Safe Retry After an Ambiguous Tool Timeout

A Hand-Checkable Conformance Suite for Agent Recovery with Provider Contracts and Webhook Evidence

Anish Prashun
Independent researcher

Independent technical report Version 1.0 November 2026
DOI: 10.5281/zenodo.24304849

Abstract

A tool call with a side effect can time out after the provider has acted, so retrying may act twice. We present a small, deterministic, hand-checkable conformance suite for the recovery layer around one such action, sending a synthetic message, under two explicit provider contracts: no deduplication, and deduplication on a stable key. Its 16 scripted scenarios (17 operations) and 120 timing schedules cover lost acknowledgements, late commits, lost requests, stale status reads and webhook receipts that are delayed, duplicated, reordered, lost or misleading. Scripted policies are scored against a private effect ledger; no language model is involved. The suite replicates known results: immediate retry and a single stale status read duplicate effects, stable idempotency keys remove those duplicates, fresh keys per attempt restore them, and a lost request cannot be told apart from an unacknowledged commit. It adds five things. For a reconciler whose evidence store is monotone and keyed by attempt, receipt delay, duplication and reordering cannot change a decision, so sequence numbers are unnecessary; receipts still reduce unresolved outcomes and latency. Evidence-based retry is only as safe as the provider's guarantee that a failed attempt never commits. A sensitivity table shows which baseline counts move under equally reasonable rules. After a lost acknowledgement, wait-then-verify duplicates exactly when the status lag is at least $\tau + 4$ ticks and the receipt arrives at $t \geq \tau + 7$. A contract checklist, decision table and replay tool support use on other providers. Counts are over authored cases, not rates.

1 Introduction

An agent that sends a message or charges a card through a provider's API may see the request time out. The provider may never have received it, may have executed it and lost the reply, or may still be executing it. Retrying risks a second external effect; giving up risks losing the work. Providers offer idempotency keys [5, 6], status endpoints whose answers may be stale [7], and webhook receipts that can arrive late, more than once, out of order or not at all [8, 9].

This report studies the client's recovery layer for one such action in a deliberately small setting: a deterministic, hand-checkable conformance suite with two explicit provider contracts. A synthetic "send message" call is driven through 16 scripted scenarios (17 operations) and 120 timing schedules. Six client policies, one ablation and two references that read hidden state are scored against a private effect ledger. No language model is involved, and every outcome can be traced event by event.

Most of the comparison replicates known results. Recent agent-recovery evaluations, above all LIMBO [16], already show that blind retry and single verification reads duplicate effects, that stable

keys largely prevent this, and that some timeouts cannot be resolved by verification alone [16, 14, 15, 17, 18]. We reproduce these results in a model small enough to check by hand. The extension is webhook receipts used as recovery evidence together with stale negative status reads, which we did not find in the agent-recovery evaluations we reviewed.

Contributions. Replicated: immediate retry and a single stale status read duplicate effects; a stable idempotency key removes policy-caused duplicates and a fresh key per attempt restores them; and a lost request and an unacknowledged commit give byte-identical observations (the "indistinguishable twins"), so no policy is both duplicate-free and complete on the pair. New:

1. **Webhook receipts as evidence.** For a reconciler whose evidence store is monotone and keyed by attempt, receipt delay, duplication and reordering cannot change a decision, and sequence numbers are unnecessary. Receipts still matter: ignoring them raises the unresolved grid schedules from 16 to 41 of 120. Handling failure-named receipts by name makes an immediate-retry client duplicate in 2 scenarios (Section 5.3).
2. **The terminal-negative guarantee.** Evidence-based retry is only as safe as the provider's guarantee that an attempt reported as failed never commits (Section 5.4).
3. **Sensitivity.** Which baseline counts move under equally reasonable rules, and which conclusions survive every variant (Section 5.5).
4. **Boundaries, not rates.** After a lost acknowledgement, wait-then-verify duplicates exactly when the status lag is at least $\tau + 4$ ticks and the receipt arrives at $t \geq \tau + 7$ (Section 5.6).
5. **Tooling.** A provider-contract checklist, a decision table, `explore` and `replay` commands and a trace schema (Section 6).

What the counts are. All counts are over authored cases: they show mechanisms and timing boundaries, not rates, and they do not rank the policies.

2 Background and related work

The classical ambiguity. A client that times out cannot tell a lost request from a lost reply: the coordinated-attack problem of distributed systems [1, 2]. Test practice records such an operation as indeterminate; in Jepsen histories an `:info` operation "may or may not have completed" [3].

Idempotency keys, status reads and webhooks. The usual remedy is an idempotent request [4]. For its v1 API, Stripe saves the result for a key once execution begins, returns it, errors included, to later requests with that key, and rejects reuse with different parameters [5]; AWS describes the same pattern [6]. A status check can be answered from stale state [7], and a "not found" after a commit is the stale negative read that misleads a check-then-retry client. Stripe states that endpoints "might occasionally receive the same event more than once" and does not guarantee delivery order [8]; Shopify states that "Webhook delivery isn't always guaranteed" and recommends reconciliation jobs [9]; GitHub keeps a delivery's identifier on redelivery and does not redeliver failed deliveries automatically [10]. Gupta describes, without an evaluation, a webhook-plus-polling status architecture with a forward-only state machine against out-of-order callbacks [11].

Recovery evaluations for agents. UndoBench [12] and Verified Tool Calls [13] show that naive retry after a lost acknowledgement duplicates effects and that verifying first helps; the latter also injects delayed visibility. Did It Happen? [14] and RetryBench [15] score recovery against hidden worlds that share one observation, and RetryBench scores whether each retry was justified. LIMBO [16], the closest work, has never-executed, lost-response and late-commit worlds, eventually consistent reads, key-optional and keys-everywhere contracts, wait-then-verify, escalation and state and outcome oracles, and proves that no verification-only policy is exactly-once under late commits without a bound on in-flight time. Permission to Retry [17], Evidence-Guided Recovery [18] and AID-Guard [19] keep ambiguous outcomes unresolved and allow a new attempt only once the earlier attempt is certified closed (a terminal result or certified no-effect evidence) or, in Permission to Retry and Evidence-Guided Recovery, under a still-valid deduplication key; Zhu and Wang keep uncertain effects charged until authenticated settlement, a fenced authoritative no-effect proof or permanent retirement [20]. Trofimov and Novikov catalogue the resulting anomalies [21].

Position. Our policies, references and fresh-key ablation correspond to conditions in these works, and our core results replicate theirs. What we add is narrow: among the works we reviewed, we found no agent-recovery evaluation that uses inbound webhook receipts (delayed, duplicated, reordered, lost or misleading) as evidence for retry and reconciliation together with stale negative status reads. The search was not exhaustive. Several of these works evaluate language-model agents; we evaluate none.

