American Journal of Smart Technology and Solutions (AJSTS)

ISSN: 2837-0295 (ONLINE)

VOLUME 5 ISSUE 2 (2026)



PUBLISHED BY
E-PALLI PUBLISHERS, DELAWARE, USA

American Journal of Smart Technology and Solutions (AJSTS)

Volume 5 Issue 2, Year 2026
ISSN: 2837-0295 (Online)
DOI: https://doi.org/10.54536/ajsts.v5i2.8833
https://journals.e-palli.com/home/index.php/ajsts

Safe Recovery of AI Agents in Enterprise Systems: Preventing Duplicate Actions and Preserving Human Corrections

Medhavee Upadhyaya[1*], Gaurav Chaulagain[1]

Article Information

Received: July 27, 2026

Accepted: October 19, 2026

Published: November 04, 2026

Keywords

AI agents, API contracts, Concurrency control, Fault recovery, Workflow reliability

ABSTRACT

Reliable use of AI agents in enterprise systems depends on what happens after an action fails or its outcome becomes uncertain. Retrying can duplicate an order, while reversing a workflow can invalidate a person's subsequent correction. The central challenge is deciding whether recovery is justified by the available evidence and still satisfies the business request. We investigate this challenge in a reproducible testbed connecting order, inventory and billing services with different API guarantees. Controlled failures and concurrent edits reveal two evaluation gaps: protecting individual records does not ensure consistency across related services, and preserving an edit does not ensure that the final workflow fulfils its instruction. A separate diagnostic identified nine apparent completions that left human instructions unsatisfied. A conventional recovery implementation with equivalent safeguards matched the evaluated controller in all 270 paired runs, locating the observed benefit in shared engineering protections rather than a distinct recovery algorithm. On the prospective test configurations, both completed 108 of 180 runs without scored violations and left 72 for manual resolution. These findings provide a practical basis for evaluating agent integrations: verify uncertain effects, check related records, and distinguish completed business goals from unresolved handoffs. The contribution is a reproducible evaluation of these recovery requirements within a synthetic enterprise workflow, with operational effectiveness remaining to be established.

INTRODUCTION

An AI agent creates a delivery order, but the service's response never arrives. Should the agent try again? If the first request succeeded, a retry can create a second order. If it failed, stopping leaves the delivery unfinished. An agent that instead cancels the workflow faces another problem: a person may have changed the records while recovery was in progress. The recovery action must account for what happened remotely and what people changed afterward. The uncertainty of remote execution is longstanding (Birrell & Nelson, 1984); adding a language model that chooses tool calls does not remove it (Yao *et al.*, 2023).

We study this problem in a workflow that reserves inventory, creates an order and issues an invoice. The services store their own records and cannot change all three as one all-or-nothing transaction. Cancellation must therefore use separate calls through their application programming interfaces (APIs). Each call may fail, succeed without returning a response, or encounter a newer record version. We ask what recovery can do with the guarantees those services actually provide.

The intended readers are researchers evaluating tool-using agents and engineers integrating agents with business systems. Reliable interaction with external systems is also a concern of NIST's AI Agent Standards Initiative (National Institute of Standards and Technology [NIST], 2026). Our testbed supports evaluation of that problem before deployment. Its provider profiles simulate different API capabilities; they are not measurements of deployed manufacturing or logistics systems.

The experiments examine established safeguards in combination. A recovery controller is the software that applies these rules when an operation fails or its outcome is uncertain. A request identity is an identifier that a service can use to recognize a repeated request and prevent duplicate execution. A conditional update can reject a change made against an outdated record. Checking related records can reveal that an otherwise valid cancellation would break the workflow. We test how these safeguards affect harmful outcomes, completed recovery and work left for manual resolution.

We examine three questions: whether a contract-aware recovery controller offers a measurable advantage over conventional recovery with equivalent safeguards; which safeguards change the observed outcomes; and whether the definition of success captures intervening human instructions. The study connects these questions through an executable workflow, recorded failures and comparisons made under shared service guarantees. Its contribution is empirical: it identifies the protections responsible for observed progress and exposes a gap between avoiding a harmful write and satisfying a person's latest instruction.

