CrewAI idempotency, cancellation, and process fencing

This reference explains how the CrewAI connector fences duplicate work, persists job identity, and reports cancellation. Use it to recover an Action without treating a transport result as proof of provider behavior.

SECTION	Connectors
TYPE	Connector
PROTOCOL	AIP 1.0
SOURCE	docs/connectors/crewai/reference/idempotency-cancellation-and-process-fencing.md

The reviewed implementation provides durable admission, not provider exactly-once execution. A terminal sidecar record can coexist with uncertain external effects when CrewAI work continues outside its asyncio task.

Keep three fences distinct

Fence	Identity or resource	What it prevents	What it does not prove
AIP Action	Immutable Action ID, input, tenant, target, and AIP idempotency key	Re-admission of changed AIP work under one identity	That CrewAI produced no external effect
Sidecar request	Action ID plus canonical request hash	Reusing one sidecar job ID with different material	Provider exactly-once behavior
Journal owner	Companion file lock for one state path	Two active sidecar processes writing one journal	Distributed or active-active ownership

Preserve all three identities in evidence. Replacing an Action ID to bypass a failed record creates new provider authority and can duplicate an earlier effect.

Understand the durable request identity

Every startable operation requires an `Idempotency-Key`. The sidecar rejects a missing or blank key, a value over 512 UTF-8 bytes, or a code point below U+0020.

The sidecar calculates SHA-256 over canonical JSON containing:

- `crew_id`;
- `action_id`;
- `operation`;
- the complete operation `input` object;
- `timeout_ms`;
- the idempotency key.

The Action ID is the journal key across every admitted crew. An exact repeat returns the existing record. Reuse with a different request hash returns HTTP `409` and does not schedule new work.

The runtime inserts the record and persists the journal before creating the asyncio task. Persistence failure removes the in-memory record and returns HTTP `507`, so provider work has not been scheduled through this path.

After successful persistence, later failures can leave the provider outcome unknown. The fence keeps the Action record available for status and event reconciliation.

Preserve the journal fence

Outside explicit loopback development, the sidecar requires a durable state file. Its parent must be a non-symlink directory without group or world write permission.

The runtime opens a companion `.lock` file as a regular file, forces mode `0600`, and acquires an exclusive nonblocking `flock`. Startup fails when another process owns the same journal.

Use one state volume for exactly one active sidecar process. Horizontal scaling requires independent connector instances and independent state volumes. This lock is local to the mounted filesystem and is not a distributed lease.

The journal must be a non-symlink regular file with mode `0600`. Restore also checks its byte bound, retained-record bound, unique Action IDs, event count, and per-event encoded size.

Persistence writes a mode-`0600` temporary file in the journal directory, flushes and `fsync`s it, replaces the journal atomically, then `fsync`s the directory. This protects one writer from partial replacement, subject to the filesystem's durability contract.

Interpret restart fencing

A restored `running` record cannot resume its Python task. The sidecar changes it to `failed` and appends a terminal event with `uncertain_outcome: true`.

Do not submit a replacement Action immediately. First reconcile the original Action with provider logs, tool effects, generated artifacts, and downstream systems.

The journal stores the request hash, events, status, and output. It does not store the original input or idempotency key, so retained dispatch evidence is also required for complete reconciliation.

Read each cancellation path precisely

Cancellation behavior depends on when the runtime observes cancellation and which connector method is active.

Path	Connector result	Sidecar request	Safe interpretation
Before a streaming HTTP response	`cancelled_unconfirmed` failure	None proved	Dispatch may have raced with local cancellation
During an events-read stream	Cancelled, `remote_stop_confirmed: false`, `dispatched: false`	None	The reader detached; the target job was not cancelled
During a streamed start	Cancelled, `remote_stop_confirmed: true` after sidecar success	`POST /jobs/{id}/cancel`	The sidecar accepted cancellation; provider stop remains unproved
During a non-stream invocation	Cancelled, `remote_stop_confirmed: false`, `dispatched: false`	Not issued by this select branch	The dispatch flag is not reliable across the race
Explicit cancel capability	Result from the cancel route	`POST /jobs/{target}/cancel`	Terminal jobs are unchanged; running sidecar tasks are cancelled
Stream validation failure after admission	Original stream error when cancel succeeds	Attempted for five cancellable operations	Provider effects may already exist
Stream validation failure with failed cancel	`cancelled_unconfirmed` failure	Attempt failed	Reconcile before any replacement Action

The non-stream select can return `dispatched: false` while its HTTP future has already sent bytes. Treat this field as local branch reporting, not network or provider evidence.