3 Model

Entities and time. A *logical operation* has a stable key `op_key` and an immutable payload; `payload_hash` is its SHA-256. Each dispatch creates an *attempt* `op_key#a`k, for $k = 1, 2, 3$. The *provider* accepts or rejects attempts, executes them, keeps records, answers status reads and emits webhooks, which a scripted queue delivers on time, late, twice, out of order or never. A private *effect ledger*, read only by the scorer, records every committed effect; effect identifiers are opaque (the first 12 hex characters of an HMAC). Time runs in integer ticks from the first dispatch at $t = 0$. Within a tick, provider events run first, then deliveries to the client, then client timers; ties are broken by a global scheduling counter. A status read issued at t is answered at $t + 1$ from the records with record time at most $t - L$, where L is the scenario's status lag (L = never: nothing is visible). An unscripted attempt is *nominal*: it arrives 1 tick after dispatch, commits 1 tick after arrival, and its `ok` response arrives 1 tick after the commit and its `committed` receipt 2 ticks after emission. For every policy and run, the client timeout is $T = 5$ ticks after dispatch; the budget is 3 dispatches per operation; the horizon is $H = 60$ ticks after the operation starts, and the belief at H is final; the scoring cutoff is $F = 200$; C, D and D+I poll status every 5 ticks; the contract-2 deduplication window is 1000 ticks, longer than the experiment; and Bw waits $\tau = 15$ ticks.

Provider contracts. **Contract 1** has no deduplication: every accepted attempt executes. **Contract 2** deduplicates on `op_key` and requires an identical `payload_hash`. Its record is created when the provider *accepts* the first attempt, at arrival rather than at commit. A later attempt with the same key creates no effect: it receives `in_progress` while the record is pending, and an `ok` replaying the original effect once it is committed. An aborted record is released, and the new attempt executes. A different payload receives `conflict`. A request lost before arrival creates no record, and window expiry is not modelled.

Observations and authority. A policy sees only the observations in Table 1, never the ledger, the script or future events. Every terminal negative is a provider guarantee that the attempt can never commit later; S15 is the only, labelled, violation. A definitive negative for an operation needs a terminal negative for every dispatched attempt. A request lost before arrival leaves no record and no terminal negative, so nothing can prove that it will not commit. Webhooks carry a sequence number, increasing per key in emission order, and an event identifier that a duplicated delivery repeats; they reach every operation and are correlated by the policy.

Table 1: What a policy observes, grouped by what each observation proves.

Authority	Observations
positive, if correlated	response `ok`; status `found`; webhook `committed` (each with attempt, effect identifier and payload hash)
terminal negative for that attempt	response `rejected` (refused before execution); under contract 2, response `in_progress` (says nothing about the earlier attempt) and `conflict` (a client bug); webhook `aborted`; each attempt listed in `terminal_attempts` of a status `not_found`
none	local timeout; response `error` (5xx-style: may or may not have executed); `not_found` itself (the replica may be stale, or the attempt may still commit); webhooks `processing` and `deferred` ("temporary failure; the provider will retry internally")

Correlation and conflict rules. C, D and D+I credit positive evidence only if its key and payload hash match and its attempt is one the client dispatched. Once such evidence is seen the operation is complete, and nothing reverts it. A terminal attempt stays terminal. A delivery with an event identifier already seen is ignored, and a webhook whose sequence number is not above the highest applied cannot regress the state. A positive and a terminal negative for the same attempt are a conflict: the commit wins, and the conflict is counted.

Policies. Table 2 gives the rules. A and B are simple comparison policies, written out in full; they act on the operation as a whole ("last terminal-looking event wins") and map any failure-named receipt, `deferred` or `aborted`, to a failure signal. C and D are a matched pair: C is D without retries. Every client policy uses one key and payload for every dispatch, a budget of 3 and the horizon H. The references run on the simulator side and read hidden state: ORACLE-S sees the ledger but not attempts in flight, so it is a state probe, not a ceiling; ORACLE-O knows whether a dispatched attempt will commit before F.

4 Scenarios and method

Core matrix and grid. Table 3 lists the 16 scenarios. Only the first attempt is scripted; every retry is nominal (Section 5.7 relaxes this). S13–S15 were added by the pre-freeze design update so that receipts could mislead. In the twins S08 and S09, every client policy sees byte-identical observations under contract 1, yet only S08 has an effect. S07 adds duplicated and out-of-order delivery to S04 and gives every policy the same outcome as S04; S11 checks that operations with the same payload but different keys are kept apart. The secondary grid explores timing boundaries; it is not a sample. G1 (lost ack: commit at $t = 2$, response lost) crosses $L \in \{0, 1, 2, 4, 8, 16, 32, 64, \text{never}\}$ with receipt delays $W \in \{1, 2, 4, 8, 16, 32, 64, \text{lost}\}$: 72 schedules. G2 (pending) crosses commit delays $E \in \{1, 3, 6, 12, 24, 48, 96\}$ with $L \in \{0, 1, 4, 16, 64, \text{never}\}$, response lost, receipt delay 2: 42

Table 2: Client policies, references and the ablation. A failure signal is a timeout, an `error`, `in_progress` or `rejected` response, or a failure-named receipt.

Policy	Rule
A: immediate retry	On a timeout or an `error`, `in_progress` or `rejected` response: retry at once if budget remains. On `ok` or a `committed` receipt (key match only): complete. On an `aborted` or `deferred` receipt: failed, then retry if budget remains. Once complete, request-path failure signals (and, for B and Bw, replies to their status reads) are ignored (revisions v1.1 and v1.1.1); receipts are not.
B: status check once	On every failure signal: one status read. `found`: complete; anything else, including `not_found` with terminal attempts: retry if budget remains. Otherwise as A.
Bw: wait-then-verify	As B, but the read is issued $\tau = 15$ ticks after the failure signal. Further signals during the wait are absorbed; no read is issued if already complete; the belief stays unresolved while waiting.
C: defer	Retries only on a synchronous `rejected` that arrives before any timeout; otherwise never dispatches again. From the first timeout or `error`, polls status every 5 ticks and applies the rules above. Final belief: complete on correlated positive evidence; failed if every attempt is terminal; otherwise unresolved.
D: reconcile	As C, plus a retry when every dispatched attempt has a terminal negative, no positive evidence exists and budget remains. Ambiguous evidence (a timeout, `error`, `not_found`, `processing` or `deferred`) never triggers a retry.
D+I (optional)	As D, plus, under contract 2 only, a resend with the same key after an ambiguous signal and one status poll that does not resolve it; `in_progress` means keep waiting. Identical to D under contract 1.
ORACLE-S (reference)	Acts at A's decision points and every 5 ticks: complete if the ledger holds an effect, otherwise retry if budget remains.
ORACLE-O (reference)	Complete if an effect exists; wait if some dispatched attempt will commit before F; otherwise retry if budget remains.
A-fk (ablation)	A with a fresh key on every attempt, run under contract 2 only.

schedules. G3 loses the request, for the same 6 lags. All run under both contracts and every policy (A-fk under contract 2 only): 272 core cells with 289 operations, and 2040 grid cells.