The primary comparison includes a conventional implementation built separately from the controller being

[1] San Francisco Bay University, Fremont, California, USA
[*] Corresponding author's e-mail: mupadhya200@student.sfbu.edu

evaluated. Their identical outcomes locate the measured protection in the shared safeguards. A second finding concerns scoring: a record may retain a human instruction without the final workflow fulfilling that instruction. Supporting tests isolate individual checks and examine the functions that connect recovery runtimes to external services. Together, these experiments provide inspectable evidence for evaluating recovery at the boundary between an agent and persistent business records.

LITERATURE REVIEW

Compensation and Concurrency Foundations

Compensation means using a new operation to counteract an earlier one, such as cancelling an order. It is an established response to failures in long-running transactions. Sagas introduced this approach (Garcia-Molina & Salem, 1987), and Korth *et al.* (1990) formalized compensation whose validity depends on the application and on what other transactions have already observed. Microsoft's guidance identifies a practical consequence: restoring old state can overwrite intervening changes (Microsoft, n.d.). Our experiments bring these questions into an agent-recovery testbed, where the controller must determine whether a reversal is still appropriate and whether it preserves later work.

Optimistic concurrency control checks whether another operation changed a record before accepting an update (Kung & Robinson, 1981). Linearizability requires overlapping operations on an object to behave as if each took effect at one point between its start and finish (Herlihy & Wing, 1990). Neither a conditional write nor a collection of reads from separately protected records creates a transaction across services. Atomic commitment requires cooperation among participants, as addressed by coordinated commit protocols (Gray & Lamport, 2006). That cooperation is unavailable in our provider model. The controller must instead sequence the operations offered by each service and stop when the available evidence cannot justify continuing.

Wu *et al.* (2026) predict service failures with an LSTM health model and terminate high-risk Saga transactions before they execute. The present study addresses uncertain effects after an operation has been attempted. Prevention and recovery therefore concern different decisions. We neither implement their predictor nor compare performance against it.

Provider Contracts and Uncertainty

The provider profiles represent differences documented in real API contracts. HTTP specifies idempotent methods, whose intended effect is unchanged by repetition, and conditional requests such as If-Match (Fielding *et al.*, 2022). AWS describes request identities that a provider uses to prevent duplicate execution (Featonby, n.d.). Stripe specifies how long results are retained and when an idempotency key, the identifier used to recognize repeated requests, can be reused (Stripe, n.d.). Dataverse supports updates and deletes conditioned on a record version (Microsoft, 2024). These guarantees protect different parts of recovery. We vary them in synthetic profiles; the simulated 60-second period of replay protection is an experimental parameter, not Stripe's documented duration.

Agent Recovery and Transactional Execution

Verified Tool Calls examines verification before retry after failures with uncertain effects (Mansoor *et al.*, 2026). AID-Guard constrains recovery using provider evidence and guarantees about when an action can still take effect (Tong *et al.*, 2026). These works already establish the importance of checking outcomes and refusing actions that the available evidence cannot justify. Our experiments examine their interaction with cancellation and edits to related business records.

RAC provides logged recovery and reversals ordered by dependency (Perera *et al.*, 2026). Atomix coordinates effects through declared operation information and adapters (Mohammadi *et al.*, 2026). CoAgent studies concurrent actions in shared environments (Lyu *et al.*, 2026), while SagaLLM combines planning, validation and compensation (Chang & Geng, 2025). TRW examines observation freshness and dependency-scoped repair (Chang, 2026); Agentic Transaction describes semantic transaction properties for agents (Sun *et al.*, 2026). These works supply the recovery and concurrency foundations for our evaluation.

The selected Atomix source revision includes a reference implementation that retries after version conflicts up to a fixed limit. It manages versions within its own runtime; our services expose versions through separate HTTP APIs. The engineering reference in this study implements conventional conditional retry for those APIs and is not a reproduction of Atomix's algorithm. Cordon stages and validates effects within a task-level transaction boundary and handles uncertain external effects conservatively (Chen *et al.*, 2026). The present study evaluates a composition of established techniques.

