commit branch
`dry_run_fidelity`	Strength of the advertised preview

The seven transaction modes are:

Mode	Reviewed runtime behavior
`execute`	Prepare and execute directly under a transaction ID
`dry_run`	Return a runtime preview or call a handler for declared downstream validation
`plan`	Create a durable plan, optionally after handler validation
`commit`	Atomically claim a plan, or use an internal direct-commit branch when its narrower preconditions apply
`compensate`	Validate a prior target and execute the declared compensation path
`reconcile`	Re-enter a transaction whose provider outcome is unknown
`rollback_not_supported`	Return an explicit non-retryable policy result

The four dry-run fidelity levels are deliberately different:

Fidelity	Meaning of the declaration
`schema_only`	Only schema validation is represented
`policy_and_schema`	Runtime schema and policy checks are represented
`downstream_validation`	The handler is invoked for provider-side validation without intended commit
`full_simulation`	The provider is declared capable of simulating the complete operation

For `schema_only` and `policy_and_schema`, the reviewed runtime creates its own preview without invoking the handler. For `downstream_validation` and `full_simulation`, it calls the handler and records the returned status and output inside a dry-run result. The fidelity value is a contract declaration; the generic runtime cannot prove that a connector or provider avoided side effects.

`CompensationContract` independently declares `not_required`, `supported`, `best_effort`, or `rollback_not_supported`. It can name a separate compensating capability, limit the compensation window, and require approval for the compensating action.

Action context carries the transaction intent

`Action.transaction` is the canonical semantic context. Its fields are the mode plus optional transaction ID, plan ID, and `compensation_for` action ID. The first-class `TransactionRequest` wraps the same action with a required transaction ID, requester, optional reason and metadata, and optional provider operation and reconciliation cursor.

The native request path checks that the `requested_by` ID matches the runtime principal ID and that the wrapper transaction ID matches the action when both are present. If the action omits the ID, the runtime copies the wrapper value into the action. The ordinary action path also creates a transaction ID when a mode needs one; a commit can recover it from the referenced plan.

The first-class wrapper and ordinary action path then share capability lookup, schema checks, authorization, approval, idempotency, transaction admission, and handler execution. The wrapper adds a `TransactionResult` and a transaction receipt chain; it is not a second transaction engine. Public envelopes first pass the core message validator, whose commit rule is stricter than one branch inside the runtime, as described below.

Preview comes before approval; mutation does not

The reviewed runtime handles `dry_run` and `plan` before creating a pending approval. Their result records whether later execution requires approval and can retain the applicable approval policy. This allows a caller or approver to inspect a stable preview without first authorizing the mutation.

`execute`, `commit`, `compensate`, and `reconcile` do not use that preview-first exception. They pass the normal approval gate before the runtime starts their pre-execution state transition. If a compensation contract requires approval, the runtime forces approval and supplies a default tenant-policy approval when no stronger policy was already selected.

Approval and transaction state remain separate. An approved action can still fail to claim a plan, reach a connector, or settle at the provider. A plan that records `approval_required` is not an approved commit.

A durable plan binds a bounded set of facts

For `plan`, the runtime creates a `TransactionPlan` and stores it in a `planned` transaction record. The implementation-generated plan expires after 15 minutes. Its default ID is `plan:` when the action does not supply one.

The plan retains:

Group	Stored values
Coordinates	Plan, transaction, planning action, and capability IDs
Requester	Principal that created the plan
Intent	SHA-256 input hash and an input snapshot
Governance	Predicted side effects, approval requirement and policy, compensation contract, and dry-run fidelity
Identity	The action identity snapshot
Time	Creation and expiration timestamps
Metadata	Commit requirements plus data, idempotency, and service-level contract snapshots

The plan input hash covers the capability ID and action input. It does not include `Action.transaction`. When the capability data contract requires redaction, the input snapshot contains only `redacted: true` and the hash. Otherwise it also contains the raw input value.

Downstream plan validation runs only for `downstream_validation` or `full_simulation`. The handler has to advertise transaction support. A failed validation creates a failed transaction record without a plan; a successful validation is retained in plan metadata and the transaction record.

What a planned commit rechecks

A commit that references a plan is admitted only when:

- the plan exists and its transaction record is still `planned`;
- the current capability ID matches the planned capability;
- the committing principal ID and kind match the planning principal;
- the plan has not expired;
- a supplied transaction ID matches the planned transaction ID;

- the hash of the current capability ID and input matches the plan hash;
- presence of identity context matches, and its tenant and external account

equal the planned values.

The commit check does not compare every field retained in the plan. In particular, it does not directly compare the entire approval policy, compensation, data, idempotency, service-level metadata, every identity field, or the planning action ID. Current capability, authorization, approval, schema, credential, and idempotency checks still run through the ordinary action path.

This bounded comparison is the implemented contract. A deployment that needs a stronger policy freeze has to enforce it at its own admission boundary or use a future protocol extension. It should not infer that stored metadata is already part of commit equality.