Measures. At F, an operation is a *duplicate* with more than one effect (extra effects are also counted), *exactly once* with one, and *zero* with none. The belief at H is compared with the ledger: a *false complete* is a complete belief with no effect at H; a *false failure* is a failed belief with an effect by F; *unresolved* operations are listed with their true effects. A duplicate is *contract-caused* if every committed retry was sent only after authoritative negatives covered every earlier attempt, and *policy-caused* otherwise. Adapting RetryBench's retry-permission score [15], a retry without such cover is *unjustified*, duplicate or not. RetryBench also admits a same-key retry under an enforced deduplication guarantee; we give that credit only to D+I's contract-2 resends, reported as contract-justified, so A's, B's and Bw's contract-2 counts are stricter than RetryBench's rule. The references' retries are not scored. *Time to confirmed completion* runs from the start to the first complete belief, over operations certified complete at H and exactly once; that count is its coverage, always reported.

Validation. Every run executes 61 checks and fails if any fails: 78 policy-free scripted runs that reproduce each fixture's effect history under both contracts; contract basics (a committed timeout followed by a resend gives 2 effects under contract 1 and 1 under contract 2); crafted observation sequences for the rules above; hidden-state isolation (deep-copied observations, a policy module that imports nothing, byte-identical twins); parity of budget and horizon; and 25 unit, property and mutation tests. Mutants that must be caught include D treating `not_found` as authoritative, D

Table 3: The core matrix. Times are for the first attempt, a1, dispatched at $t = 0$. "Receipt" is the `committed` webhook. Truth: effects if no retry happens.

ID	Family	Script for a1	Truth
S00	control	nominal; $L = 1$	1 at $t = 2$
S01	authoritative negative	rejected at arrival; `rejected` response at $t = 2$; $L = 1$	0
S02	authoritative negative	accepted at $t = 1$, aborted at $t = 3$; response and receipts lost; $L = 1$	0
S03	lost ack	commit at $t = 2$; response lost; receipt at $t = 8$; $L = 1$	1
S04	lost ack	commit at $t = 2$; response lost; receipt at $t = 20$; $L = 30$	1
S05	pending	commit at $t = 15$; late `ok` at $t = 16$; receipt at $t = 17$; $L = 1$	1 at $t = 15$
S06	receipt anomaly	aborted at $t = 3$; response lost; `aborted` receipt delayed to $t = 30$; $L = 1$	0
S07	receipt anomaly	commit at $t = 2$; response lost; receipt delivered at $t = 8$ and again at $t = 9$; `processing` delivered at $t = 12$; $L = 30$	1
S08	no confirmation	commit at $t = 2$; response and receipts lost; $L =$ never	1
S09	no confirmation	request lost before arrival; $L =$ never	0
S10	pending	commit at $t = 90$, after H; response lost; receipt at $t = 92$; $L = 1$	0 at H, 1 at F
S11	distinct operations	op-1: commit at $t = 2$, response and receipts lost; op-2: starts at $t = 1$, nominal, same payload, different key; $L =$ never	1 each
S12	lost ack (error)	commit at $t = 2$; `error` response at $t = 3$; receipt at $t = 10$; $L = 1$	1
S13	webhook harm	`deferred` delivered at $t = 3$; the provider retries internally and commits at $t = 8$; receipt at $t = 10$; response lost; $L = 1$	1 at $t = 8$
S14	webhook harm	commit at $t = 3$; `ok` at $t = 4$; receipt at $t = 6$; a `deferred` emitted at $t = 2$ arrives at $t = 12$; $L = 1$	1 at $t = 3$
S15	webhook harm (contract violation)	`aborted` receipt at $t = 4$; status lists a1 as terminal from $t = 4$; the provider commits a1 anyway at $t = 12$, receipt at $t = 14$; response lost; $L = 1$	1 at $t = 12$

ignoring attempt correlation, and a counting sequence number that leaks hidden state. A separate command compares every output byte across two runs.

4.1 Pre-specification, freeze and revisions

Pre-specification holds for part of the study and is self-attested throughout (Table 4). The original protocol, with predictions P01–P19 derived by hand from the rules, was committed 22 minutes before any code, and those predictions are unchanged in later commits. The design update, its predictions U01–U10 and T01, and all 22 implementation clarifications first appear in the same commit as the complete implementation, so git cannot show that the design-update predictions preceded the code that satisfies them. Before the freeze, D ran on S04 and S06 for the mutation tests, every client policy ran on S08 and S09 for the isolation check, and smoke runs reused S15's script; the one prediction mismatch was noticed by hand before the freeze and left unchanged. One clarification, that Bw skips its read when already complete, was chosen because the study lead's prediction for S04 required it. The freeze is self-attested: the repository has no remote and no signed tag, the freeze file can be regenerated, and the whole recorded history spans about 67 minutes.

The post-freeze revisions v1.1 and v1.1.1 were made after results were visible, and both helped the baselines: contract-1 grid duplicates fell from 114 to 90 for A, and from 86 to 70 and then 60 for B, while no core cell and nothing for C or D changed. They also show that the baseline rules were not settled at the freeze: the same "stop once complete" principle was applied to Bw before the freeze and to A's and B's request path after it, but never to their handling of receipts, which

still produces A's duplicates in S06 and S14. Revisions v1.2 to v1.2.2 are post-review fixes and exploratory analyses. v1.2 changed no core outcome; it restricted time to confirmation to certified, exactly-once operations and relabelled ORACLE-S. v1.2.1 and v1.2.2 changed only labels in the exploratory escalation output. The v1.1.1 core tables remain the pre-specified record, kept byte for byte in the repository. Sections marked "exploratory" were added after an independent review and were not pre-specified.

Table 4: Freeze and revision history. Grid duplicates are contract-1 counts over the 120 grid schedules.