The explicit AIP cancel capability has its own Action identity and policy decision. The private sidecar route receives the target crew and Action ID but does not receive that AIP cancel Action's idempotency key.

Separate sidecar cancellation from provider stop

The sidecar cancellation handler closes an attached streaming output when possible, cancels the owning asyncio task, waits for that task, and persists the record. The task records `cancelled` when it receives `asyncio.CancelledError`.

Operation	Execution mechanism	Provider-stop boundary
`run`	Native async kickoff and optional async stream	Closing the stream and cancelling the task does not prove rollback of model or tool effects
`batch_run`	Native async methods when present; synchronous fallback in a worker thread	The fallback thread can continue after its waiter is cancelled
`replay`	Synchronous CrewAI call in a worker thread	The thread can continue
`train`	Synchronous CrewAI call in a worker thread	Training and artifact writes can continue
`test`	Synchronous CrewAI call in a worker thread	Model and evaluator calls can continue
`knowledge_query`	Async method or worker-thread fallback	Cancellation is not advertised for this operation
`memory_reset`	Synchronous CrewAI call in a worker thread	Cancellation is not advertised for this operation

The connector advertises runtime cancellation for `run`, `batch_run`, `replay`, `train`, and `test`. The sidecar route itself accepts any running job, including operations without advertised cancellation support.

`remote_stop_confirmed: true` means only that the private cancel route returned success. It is not a provider receipt, rollback record, thread-termination proof, or confirmation that external tools stopped.

Interpret timeout as an uncertain boundary

The sidecar wraps operation execution in `asyncio.wait_for` using the request timeout. The blocking HTTP route separately waits for the same task with the same duration and returns HTTP `504` when its wait expires.

For thread-backed operations, cancelling the asyncio waiter does not terminate the Python worker thread. A timeout can therefore produce a failed or cancelled record while provider work continues.

An ordinary execution exception emits a generic terminal `failed` event with the exception class. It does not set `uncertain_outcome`; only restart recovery adds that field at this revision.

Interpret connector failures

Condition	Category	`uncertain_outcome`	`retryable`
Sidecar `401` or `403`	Auth	False	False
Sidecar `429` or `5xx`	Temporary	True only for connector invocation	False
Other sidecar status	Permanent	False	False
HTTP transport failure	Transport	True only for connector invocation	False
Invalid or incomplete stream	Connector	True	False
Response limit exceeded	Connector	True only for connector invocation	False
Cancellation not confirmed	Temporary	True	False
Invalid credential, configuration, or input	Auth or permanent	False	False

The connector marks every reviewed `ConnectorFailure` as non-retryable. A caller must not convert a temporary category into an automatic retry without first reconciling the original Action.

The invocation-wide uncertainty rule is coarse. It can mark a read operation uncertain because reads and mutations share the same connector operation classification.

Choose a recovery action

Evidence	Recovery decision
Exact repeat returns completed	Use the retained result; do not dispatch again
Exact repeat returns running	Continue status or event observation within policy
Restored record reports uncertain failure	Reconcile external effects before retry or compensation
Cancel route succeeded for an async stream	Confirm provider and tool state independently
Cancel route succeeded for thread-backed work	Assume work may continue; contain credentials and downstream authority if necessary
Request changed under the same Action ID	Correct the caller; the `409` collision is protective
Journal is full	Archive only terminal records after the AIP replay-retention window
Journal owner lock fails	Stop the new replica; never bypass the lock or share the writable state path

Retain the original Action, request digest, journal record, event cursor, sidecar logs, provider request evidence, tool effects, and operator decision. Compensate only through a separately authorized Action.

Related documentation

- CrewAI connector overview
- CrewAI sidecar contract and lockfile
- CrewAI run lifecycle and replay
- CrewAI training and testing
- Errors and retry decisions