We execute selected source revisions of RAC and Atomix through adapters: code that connects their recovery software to the external services. An adapter supplies a callback, a function that the recovery software calls to perform or reverse a business operation. We use runtime to mean that executing recovery software. We vary those callbacks to identify which observed protections they supply. These tests cover the selected runtime components and adapters, rather than the full published benchmark suites.

Evaluation Methodology

AgentBench evaluates agents in diverse interactive environments (Liu *et al.*, 2024). In τ-bench, domain policies and final database state determine the outcome of tool-agent-user interactions (Yao *et al.*, 2024); τ^2-Bench also gives users actions in a shared environment (Barres *et al.*, 2025). Hell or High Water tests alternative planning after external functions fail (Wang *et al.*, 2025). We restrict

the action space to examine stored effects, provider guarantees and selected external edits in isolation. The resulting testbed addresses recovery decisions within that boundary, not the broader planning and conversational abilities covered by these benchmarks.

Fault injection means deliberately introducing a failure at a chosen point in a test. This practice and explicit recovery specifications predate agent benchmarks. FATE and DESTINI systematically test recovery under failures (Gunawi *et al.*, 2011). We use the same experimental principle on a smaller scale: specified failure schedules and stored state allow us to inspect failures. The finite scenario set is not exhaustive and supports neither a state-of-the-art superiority claim nor a formal coverage guarantee.

ConsequenceBench also evaluates external-state correctness, recovery and compensation in synthetic environments (Yuvin Labs, n.d.); the inspected release is a development preview. Recovery benchmarking is therefore an existing research area. Our contribution is a focused case study with separate HTTP services, saved operation histories, intervening edits and explicit comparisons under a common set of service guarantees.

MATERIALS AND METHODS

Workflow and Outcome Definitions

The workflow has three records: an inventory reservation that sets aside stock, an order that uses that stock, and an invoice recording the amount billed for the order. These relationships define the dependency graph. Each service stores its own record versions and operation history. People or other processes can edit protected fields or change which workflow owns the reserved stock through the service APIs. The experiments keep the three-record graph fixed; they do not cover new dependencies or direct database changes that bypass those APIs.

A provider is one of the services exposing an API. Its contract states whether it can recognize a repeated request, look up a recorded operation, or reject an update based on an outdated record version. A request identifier prevents duplicates only if the provider enforces that guarantee. Similarly, finding no recorded result is not proof of failure: the original request might still complete. Separate tests check that the providers follow their declared guarantees, including when a request finishes late or its replay protection expires.

An episode is one execution of a recovery policy under a specified workflow state and fault schedule. The evaluator reads the stored records and histories to check for duplicate effects, damage to protected edits, broken dependencies and incorrect inventory counts. Any of these makes an episode unsafe under the study's rules. Those rules require every active order to have a reservation and every active invoice to have an order. Available stock must equal initial stock minus active reservations. The invoice rule represents the selected cancellation policy; other businesses may permit invoices to remain after cancellation.

Safe completion means that the stored workflow reaches the requested goal without a scored violation. A handoff means that the controller stops and requests manual resolution, also called reconciliation. These measures answer different questions. A lost response may leave the controller unable to confirm an action that the evaluator can see completed safely. An episode can therefore have both safe completion and a handoff; the table columns need not add up to the episode total.

False completion means that the controller reports success even though the stored workflow is incomplete or unsafe. We count it when the controller returns ok, finished or compensated and the evaluator does not find safe completion. This separates an agent's claim from the outcome recorded by the services.

The experiments do not measure a person's work after a handoff. A handoff leaves recovery unresolved from the controller's perspective and cannot be counted as a successful manual repair. Most stages stop observation when the controller returns; the targeted delayed-commit case is also checked after the pending request settles.

For example, suppose request K creates an order but its response is lost. An authoritative lookup that returns K's recorded result can resolve the uncertainty. Otherwise, the controller may repeat the same request only while the provider guarantees that doing so will not create another effect. When neither option is available, it requests manual resolution. Searching for an order and finding nothing cannot exclude delayed visibility or a request still executing.

Now suppose a person changes the order during cancellation. The controller reads the current record and checks for an instruction to keep it active. A version condition prevents its update from silently overwriting a change made after that read. The condition protects that record only: the controller must also check the invoice and reservation, and stop reversing dependent records when an earlier reversal remains unconfirmed.