Label	Commits	What changed	Effect on reported numbers
protocol	`8dc59af`	original design and predictions P01–P19	committed 22 minutes before any code
v1.0	`2ab9c6f`, `3f6cb9c`, `924aef2`	design update (S13–S15, `deferred`, Bw, references, A-fk, new measures), predictions U01–U10 and T01, clarifications C1–C22, implementation, freeze, results	first results
v1.1	`2b2d5e0`, `47289f8`, `622e1f1`	A, B and Bw ignore request-path failure signals once complete	no core cell; grid duplicates A 114→90, B 86→70
v1.1.1	`6ae14a9`, `7da9ec8`, `b95a667`	they also ignore replies to their own status reads once complete	no core cell; grid duplicates B 70→60
v1.2	`9d4021c`, `bc1fea0`, `0b01478`	post-review fixes; exploratory analyses; tooling	no core outcome; time to confirmation and labels
v1.2.1, v1.2.2	`4ffb5ce` to `d11234a`	labels in the exploratory escalation output	exploratory labels only

5 Results

5.1 Core matrix and grid

Table 5 gives the outcome of each of the 17 operations for every policy and contract; Table 6 totals them and the 120 grid schedules. The totals are counts over authored cases, and the mix of cases decides them.

Replicated results. Without deduplication, immediate retry (A) duplicates 12 of 17 operations: every lost acknowledgement and pending commit, the violation S15, and S06 and S14, where a late failure-named receipt overturns a confirmed completion. A single immediate status read (B) duplicates 8: where its read is stale (S04, S07, S08, S11/op-1), where the attempt is still pending (S05, S10, S13), and in S15. In the grid, A duplicates 90 of 120 schedules and B 60. With a stable key, the only duplicate of A and B is S15. A fresh key per attempt restores them: A-fk duplicates the same 12 operations and 90 schedules as A under contract 1. On the twins, A, B and Bw complete S09 but duplicate S08, and C and D leave both unresolved; no policy is both duplicate-free and complete on the pair.

What D's absence of policy-caused duplicates means. D retries only when every dispatched attempt has a terminal negative, which the provider guarantees will never commit. That D causes no duplicate therefore follows from its rules; only a broken guarantee (S15) can contradict it, and D's 240 grid cells add no information about its safety. Its measurable property is its cost: under contract 1 it leaves 4 of 17 core operations and 16 of 120 grid schedules unresolved at H (S08, S09, S10 and S11/op-1; Section 5.8). C has the same unresolved set, and where an authoritative

Table 5: Outcome of every core operation: effects at F, then the belief at H (C complete, F failed, U unresolved). Bold: duplicate. †: contract-caused duplicate (every committed retry was justified). 0F is a correct failure, 1F a false failure. OS and OO are the references ORACLE-S (a state probe, not a ceiling) and ORACLE-O.

	Contract 1: no deduplication								Contract 2: deduplication								
Operation	A	B	Bw	C	D	D+I	OS	OO	A	B	Bw	C	D	D+I	A-fk	OS	OO
S00	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C
S01	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C
S02	1C	1C	1C	0F	1C	1C	1C	1C	1C	1C	1C	0F	1C	1C	1C	1C	1C
S03	**2C**	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	**2C**	1C	1C
S04	**2C**	**2C**	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	**2C**	1C	1C
S05	**2C**	**2C**	1C	1C	1C	1C	**2C**	1C	1C	1C	1C	1C	1C	1C	**2C**	1C	1C
S06	**2C**	1C	1C	0F	1C	1C	1C	1C	1C	1C	1C	0F	1C	1C	**2C**	1C	1C
S07	**2C**	**2C**	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	**2C**	1C	1C
S08	**2C**	**2C**	**2C**	1U	1U	1U	1C	1C	1C	1C	1C	1U	1U	1C	**2C**	1C	1C
S09	1C	1C	1C	0U	0U	0U	1C	1C	1C	1C	1C	0U	0U	1C	1C	1C	1C
S10	**2C**	**2C**	**2C**	1U	1U	1U	**2C**	1U	1F	1F	1F	1U	1U	1U	**2C**	1F	1U
S11/op-1	**2C**	**2C**	**2C**	1U	1U	1U	1C	1C	1C	1C	1C	1U	1U	1C	**2C**	1C	1C
S11/op-2	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C
S12	**2C**	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	**2C**	1C	1C
S13	**3C**	**3C**	1C	1C	1C	1C	**2C**	1C	1C	1C	1C	1C	1C	1C	**3C**	1C	1C
S14	**2C**	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	1C	**2C**	1C	1C
S15	**3C**	**3C**	1C	1C	**2C**†	**2C**†	**3C**	1C	**2C**†	**2C**†	1C	1C	**2C**†	**2C**†	**3C**	**2C**†	1C

Table 6: Core matrix totals: counts of the 17 authored operations per row, not rates. Cause: contract-caused/policy-caused duplicates. Unjust.: unjustified retries (D+I's contract-justified resends in brackets; not scored for references). TTC: time to confirmed completion in ticks, median/max, over operations certified complete at H and exactly once; the coverage, in brackets, counts those operations. Medians over different subsets are not comparable. Grid: duplicates/unresolved/false failures over the 120 grid schedules (A-fk: contract 2 only). False complete is 0 everywhere. The contract-2 S15 duplicates of A, B and ORACLE-S count as contract-caused although each also sent an unjustified retry, which was replayed without effect.

Policy	Dup.	Extra	Cause	Once	Zero	Unres.	False fail.	Unjust.	Confl.	TTC (cov.)	Grid
Contract 1: no deduplication											
A	12	14	0/12	5	0	0	0	16	0	5/8 (5)	90/0/0
B	8	10	0/8	9	0	0	0	10	0	6/9 (9)	60/0/0
Bw	3	3	0/3	14	0	0	0	4	0	12/24 (14)	27/0/0
C	0	0	0/0	14	3	4	0	0	1	6/20 (11)	0/16/0
D	1	1	1/0	15	1	4	0	0	1	7/20 (12)	0/16/0
D+I	1	1	1/0	15	1	4	0	0	1	7/20 (12)	0/16/0
ORACLE-S	4	5	0/4	13	0	0	0	n/a	0	5/8 (13)	30/0/0
ORACLE-O	0	0	0/0	17	0	1	0	n/a	0	5/15 (16)	0/6/0
Contract 2: deduplication on a stable key											
A	1	1	1/0	16	0	0	1	18	0	7/16 (15)	0/0/6
B	1	1	1/0	16	0	0	1	12	0	8/16 (15)	0/0/6
Bw	0	0	0/0	17	0	0	1	5	0	15/24 (16)	0/0/6
C	0	0	0/0	14	3	4	0	0	1	6/20 (11)	0/16/0
D	1	1	1/0	15	1	4	0	0	1	7/20 (12)	0/16/0
D+I	1	1	1/0	16	0	1	0	0 (+9)	1	8/16 (15)	0/6/0
A-fk	12	14	0/12	5	0	0	0	16	0	5/8 (5)	90/0/0
ORACLE-S	1	1	1/0	16	0	0	1	n/a	0	5/15 (15)	0/0/6
ORACLE-O	0	0	0/0	17	0	1	0	n/a	0	5/15 (16)	0/6/0

negative arrives (S02, S06) it ends with zero effects and a correct failed belief, while D retries and completes. Under contract 2, D+I's same-key resend completes S08, S09 and S11/op-1, leaving only S10, which commits after the horizon.

