Can AI Agent Replays Double-Execute? SQLite Experiment

Exzil Calanza

Independent research preprint. This work is not peer reviewed.

Source commit: db7d2360b4cf62128dccc3d0bdd8fb96be7113eb

The reported counts characterize only the frozen evidence described here; they are not universal reliability estimates.

Independent research preprint — 21 November 2026

Author: Exzil Calanza

Platform research purpose: Answer a practical AI-systems reliability question with an independently checked, executable local experiment. This is not a claim about a deployed agent framework, a cloud payment processor, or production network reliability.

Abstract

Question. If an agent tool finishes an action but its acknowledgment is lost, will a retry perform it again? A documented Pydantic AI durability change addressed repeated evaluation dispatch on replay [1]; that upstream report motivates this study but is not a measurement made by us.

Method. We evaluated two actual SQLite write policies on the same deterministic reply-loss traces: every-attempt insertion ("blind retry") and an atomic transactional key/receipt ledger ("stable key"). Each policy received 10,000 logical actions at each hypothetical independent post-commit acknowledgment-loss probability of 5%, 15%, and 30% (three paired simulations of 10,000 logical actions per policy; four attempts maximum; seed 20261102). Unlike the earlier purely mathematical stable-ledger model, this supplemental experiment physically inserted and independently counted SQL effect rows and ledger records. It included connection reopen, same-key/different-payload refusal, exception rollback, and a 12-caller contention control.

Result. At the assumed 15% lost-reply rate, blind retry committed 11,758 SQL effects for 10,000 intended actions, including 1,758 extra effects across 1,525 affected actions. The stable-key SQLite policy committed 10,000 effects, with zero duplicate-key effect groups on an independent read-only database check. Seven actions still lacked a visible acknowledgment by attempt four under both policies. The result is about a local transaction design and synthetic receipt streams, not an end-to-end exactly-once guarantee.

Audience question and primary-source context

An engineering question is: "After an agent times out, how do we know whether retrying will repeat a charge, notification, publication, or evaluation?" The Pydantic AI maintainers described a case of evaluation dispatch on durable replay in upstream PR #11577, merged on 28 October 2026 [1]. The resulting software change is a motivating example only; our experiment did not benchmark or exercise Pydantic AI.

SQLite documentation describes transactional locking and atomic commit behavior [2,3]. Microsoft's Idempotent Tool Actions pattern describes persistent operation identity and payload binding [4]. Temporal documentation explains why retried Activities must be designed for repeat execution [5]. These sources were accessed from ZEKE on 21 November 2026 and returned HTTP 200. They provide design context; all numerical measurements below originate from our local SQL experiment. No cited source is represented as published within the previous seven days.

Hypothesis and experimental design

H1. When a caller repeats an already-committed operation after losing the reply, blind retry produces extra local effects. Stable-key retry can suppress duplicate local effects if effect and receipt persist in the same SQLite transaction.

Experimental unit and sample. One logical action, represented by a unique operation key and immutable payload digest, is the unit. For each probability p in (0.05, 0.15, 0.30), we evaluated 10,000 units per policy using the same precomputed four-attempt acknowledgment sequence. The simulated acknowledgment can be lost after the SQL effect is committed; this isolates uncertainty of receipt from uncertainty of execution. We did not sample any real incident rate or network.

Policy A — blind retry. Each attempted call inserts a new committed effect row; no stable operation key prevents a second insert.

Policy B — transactional stable-key retry. Each attempted call begins an immediate SQLite write transaction. If the key exists with an equal payload digest, the original stored receipt is returned and no new effect is inserted. If the key exists with a different digest, the call fails. Otherwise, the effect and its key/receipt are inserted in the same transaction before commit. Database mode: WAL, synchronous=NORMAL.

Retry trace and metrics. Python 3 standard library random.Random(20261102) supplies four independent acknowledgment draws per unit, whether or not an early draw succeeds. A caller attempts until the first visible acknowledgment or the fourth attempt. Metrics are committed SQL effect rows, extra effect rows over 10,000, affected logical actions, and actions still lacking an observed acknowledgment. Unacknowledged does not mean no effect: the modeled first attempt already committed.

Boundary controls. The experiment reopened SQLite connections after 2,500-operation batches (12 reopen events), retried the same key after reopening, attempted the same key with changed payload, injected an exception between two same-transaction writes to confirm rollback, and made 12 concurrent separate-connection calls with one key. These controls do not simulate OS crashes or external systems.

Results: committed database records

Each probability is a paired synthetic simulation, not a measured production failure rate: the blind and stable-key policies receive the same 10,000 logical actions and the same frozen acknowledgment traces. Across three scenarios, 30,000 logical action keys were evaluated per policy. The two policies are not independent samples of users or real incidents.

Table 1. Blind-retry committed effects, from independent SQL row counts.

Assumed reply loss	Logical actions	Committed effect rows	Extra effect rows	Actions affected
5%	10,000	10,502	502	486
15%	10,000	11,758	1,758	1,525
30%	10,000	14,079	4,079	2,963

Table 2. Stable-key transactions and unknown acknowledgments, from the same paired traces.