Figure 1 separates resolution of an uncertain request from the checks required before compensation. A recorded outcome resolves an individual operation; workflow completion still requires the final-state checks described below.

Explicit Provider Guarantees

Table 1 lists the four simulated API profiles. Replay deduplication means that repeating a request with the same identity within the supported period does not repeat its effect. Authoritative lookup returns a recorded outcome for that request; an absent entry does not rule out later completion. A conditional write checks whether the target record still has the version that the controller read. It does not make updates across services atomic.

The provider reports how long replay protection lasts and how long a stored result remains hidden from

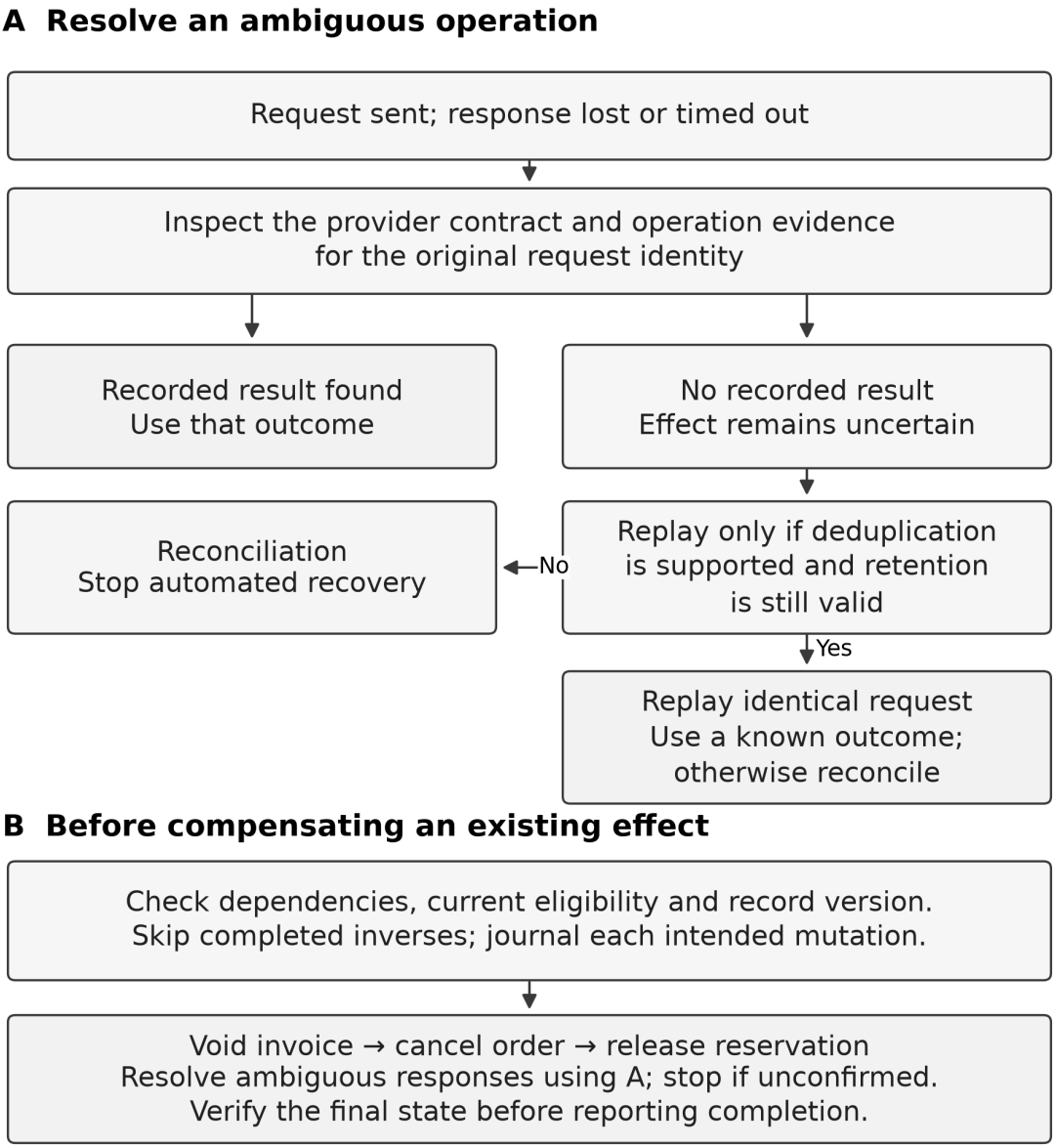


Figure 1: Resolving an uncertain request and checking whether cancellation may proceed. Panel A checks the request outcome; panel B shows cancellation checks shared with the stop-on-conflict policy. The execution procedure adds limited retries after a confirmed rejection with no effect. Reconciliation means stopping for manual resolution.

Table 1: Synthetic provider contracts

Profile	Replay deduplication	Authoritative lookup	Conditional writes
strong	Within retention	Yes	Yes
dedup_only	Within retention	No	No
observable	No	Yes	Yes
legacy	No	No	No

a query. Both periods default to 60 seconds. Fault controls can break the connection before or after an action takes effect, create an incomplete record, reject a request or introduce an external edit. Separate provider checks also stop a process, advance the test clock or pause a request during execution. The model comparisons do not exercise every such check.

Interpretation of Final State

For creation or repair, the goal is one active invoice, order and reservation, with an address on the order. For cancellation, the goal is no active records and restored stock. The evaluator combines this goal check with the safety checks above. A workflow that has no violation

but has not reached its goal is unresolved. The results examine a separate limitation of the edit-protection rule when a new hold is introduced after cancellation.

Experimental Design

The testbed runs separate local HTTP processes for order, inventory and billing, each backed by a SQLite database. For each episode, the harness creates an initial workflow and injects the specified failures or external edits. The controller receives only the service contracts, public responses and its own decision history. The evaluator can inspect the databases and event histories, so it can check what actually happened even when the controller received a timeout.