Contract 2 does not make the naive policies safe. Under contract 2, A, B and Bw end failed in S10 although the effect commits later, because they read `in_progress` as a failure; and A, B, D, D+I and ORACLE-S duplicate in S15, because the abort released the deduplication record. Both outcomes depend on a rule choice (Section 5.5).

Latency and the references. Under contract 1, A's median time to confirmed completion of 5 ticks covers 5 operations and D's median of 7 covers 12; medians over different subsets do not show that either confirms faster. ORACLE-S is not a ceiling: blind to attempts in flight, it duplicates 4 core operations (S05, S10, S13, S15) and 30 grid schedules under contract 1 and ends falsely failed in S10 under contract 2. ORACLE-O has no duplicate and leaves only S10 unresolved.

5.2 Worked traces

Table 7 follows policy D under contract 1 through S04, S02 and S06, from the run's traces.

Table 7: Worked traces for policy D under contract 1. Top: S04, a `not_found` read while an effect exists. Bottom: S02 and S06, where an authoritative negative enables a useful retry; they differ only in the fate of a1's `aborted` receipt.

t	Provider knows	D sees and does	Scorer records
S04			
0	–	dispatches a1	–
1	a1 arrives and is accepted	–	–
2	a1 commits; its receipt is scheduled for $t = 20$; the `ok` response is lost	–	effect of a1 at $t = 2$
5	replica (lag 30) shows no record	timeout (ambiguous); status read	–
6	–	`not_found`, no terminal attempts	–
10, 15	–	polls; `not_found` again at 11 and 16	–
20	–	`committed` receipt for a1: complete	–
S02 and S06			
0	–	dispatches a1	–
1	a1 arrives and is accepted	–	–
3	a1 aborts; response lost; `aborted` receipt lost (S02) or delayed to $t = 30$ (S06)	–	–
5	replica (as of $t = 4$) lists a1 as aborted	timeout (ambiguous); status read	–
6	–	`not_found` with a1 terminal: every attempt is terminal, so D dispatches a2	–
8	a2 commits	–	effect of a2 at $t = 8$
9	–	`ok` for a2: complete	–
30	–	S06 only: stale `aborted` receipt for a1, already terminal; nothing changes	–

In S04, A retries at $t = 5$, and B, believing the stale `not_found`, at $t = 6$; both duplicate, with second effects at $t = 7$ and $t = 8$. Bw's verify timer and the receipt both fall at $t = 20$; deliveries run before timers, so Bw completes without reading. Under contract 2, A's and B's retries receive a replay of the original effect. In S02 and S06, C never retries after the timeout and ends with zero effects and a correct failed belief. In S06, A retries at the timeout ($t = 5$, effect at $t = 7$) and again

when the stale `aborted` receipt arrives at $t = 30$, because its last terminal-looking event wins; a second effect commits at $t = 32$.

5.3 Webhook evidence (exploratory ablations)

The argument. The decision-relevant state of C, D and D+I is the set S of attempts sent, the set P with correlated positive evidence, the set N with an authoritative terminal negative, and a flag that only starts polling. The decision is a function of (S, P, N) and the remaining budget: complete if $P \neq \emptyset$; otherwise retry if $S \subseteq N$ and budget remains (D); otherwise failed if $S \subseteq N$; otherwise unresolved. (i) Each observation only adds an attempt to P or N; nothing removes one, since a commit cannot be undone and a terminal negative is a guarantee. (ii) Set union is commutative, associative and idempotent, so at any tick the decision depends only on the set of facts delivered so far, not on their order or multiplicity. (iii) Sequence numbers and event identifiers can only reorder or drop repeats of facts already in the sets, so they cannot change a decision. The argument has limits. *When* a fact arrives still decides when the reconciler can act and, past the horizon, whether it can. It needs monotone facts; reversible provider states (scheduled, cancelled, rescheduled) would make order matter, and none is modelled. It needs the guarantee (Section 5.4). And keying by attempt matters: the mutant that ignores attempt correlation is caught. A, B and Bw are not monotone: one belief, overwritten by later events, so a stale failure-named receipt can flip a correct completion.

Table 8: Ablations of C and D (exploratory). Changed: operations or schedules whose effects, belief, dispatches, time to first completion or conflicts differ from the unablated policy, over both contracts (34 core operations, 240 grid schedules). Core d/o/u/f (duplicates, exactly once, unresolved, false failures, of 17) and Grid unres. are contract-1 counts.

Policy	Ablation	Core changed	Grid changed	Core d/o/u/f	Grid unres.
D	ignore provider_seq	0	0	1/15/4/0	16
D	no event-ID de-duplication	0	0	1/15/4/0	16
D	no payload check	0	0	1/15/4/0	16
D	ignore every webhook	8	136	1/15/4/0	41
D	no status reads	8	58	1/14/5/0	30
C	ignore provider_seq	0	0	0/14/4/0	16
C	no event-ID de-duplication	0	0	0/14/4/0	16
C	no payload check	0	0	0/14/4/0	16
C	ignore every webhook	8	136	0/14/4/1	41
C	no status reads	6	58	0/14/5/0	30

The ablations. Switching off sequence numbers, event-ID de-duplication or the payload check changes none of the 34 core operations and none of the 240 grid schedules, for C or D (Table 8; no frozen scenario reuses a key with another payload). The study thus does not test sequence-number or duplicate-delivery handling where they could matter; it shows that for this kind of reconciler they cannot. Receipts still matter as a second channel. Ignoring every webhook changes no outcome of D in the core matrix but delays confirmation in four scenarios (S04 from 20 to 36 ticks, S07 from 8 to 36, S13 from 10 to 11, S15 from 7 to 9), and raises the unresolved grid schedules from 16 to 41 of 120, because receipts resolve cases in which the replica lags past the horizon. For C it also leaves a false failure in S15. Ignoring status instead raises the grid count to 30 and, for D, loses S02, whose only authoritative negative is the status listing.

Name-based handling of failure receipts. Mapping `deferred` and `aborted` receipts to failure by name harms the immediate-retry client in 2 scenarios. In S14 a `deferred` receipt arrives at $t = 12$, after A confirmed completion at $t = 4$; A's belief flips to failed and a second effect commits at $t = 14$. In S06 the stale `aborted` receipt does the same. With `deferred` as progress and failure receipts ignored once complete, A's contract-1 duplicates fall from 12 to 10, and B's and Bw's counts do not change (Table 9). S13 is not a webhook-harm case for B, which also duplicates there through its timeout and a `not_found` on a pending attempt, as in S05. The webhook-harm family shows receipt-specific harm for A in 2 scenarios, and none for B or Bw.

