On this page >

← Blog (https://runcycles.io/blog/)

August 7, 2026 · Albert Mavashev · 5 min read

engineering | reliability | SDKs | recovery | langchain | runtime-authority

Copy link | Share | Share | PDF (https://runcycles.io/pdfs/the-model-call-succeeded-then-the-process-died.pdf)

The Model Call Succeeded. Then the Process Died.

The budget check passed. The model returned an answer. The provider reported the actual token usage. Then the connection failed while the application was committing that usage to its ledger.

At that point, the expensive action is no longer hypothetical. It happened. But the client may not know whether the commit reached the server, and the server may reclaim the reservation before the client comes back.

This is the accounting failure that clean request-response demos miss:

```text
reserve allowed
    → provider completed the model call
    → actual usage became known
    → commit response was lost
    → process stopped
```

If the client releases the reservation, it returns capacity for spend that already happened. If it forgets the commit, the ledger undercounts the action. If it invents a new retry key, an ambiguous first commit can be charged twice.

Cycles now handles that sequence through one tested recovery contract across its official SDKs.

A reservation is not the final accounting record

Cycles uses a reserve-commit lifecycle ↗:

1. reserve authority before an action executes;
2. run the model or tool only after an allowed response; and
3. commit the actual amount after the action completes.

The reservation prevents concurrent work from spending the same budget. It is still temporary. Its lease can expire, and a successful commit can become ambiguous when the response disappears between the server and client.

That makes the execution boundary decisive:

- **Before the handler runs**, a failed reserve must stop the action.
- **While the handler runs**, a heartbeat failure is observable, but cannot make an already-started side effect safe to undo.
- **After actual usage is known**, settlement failure must preserve the known amount rather than release it.

Treating every exception as "release in a `finally` block" collapses those three phases into one unsafe rule. It is convenient control flow and incorrect accounting.

Persist before the first commit attempt

The durable path starts as soon as the lifecycle helper knows the amount it must settle. Before sending the first commit request, the SDK writes the settlement intent to its local journal.

That record contains the reservation, subject, action, amount, metrics, metadata, and stable settlement identity needed for replay. A process restart therefore changes who performs recovery, not what is being recovered.

The default journal is under `~/.runcycles/commit-journal`. Records are partitioned by server and principal so unrelated tenants or deployments do not replay each other's work. A terminal result removes the record; an unresolved result leaves it available for the next client lifecycle.

This ordering matters. An in-memory retry scheduled before durable persistence still has a process-crash gap. Persisting only after the first HTTP request has the same problem: the process can stop between the request and the write.

Recovery keeps the same identity

An absent response does not prove that the server did nothing. The only safe way to resolve that ambiguity is to retry the same operation with the same idempotency key.

Depending on the result, recovery follows one of four paths:

Result	Recovery behavior
Network error, 5xx, or eligible 429	Keep the journal entry and retry with the same key
Authentication failure	Keep the known spend, stop the current attempts, and replay after credentials are repaired
Reservation expired	Record the actual spend through idempotent `POST /v1/events` recovery
Recognized terminal rejection	Stop and surface the result without releasing known spend

The expired-reservation path is important. Once the server has reclaimed the hold, repeatedly committing the dead reservation cannot restore the missing charge. The durable record instead becomes a direct event carrying the same subject, action, actual amount, metrics, and recovery linkage.

This is idempotent accounting recovery. It is not a claim of exactly-once model or tool execution. The external action may need its own idempotency contract.

Error surfacing is separate from recovery

Framework adapters have to decide what the current caller observes when a commit cannot complete synchronously.

For example, `langchain-runcycles` supports two settlement policies:

- `"raise"` queues durable recovery and then raises the settlement error;
- `"log"` queues the same recovery, logs the synchronous failure, and returns the model or tool result.

The accounting guarantee is the same. The difference is application control flow.

That distinction matters for non-idempotent tools. If an email was sent or a payment was submitted, raising into an autonomous agent loop may cause the agent to repeat the side effect. `"log"` can be the safer caller policy even though settlement remains durable underneath it.

One wire protocol was not enough

Python, TypeScript, Java, and Rust could all send valid Cycles requests while still making different decisions after a timeout, restart, lease expiry, or credential failure. Wire compatibility alone does not prove that their ledgers converge under failure.

Cycles therefore publishes a shared 12-scenario recovery profile and per-SDK matrix ↗. The current snapshot contains 48 visible scenario results across the four official SDKs, each pinned to an implementation commit and backed by named native tests.

The scenarios cover the behavior that matters at the failure boundary, including:

- an ambiguous first commit;
- replay after process restart;
- concurrent journal workers;
- credentials failing after execution;
- an expired reservation requiring event recovery;
- heartbeat failure; and
- the boundary where actual usage first becomes known.

The runner does not merely validate JSON shapes. It executes the SDK's bound native behavior tests in a fresh process and publishes the evidence report. The settlement durability contract ↗ documents what each pass means and where the guarantee stops.

The guarantee has a boundary

No client can recover information it never received.

If the process dies before the provider returns final usage, the SDK cannot invent the missing token count or cost. Applications that need protection across that boundary must durably checkpoint the provider receipt or reconcile from provider billing telemetry.

The durable guarantee begins after the lifecycle helper knows the amount and persists it. From there, a lost response, restart, credential repair, or expired reservation does not turn known spend into returned budget.

Being explicit about this boundary is more useful than claiming a system can never lose accounting data. It tells operators which part Cycles owns and which evidence their application still has to preserve.

Available now

The hardened lifecycle is available in:

- `runcycles` Python 0.5.3 ↗;
- `runcycles` TypeScript 0.4.3 ↗;
- Cycles Spring Boot starter 0.3.3 ↗; and
- `langchain-runcycles` 0.4.0 ↗.

The Rust SDK already implements the shared durable recovery profile and is included in the four-SDK evidence matrix.

For LangChain agents:

```bash
pip install "langchain-runcycles==0.4.0"
```

For the core Python lifecycle:

```bash
pip install "runcycles==0.5.3"
```

For the complete operational argument—latency, fail-closed reserve behavior, ledger reconciliation, and recovery evidence—read what we had to prove before putting an agent gate in the critical path ↗.

Inspect the evidence directly: SDK recovery conformance ↗, settlement durability ↗, and LangChain budget control ↗.

MORE FROM THE BLOG

Proving a Critical-Path Agent Safety Gate
July 31, 2026
(https://runcycles.io/blog/proving-a-critical-path-agent-safety-gat

Closing the Estimate-Actual Gap with cost_fn
May 15, 2026
(https://runcycles.io/blog/langchain-runcycles-cost-fn-actual-cost)

Webhook Idempotency for Agent Budget Events
April 24, 2026
(https://runcycles.io/blog/webhook-idempotency-patterns-for-ai-ag

e)

ent-budget-events)

← Back to all posts (https://runcycles.io/blog/)