Assumed reply loss	Stable-key effect rows	Duplicate-key effect groups	Unacknowledged after four attempts
5%	10,000	0	0
15%	10,000	0	7
30%	10,000	0	69

At the 15% hypothetical loss rate, the blind policy wrote 17.58% more effect rows than the intended 10,000 actions. The transactional policy wrote one effect per stable key for the tested local scenarios. The zero column was measured by SQLite effect and key-group queries, not assigned by an idealized ledger assumption. Both policies share the same fixed trace within each scenario. The three scenarios use hypothetical input probabilities and provide no independent production sample, confidence interval or measured incident rate.

The independent verifier opened the completed SQLite database read-only and checked effect and ledger group counts, unique effects per stable key, stored results, and SQLite PRAGMA integrity_check. Its result was pass=true for all three parameter settings. Twelve racing callers of the same key yielded one effect and one saved receipt; the changed-payload attempt was rejected; a rollback left no partial committed effect; and the same key returned the saved receipt after connection reopening. Two auxiliary control keys make the final experimental DB contain 30,002 stable-key effect and ledger rows rather than 30,000; they are excluded from the table.

Reproduction and evidence

The public reproducibility ZIP [6] contains the runnable experiment, independent verifier, frozen results and SQLite database. ZEKE fetched it publicly on 22 November 2026 (HTTP 200; ZIP SHA-256 57b8bf93598f3dd9fa686afe65de438ffdf5de569987de121bcfb5503f89337b); its four core files matched the frozen hashes.

Extract [6] and run python verify_sqlite_replay.py to check the archived database. For a new trial, copy both Python scripts to a fresh empty directory; run python sqlite_replay_experiment.py and then python verify_sqlite_replay.py. A fresh 22 November ZEKE run reproduced the frozen results JSON and database hashes exactly. No external services or production data are involved.

Frozen SHA-256: experiment b7555992c214790f161b71203f0ebd0f38e5dbdf61e55064a23ff0c0788f134c; verifier d2c26335eb1b9cc2b428cfe2aa039befa678c4cc534c7c6877db20513005fc5e; results 664c648f3edcc339f3ba561fd30d43baa8063ed7569eb57024bcfed3b17f1eff; SQLite DB 60cec5e6af912474b827147b034bda2b28925f20dce443ffa7127e35c0885a65. Synthetic assumptions and power-loss limitations still apply.

Interpretation and limitations

Reliable retry requires two separate answers. First, can the service recognize a replay by the same logical operation identifier and payload? Second, can the caller reconcile an unknown acknowledgment with the committed record? Duplicate suppression does not by itself recover an unreadable receipt.

The measured effect boundary was inside one SQLite database transaction. A network payment, email, third-party publication, or queue delivery may not participate in that transaction; a saved key alone cannot guarantee exactly-once external effects. To extend the design to an external side effect, one would need a supported idempotent downstream API, an outbox/reconciliation process, or a different independently tested commit protocol.

WAL with synchronous=NORMAL and connection reopen does not prove power-loss durability. We did not simulate an operating-system crash, disk corruption, network partition, host failover, transactional key expiry, real LLM inference, adverse scheduling across machines, or any actual agent/SDK. The 5%, 15% and 30% values are hypothetical sensitivity assumptions, not estimated incident rates. No cross-service safety guarantee or universal failure probability follows from the recorded counts.

Conclusion. Under deterministic lost-acknowledgment simulations, blind retry replicated committed local effects. A SQLite transaction binding effect, operation identity, payload fingerprint, and saved receipt prevented extra local effects under the tested scenarios; uncertain caller acknowledgments remained. This is a narrow reproducible engineering result, not a promise of exactly-once behavior for agent systems in general.

A timeout is not evidence that the action never happened. Preserve a replay-stable key and reconcile the effect.

References

[1] Pydantic AI maintainers, PR #11577, "OnlineEvaluation" replay dispatch change, merged 28 October 2026. https://github.com/pydantic/pydantic-ai/pull/11577

[2] SQLite, "Transaction". https://www.sqlite.org/lang_transaction.html

[3] SQLite, "Atomic Commit in SQLite". https://www.sqlite.org/atomiccommit.html

[4] Microsoft, "Idempotent Tool Actions" agent design pattern. https://github.com/microsoft/agentdesignpatterns/blob/main/docs/patterns/idempotent-tool-actions.md

[5] Temporal, "Activity Definition" and idempotency. https://docs.temporal.io/activity-definition

[6] Exzil Calanza, public SQLite replay reproducibility bundle, 21 November 2026. https://github.com/Zek21/Skynet-AI/releases/download/formal-research-ai-agent-replay-duplicate-side-effects-10000-runs-2026/ai-agent-replay-sqlite-reproducer-20261121.zip

Context sources checked from ZEKE on 21 November 2026; public experiment archive checked on 22 November 2026. The manuscript is original independent research; listed upstream documents are attributed to their authors. This manuscript has not yet received the current revision's independent publication reviews.

Signed by Skynet.