5.4 A provider that breaks its guarantee

In S15 the provider reports a1 as aborted, by receipt at $t = 4$ and in the status listing, and then commits a1 at $t = 12$. D retries at $t = 4$; a2 commits at $t = 6$ and D confirms completion at $t = 7$. At $t = 14$ the `committed` receipt for a1 arrives, D records a conflict (the commit wins), and the ledger holds 2 effects. The duplicate is contract-caused: the retry was justified by the provider's own guarantee. Under contract 2 the abort released the deduplication record, so A, B, D, D+I and ORACLE-S duplicate under both contracts; Bw (whose wait outlasts the receipt), C (which does not retry) and ORACLE-O do not. If an aborted key instead refuses later attempts, the contract-2 duplicates in S15 disappear for every policy, but D and D+I can no longer complete S02 and S06 under contract 2 (Table 9). Evidence-based retry is only as safe as the provider's terminal-negative guarantee, and release-on-abort deduplication does not protect against a violation.

5.5 Sensitivity to the rules (exploratory)

Each alternative rule raised in the independent review is a named option, applied to every policy, contract, scenario and schedule. Table 9 shows the counts that move.

What survives every variant. Under contract 1, A duplicates at least 10 of 17 core operations and 90 of 120 grid schedules in every row, and B at least 5 and 40. Under contract 2, no client policy other than A-fk duplicates a grid schedule or more than 1 core operation in any row, and A-fk reproduces A's contract-1 counts in every row. C never duplicates, and D at most once, contract-caused. C and D leave at least 3 of 17 core operations unresolved in every row: a longer horizon resolves S10, but nothing resolves the twins.

What moves. B's duplicates depend on whether status shows pending work, A's harm from receipts on naming, and Bw's count on timing (with $\tau = 5$ Bw duplicates even under contract 2, in S15). The contract-2 false failures in S10 come from reading `in_progress` as failure. "Contract 2 removes duplicates" depends on release-on-abort. Every head-to-head count between D and a baseline moves under at least one rule, so the baselines' counts are functions of the provider contract and the client's timing, not a ranking.

5.6 Where wait-then-verify duplicates (exploratory)

Bw is the fairest baseline, and its 3 contract-1 core duplicates rest on fine timing. In a unit-step lost-acknowledgement grid under contract 1, a1 commits at $t = 2$ with its response lost, the receipt arrives at $t = R$ for R from 3 to 70 or never, and L runs from 0 to 64. For every τ tested, the duplicating cells are exactly $L \geq L^*$, $R \geq R^*$ (or receipt lost), with $L^* = \tau + T - c + 1$ and $R^* = \tau + T + 2$ as derived from the timing rules, where $c = 2$ is the commit time (Table 10); with

Table 9: Sensitivity (exploratory). Each entry gives a policy, a contract (c1, c2), a count and its change from the frozen rules to the variant, for the 17 core operations and then the 120 grid schedules (core / grid). Counts are over authored cases, not rates.

Variant (named option)	What moves (core / grid)
`status_lists_pending`: status lists pending attempts; B and Bw re-read while one is pending	B c1 dup 8→5 / 60→40; Bw c1 dup 3→2 / 27→15; B c2 false fail 1→0 / 6→3; Bw c2 false fail 1→0 / 6→2
`webhooks_not_by_name`: `deferred` is progress; failure receipts ignored once complete	A c1 dup 12→10 / 90→90; B c1 dup 8→8 / 60→60; Bw c1 dup 3→3 / 27→27
`bw_without_C15`: Bw reads even if already complete	Bw c1 dup 3→3 / 27→27
`bw_without_C15_or_reply_guard`: also act on status replies after completion	B c1 dup 8→8 / 60→70; Bw c1 dup 3→5 / 27→44
`tau_5`: Bw waits $\tau = 5$	Bw c1 dup 3→7 / 27→40; Bw c2 dup 0→1 / 0→0
`tau_10`: Bw waits $\tau = 10$	Bw c1 dup 3→4 / 27→34
`tau_20`: Bw waits $\tau = 20$	Bw c1 dup 3→3 / 27→26; Bw c2 false fail 1→0 / 6→0
`tau_30`: Bw waits $\tau = 30$	Bw c1 dup 3→3 / 27→16; Bw c2 false fail 1→0 / 6→0
`in_progress_is_wait`: A, B, Bw wait on `in_progress`	A c2 false fail 1→0 / 6→0; B c2 false fail 1→0 / 6→0; Bw c2 false fail 1→0 / 6→0; A c2 unres 0→1 / 0→6
`abort_record=replay_failure`: an aborted key refuses later attempts	A c2 dup 1→0 / 0→0; D c2 dup 1→0 / 0→0; D c2 once 15→14 / 114→114; D+I c2 once 16→15 / 120→120
`retry_fault=copy_first`: retries repeat the first attempt's fault	A c1 false fail 0→4 / 0→24; B c1 false fail 0→3 / 0→14; Bw c1 unres 0→6 / 0→16; D c1 dup 1→1 / 0→0; D+I c2 unres 1→4 / 6→16
`horizon=120`: a longer horizon	D c1 unres 4→3 / 16→7; A c2 false fail 1→0 / 6→0

$T = 5$, $L \geq \tau + 4$ and $R \geq \tau + 7$. For the frozen $\tau = 15$, Bw duplicates exactly when $L \geq 19$ and the receipt arrives at $t \geq 22$. S04 (receipt at $t = 20$, $L = 30$) lies 2 ticks on the safe side, and G1's doubling steps jump over the boundary. In a second grid with the receipt lost and the commit at $t = 1 + E$, Bw ($\tau = 15$) duplicates exactly when $E + L \geq 20$. In general Bw is duplicate-free if and only if its verify read sees the commit, i.e. the commit time is at most $t_{\text{signal}} + \tau - L$; with commits bounded by dispatch $+ E_{\max}$ and the first signal at dispatch $+ T$, a sufficient condition is $\tau \geq E_{\max} + L_{\max} - T$.

Table 10: Where Bw duplicates in the unit-step lost-acknowledgement grid (exploratory; contract 1). L^*, R^*: smallest status lag and receipt time with a duplicate. Exact: the duplicating cells are exactly $\{L \geq L^*, R \geq R^*$ or lost$\}$.