Commit ownership is atomic

After validation, `claim_plan` changes one durable record from `planned` to `committing`, associates it with the commit action ID, and increments its compare-and-set revision. A competing claimant observes a non-planned state and is rejected. Durable backends have to perform that state change atomically.

The runtime also contains a direct-commit branch. When the selected capability contract sets `requires_plan_before_commit` to false and the action has no plan ID, `claim_direct_commit` creates the transaction directly in `committing`; an existing transaction ID causes a conflict.

That branch is not reachable through a validated public AIP envelope at the reviewed revision. Core `validate_action` unconditionally requires `plan_id` for every `commit`, including a commit inside `TransactionRequest`. It is reachable only through a trusted direct runtime call that bypasses envelope validation.

Public clients therefore need a plan ID even when the capability contract field is false. The contract and validator are not yet one uniform direct-commit surface.

This claim fences control-plane ownership. It does not prove that only one network request reached the provider. Provider-side idempotency, a durable provider operation checkpoint, and connector-specific request semantics remain necessary at the side-effect boundary.

Direct execution has its own replay gate

`execute` creates a `prepared` record before handler execution. If the same transaction ID already exists, the action ID, capability ID, full transaction context, and identity snapshot have to match the stored execution contract.

The existing state then controls the outcome:

Stored state	Reviewed behavior
`prepared`	Continue the same attempt path
`committed`	Return the durable result when it belongs to the same action
Retryable `failed`	Reopen as `prepared` only with an idempotency key and no provider operation, cursor, or reconcile deadline
Other `failed`	Return the stored failure or `transaction.retry_denied`
`committing`, `outcome_unknown`, or `reconciling`	Reject re-execution with `transaction.reconciliation_required`
Any other state	Reject with `transaction.execute_state_invalid`

This replay gate is stricter than reusing a transaction ID as a correlation label. It treats the stored execution contract as immutable.

Checkpoint provider acceptance before uncertainty

A connector can publish a `ProviderOperationRef` through the execution context after a provider accepts work. It contains a provider name, provider-local operation ID, and optional request ID. Provider and operation ID must be non-empty, and a transaction cannot replace an existing reference with a different one.

The checkpoint can be written while the transaction is `prepared`, `committing`, `compensating`, or `reconciling`. It uses the transaction record's revision fence and can also retain an opaque reconciliation cursor. On another attempt, the connector execution context exposes the recovered provider operation ID and cursor without making the whole durable record part of the public action payload.

This checkpoint is the bridge between a request sent to a provider and later reconciliation. If a connector learns the provider operation ID but does not publish it before losing the response, the generic runtime cannot invent that identifier.

Unknown outcome blocks blind execution retry

For `execute` and `commit`, a completed action result settles the transaction as `committed`. A failed result becomes `outcome_unknown` when its error code is `sla.timeout_exceeded` or its error details contain `uncertain_outcome: true`. Other failed results settle as `failed`.

An uncertain record retains any provider operation parsed from error details, an available reconciliation cursor, the redacted error, and an earliest reconciliation time. The default delay is five seconds when the error does not provide `retry_after_ms`.

The runtime rejects `execute` while the record is `outcome_unknown`, `reconciling`, or `committing`. This is the central safety property: a timeout is not interpreted as proof that nothing happened.

Reconciliation asks the provider what happened

A `reconcile` action requires a transaction ID, an existing transaction in `outcome_unknown`, and a durable provider operation reference. The first-class request can provide the reference and cursor; otherwise the runtime recovers them from the transaction record.

The runtime uses a compare-and-set transition from `outcome_unknown` to `reconciling` before invoking the handler. The connector receives the provider operation ID and last cursor in its trusted execution context. A completed handler result settles the record as `reconciled`; a failed result returns it to `outcome_unknown` with updated recovery data.

`reconciled` means the reconciliation action produced a definitive result under the connector contract. The generic runtime does not reinterpret provider output as `committed`, roll back work, or automatically launch compensation.

Compensation is a new governed action

`compensate` is admitted only when the capability's compensation mode is `supported` or `best_effort`, and the action names `compensation_for`. The target normally needs a completed action result. A narrow partial-failure path also qualifies when a non-retryable error explicitly records both `side_effects_committed: true` and `compensation_required: true`.

If the target has transaction records, one must be `committed`, or a failed record must carry that partial-side-effect evidence. An optional compensation window is measured from the selected target record's last update. When no transaction record exists, the implementation can use an RFC 3339 `original_completed_at` value from action observability metadata. That fallback is caller-carried metadata, not equivalent to a durable completion timestamp.

The runtime creates a separate `compensating` transaction record for the new action. If `compensation_capability_id` differs from the requested capability, it resolves the alternate capability, rewrites the connector-facing action to that capability, and adds the original result and compensation coordinates to runtime memory context. The public action still identifies the governed compensation request.