τ	L^*	R^*	Exact	Predicted L^*	Predicted R^*
5	9	12	yes	9	12
10	14	17	yes	14	17
15	19	22	yes	19	22
20	24	27	yes	24	27
30	34	37	yes	34	37

5.7 Persistent faults (exploratory)

In the frozen matrix every retry is nominal, so no policy meets a second ambiguous timeout; Table 11 repeats attempt 1's fault on every retry. Under persistent lost acknowledgements A commits three times and ends failed (PF1, PF2), and so does B when status never shows the commit (PF2): false failures with triple effects, impossible in the frozen matrix. Bw also commits three times in PF2 but ends unresolved. Under contract 2 the key prevents the extra effects, but A (PF1, PF2) and

Table 11: Persistent faults (exploratory): the same fault on all three attempts. Cells as in Table 5; bold: duplicate.

	Contract 1						Contract 2					
Scenario	A	B	Bw	C	D	D+I	A	B	Bw	C	D	D+I
PF1 lost ack, $L = 1$	**3F**	1C	1C	1C	1C	1C	1F	1C	1C	1C	1C	1C
PF2 lost ack, $L =$ never	**3F**	**3F**	**3U**	1U	1U	1U	1F	1F	1U	1U	1U	1U
PF3 lost request, $L = 1$	0F	0F	0U	0U	0U	0U	0F	0F	0U	0U	0U	0U
PF4 abort, $L = 1$	0F	0F	0U	0F	0F	0F	0F	0F	0U	0F	0F	0U

B (PF2) still end falsely failed. C and D never duplicate and stay unresolved where nothing is observable (PF2, PF3). Under contract 2, D+I ends unresolved in PF2, PF3 and PF4: its resends are lost (PF3) or deduplicated with their replies lost, leaving no provider record that could close them (PF2, PF4).

5.8 Unresolved operations and what would resolve them (exploratory)

For every operation unresolved at H, the suite reports the attempts the client could not close and the evidence that would close them; in the core matrix the open attempt is always a1. C and D leave S08, S09, S10 and S11/op-1 unresolved under both contracts, D+I the same four under contract 1 but only S10 under contract 2, and ORACLE-O only S10. S08 and S11/op-1 (a1 committed at $t = 2$) need a correlated positive that never comes; S10 (a1 commits at $t = 90$) needs a positive after the horizon, or a longer horizon; S09 (a1 never arrived) can be closed only by a guarantee this provider cannot give, a terminal negative for a request that never arrived or a bound on in-flight time, the bound LIMBO's impossibility result requires [16]. In the grid, C and D leave unresolved, under each contract, 4 G1 schedules lacking a positive, 6 G2 schedules committing after H and the 6 G3 schedules.

5.9 Prediction check

The protocol's predictions are hand derivations from deterministic rules, so a match shows only that the implementation agrees with the author's reading of the rules. All 20 rows of the original protocol match (16 from its prediction section, scoped to S00–S12, and 4 from its worked timeline), as do 16 of the 17 design-update rows and the 1 tie-note row. That 37 of 38 rows match is a consistency check, not a confirmation rate, and the rows are not independent. U04b matches vacuously, because Bw issues no read in S15. The mismatch, U09a, is a prediction error noticed before the freeze and left unchanged: it omitted D+I, which inherits D's justified retry, from the contract-2 duplicates in S15.

6 Using the suite on your own provider

The questions that changed results in Section 5.5 are the ones a practitioner has to answer about a provider. Table 12 lists them with the scenario field or option that encodes each answer; a scenario is a JSON file in the same format as the frozen fixtures. Table 13 condenses the suite's decision table; it records what the suite measured or argued, not a validated recommendation for real providers.

The suite is driven by one script:

```
python3 run_sim.py
python3 run_sim.py --check-reproducible
```

Table 12: Provider-contract checklist. Each answer selects a scenario field or a named option.

	Question	Field or option
Q1	Is there an idempotency key, and does it require an identical payload?	`--contract 1` or `--contract 2`
Q2	Is the key's record created on acceptance or only on commit?	`dedup_record_at`
Q3	After a failed execution, is the key released or is the failure replayed?	`abort_record`
Q4	Does the status API show accepted but unfinished work?	`status_lists_pending`
Q5	Is there a bound on how stale a status read can be?	`status_lag`: an integer or `"never"`
Q6	Which receipts are terminal guarantees and which are progress? Can the provider break a guarantee?	`webhooks`, `deferred_after`, `violation_commit_after`; `deferred_is_progress`
Q7	Is in-flight time (arrival plus execution) bounded?	`arrive`, `commit_after`; `horizon`
Q8	Is the deduplication window longer than the longest retry sequence?	`dedup_window` (not exercised here)
Q9	Do faults persist across retries?	`retry_fault=copy_first`
Q10	Can the client remember its attempt identifiers across a restart?	not modelled

Table 13: Condensed decision table. "Basis" points to the measurement or argument behind each row.

If the provider ...	Use	Basis
honours a stable key (record at acceptance, `in_progress` while pending, result replayed)	D+I: reconcile, then resend with the same key after one unresolving poll; treat `in_progress` as wait; never rotate keys	Table 6, contract 2; A-fk
has no key, but bounds status staleness ($L_{\max}$) and in-flight time ($E_{\max}$)	Bw with $\tau \geq E_{\max} + L_{\max} - T$	Section 5.6
has no key, and its status API lists pending attempts	check-then-retry that waits while an attempt is pending	Table 9
has no key and no bounds, but gives authoritative terminal negatives	D, and escalate unresolved operations	safe by construction; cost: Section 5.8
sends webhooks	treat only terminal-guarantee events as failures, per attempt; never let a late failure overwrite a completion; use receipts as evidence	Section 5.3
replies `in_progress` to a duplicate request	treat it as wait, not failure	Table 9
may violate its own terminal guarantees	no evidence-based retry is safe; prefer keys that replay failures, or do not retry	Section 5.4
may take longer than the agent's horizon	lengthen the horizon, or escalate	Table 9
(any)	avoid immediate retry without a key, and a single immediate status read	Section 5.1

```
python3 run_sim.py --show 1 D S04
python3 run_sim.py explore --scenario my.json --contract 2 --policy D+I
python3 run_sim.py explore --scenario my.json --policy mine.py:MyPolicy
python3 run_sim.py replay --observations my.jsonl --policy D
```

The first command runs the frozen study, all validation and every output; the second compares every byte across two runs. `--show` prints the trace of any cell. `explore` runs a built-in or user-supplied policy (any class with `observe(obs)` returning actions, and `belief()`) on any scenario, without the freeze gate. `replay` feeds a recorded observation log, such as a converted production client log, to a policy and prints the actions it would have taken. Simulator traces are JSON lines that include hidden provider state; observations, the only input a policy receives, are a separate,

documented vocabulary.