A completed handler result settles the record as `compensated`; another result settles it as `failed`. Neither `supported` nor `best_effort` means that the original provider effect was erased. Compensation can itself need approval, idempotency, reconciliation, and operator follow-up.

Wire status and durable status are not identical enums

The durable runtime record has these statuses:

`dry_run_completed`, `planned`, `prepared`, `committing`, `committed`, `compensating`, `compensated`, `failed`, `outcome_unknown`, `reconciling`, `reconciled`, and `rollback_not_supported`.

The wire-facing `TransactionProtocolStatus` adds `requires_human` and `cancelled`. Those two values are projected from the surrounding action result; they are not durable `TransactionStatus` variants in the reviewed store.

Every durable transition names the allowed current states and expected record revision. The backend applies the update only when both match, then increments the revision. A transaction view can be selected by exactly one of transaction ID, plan ID, or action ID and can optionally include the terminal action result and receipt chain. Normal transport identity, owner, tenant, and query scopes still govern that read.

Connector routing is action-scoped

The connector registry's immutable `RouteAssignment` is keyed by action ID. It pins the capability, verified tenant, connector instance and replica, endpoint, peer identity and DID, version and manifest, policy and credential revisions, health revision, admission reservation, and fence token.

Resolving the same action ID again reuses that assignment only when capability and tenant still match. Retries therefore remain on the pinned route, and cancellation looks up that same action assignment. A reconciliation action also reuses it only when it carries the same action ID.

There is no transaction-ID or plan-ID lookup in the reviewed route resolver. Distinct plan, commit, compensation, or reconciliation action IDs resolve independently and may select different replicas. A schema- or policy-only plan does not invoke a connector at all, so it creates no remote assignment.

This boundary is important for connector-local state. An integration that requires plan and commit on one provider-side replica cannot infer that stickiness from the transaction record. It needs an action-scoped route that is actually reused or a provider operation that is portable across replicas.

Trust and data boundaries

Boundary	Runtime evidence	What it does not prove
Capability to caller	Supported modes, fidelity, and compensation declaration	That the connector honors the declaration or has been qualified
Plan to commit	Exact implemented principal, expiry, capability, input, transaction, tenant, and account checks	Equality of every policy or identity field retained in plan metadata
Commit claimant to store	Atomic state and revision transition	Exactly-once delivery to an external provider
Runtime to connector	Trusted transaction ID, provider operation ID, and cursor	Provider outcome before the connector reconciles it
Action to route registry	Immutable action-scoped assignment	Transaction-wide replica affinity across distinct action IDs
Original action to compensation	Durable result or explicit partial-side-effect evidence	Guaranteed reversal or restoration of all external state
Operator to transaction view	Authorized durable record, optional result, and receipts	State newer than the selected store revision or provider truth not yet reconciled

Input snapshots can contain raw action data when the capability does not require redaction. Transaction stores and evidence exports therefore need the same tenant isolation, retention, and access controls as the governed action.

Design choices and trade-offs

- Explicit modes separate preview, ownership, recovery, and reversal intent, but connector authors have to implement the declared semantics correctly.
- A bounded plan comparison is deterministic and inexpensive, but stored plan metadata is broader than the fields currently enforced at commit.
- Atomic commit claims prevent two runtime workers from owning the same durable plan, but provider-side idempotency is still required.
- Provider-operation checkpoints make uncertain outcomes recoverable, but only when a connector publishes the identifier before losing it.
- Reconciliation blocks blind retry and preserves evidence, at the cost of a provider-specific lookup path and durable work scheduling.
- Compensation models business reversal honestly as another action, but it cannot provide distributed rollback.
- Action-scoped route pinning keeps retries and cancellation consistent, but it does not create transaction-wide connector affinity.

What transactions do not establish

The reviewed implementation does not establish any of the following:

- distributed ACID behavior across runtime, connector, and provider;
- exactly-once provider effects;
- proof that `dry_run` or `plan` is side-effect-free beyond the declared and implemented handler contract;
- automatic equality of every plan metadata or identity field at commit;
- a public planless commit merely because `requires_plan_before_commit` is false;
- permission to retry an `outcome_unknown` operation without reconciliation;
- automatic conversion of `reconciled` into `committed` or `compensated`;
- guaranteed compensation, even when the mode is `supported`;
- route reuse across distinct action IDs merely because their transaction or plan ID matches;
- process durability when an in-memory transaction backend is selected;
- live qualification of these paths against a particular provider.

Related documentation

- Capabilities and contracts explains how transaction, idempotency, data, and compensation support is declared.
- Actions and sessions owns action identity, queue, result, cancellation, and replay behavior.
- Approvals and policy explains the durable approval gate used before commit and compensation.
- Observe and recover provides operational recovery workflows for uncertain and stalled work.
- Connector registry and routing owns route assignment, admission, settlement, and replica selection details.