7 Limitations

- **Authored cases.** The 16 core scenarios, 120 grid schedules and 4 persistent-fault scenarios are hand-written, and their mix decides the counts. D's caution, for example, costs completion only where a request never arrived, 1 of 17 core operations and 6 of 120 grid schedules; how common such cases are in real traffic is unknown.
- **Simple, scripted baselines.** A, B and Bw act on every failure signal, even while a later attempt is in flight, and several of their rules are choices (Section 5.5).
- **Not modelled:** client restarts that lose attempt identifiers; reversible provider states; authorisation changes during recovery; window expiry; more than one action type or provider; delayed arrivals in the frozen scenarios.
- **No language-model agents.** The results describe scripted policies, not models, prompts or harnesses.
- **Self-attested freeze.** Only P01–P19 verifiably preceded the code, and two revisions were made after results were visible (Section 4.1).
- **A residual leak.** A delivered receipt's sequence number reveals the tick at which it was emitted.
- **Literature search.** The search for related work was not exhaustive.

8 Conclusion

In a model small enough to check by hand, the suite reproduces the known results on retry after an ambiguous timeout and adds an account of webhook receipts as evidence: for a monotone, attempt-keyed reconciler their delay, duplication and reordering are harmless, their timing decides latency and resolution, and reading failure-named receipts by name is what harms an immediate-retry client. Evidence-based retry inherits every weakness of the provider's terminal-negative guarantee. Because most head-to-head counts move under equally reasonable rules, the useful outputs are the mechanisms, the measured boundaries and the questions to ask of a provider.

Code and data availability

The simulator (Python standard library only, no network access, no randomness), fixtures, protocol with its full revision log, freeze history, frozen baselines, traces and outputs are in a public git repository, `https://github.com/Anishpras/ambiguous-retry-recovery`. The frozen run is v1.2.2 (results commit `d11234a`). `python3 run_sim.py` reproduces every result in this report and exits 0 only if every check passes; it takes about 15 seconds (CPython 3.13.7, macOS, arm64). Every number in this report is listed with its source in `paper/NUMBERS.md` and checked against the result files by `paper/check_numbers.py`.

Acknowledgements and AI-assistance statement

The simulator code, analysis and manuscript were prepared with substantial assistance from Claude (Anthropic), an AI assistant, working under the author's direction. Separate AI agents carried out the literature search, the implementation and the independent senior review. The author is responsible for the content. The work used no paid API, cloud compute or external funding.

References

[1] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. In *Proc. Fifth ACM Symposium on Operating Systems Principles (SOSP '75)*, pp. 67–74, 1975. https://doi.org/10.1145/800213.806523

[2] J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, LNCS 60, pp. 393–481, Springer, 1978. https://doi.org/10.1007/3-540-08755-9_9

[3] Jepsen. jepsen-io/history: Support library for working with Jepsen histories. GitHub repository README. Accessed 13 Nov. 2026. https://github.com/jepsen-io/history

[4] P. Helland. Idempotence is not a medical condition. *ACM Queue*, 10(4):30–46, 2012. https://doi.org/10.1145/2181796.2187821

[5] Stripe. Idempotent requests. Stripe API Reference. Accessed 13 Nov. 2026. https://docs.stripe.com/api/idempotent_requests

[6] M. Featonby. Making retries safe with idempotent APIs. Amazon Builders' Library, 2021. https://builder.aws.com/content/3Ev0BENTyBr0XxzRk5FDZzgNYos/making-retries-safe-with-idempotent-apis

[7] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009. https://doi.org/10.1145/1435417.1435432

[8] Stripe. Receive Stripe events in your webhook endpoint. Stripe Documentation. Accessed 13 Nov. 2026. https://docs.stripe.com/webhooks

[9] Shopify. About webhooks. Shopify Developer Documentation. Accessed 13 Nov. 2026. https://shopify.dev/docs/apps/build/webhooks

[10] GitHub. Best practices for using webhooks; Handling failed webhook deliveries. GitHub Docs. Accessed 13 Nov. 2026. https://docs.github.com/en/webhooks/using-webhooks/best-practices-for-using-webhooks and https://docs.github.com/en/webhooks/using-webhooks/handling-failed-webhook-deliveries

[11] D. Gupta. Two-path status verification for outbound enterprise messaging pipelines: Webhook and scheduled polling fallback architecture. arXiv:2607.15529 [cs.SE], 2026. https://arxiv.org/abs/2607.15529

[12] D. Sah, T. Sah, H. Jain, and T. Sah. UndoBench: Separating task competence from recovery capability in tool-using AI agents. arXiv:2610.05622v1 [cs.SE], 4 Oct. 2026. https://arxiv.org/abs/2610.05622

[13] I. K. Mansoor, A. Phadke, and P. Rana. Verified tool calls improve LLM agent reliability under non-atomic failures. arXiv:2608.02645v1 [cs.SE], 31 Jul. 2026. https://arxiv.org/abs/2608.02645

[14] S. Sun. Did it happen? Counterfactual evaluation of LLM agent recovery from ambiguous tool outcomes. Research Square preprint, 2026. https://doi.org/10.21203/rs.3.rs-10730245/v1

[15] Y. Liu, Y. Liu, and Q. Lu. RetryBench: Separating retry permission from recovery outcomes in GPT tool use. arXiv:2610.15884v4 [cs.AI], 4 Nov. 2026. https://arxiv.org/abs/2610.15884

[16] J. Li. Where does exactly-once live? Model, harness, and tool-contract effects on duplicate side effects in LLM agents. arXiv:2609.29095 [cs.LG], Sep. 2026. Introduces the LIMBO benchmark. https://arxiv.org/abs/2609.29095

[17] D. Marla and K. Bhatnagar. Permission to retry: A study of authorization and recovery in AI agent workflows. arXiv:2610.31916v1 [cs.AI], 22 Oct. 2026. https://arxiv.org/abs/2610.31916

[18] G. Zhu, X. Chen, Z. Wang, and C. Wang. Evidence-guided recovery for AI agents with ambiguous tool outcomes and partial execution. arXiv:2610.41453 [cs.AI], 30 Oct. 2026. https://arxiv.org/abs/2610.41453

[19] Y. Tong, L. Dai, and S. Guo. AID-Guard: Stateful authorization for delegated agent effects. arXiv:2608.21159 [cs.CR], 2026. https://arxiv.org/abs/2608.21159

[20] G. Zhu and C. Wang. Fault-tolerant budget conservation in distributed multi-agent delegation. arXiv:2610.00349 [cs.AI], 2026. https://arxiv.org/abs/2610.00349

[21] A. Trofimov and B. Novikov. When tool calls succeed but workflows fail: Anomalies at the agent–tool boundary. arXiv:2609.15397v1 [cs.AI], 14 Sep. 2026. https://arxiv.org/abs/2609.15397