The supporting model experiments use Claude Haiku 4.5 (claude-haiku-4-5-20251001) and Gemini 3.1 Flash-Lite (gemini-3.1-flash-lite), with temperature zero and a maximum of 1,024 output tokens per call. An episode allows at most four decisions from seven actions: query, read, retry, repair, compensate, finish and reconcile. Compensate requests a reversal; reconcile requests manual resolution.

The model selects an action from this fixed interface. It cannot supply arbitrary URLs, request identities, payloads or code. In the cancellation studies, retry, repair and compensate call the assigned deterministic recovery controller, which chooses the individual service operations. The model experiments therefore test recovery triggered through a language-model interface. They do not test whether the model can independently plan a safer cancellation sequence.

The primary comparison is deterministic because it tests the recovery execution layer. Model runs, failure tests and tests of individual safeguards address supporting questions. Table 2 identifies their separate designs and episode counts. Repeated runs of one configuration are not independent samples of enterprise workflows.

The primary comparison tests four policies on ninety configurations: twenty established dependency-challenge configurations, ten cases targeting retry behavior, and sixty additional combinations of faults and edits. Each policy runs each configuration three times, giving 1,080 episodes. We call the last sixty configurations prospective because they were specified after the controllers were frozen and before execution. The other configurations were used during development. All were designed within this project; the prospective set is not an independently authored validation set.

Development pilots exposed failures used to improve the testbed and controller. One showed that inventory release could execute twice even when the second call used a fresh record version. This motivated checking whether a reversal had already completed. Pilot outcomes are retained separately and excluded from the reported study counts.

The initial 720-episode study asks whether enforced checks change the results of model-selected actions. For each model, five conditions use the same twelve scenario families and five repetitions. Two deterministic controls use those families and repetitions as well. Prompt-only controls ask the model to retry, verify or compensate; they are not implementations of named published methods. The guarded verification condition keeps the verification instructions and adds enforced checks.

The 60-episode cross-service stress test asks whether checking an individual record also protects related records. It applies four interventions under five conditions, with three repetitions. The retained failures identify dependency problems that the 840-episode challenge subsequently examines.

The revised challenge tests those dependency checks and seven ablations, each removing one safeguard. It contains 840 episodes across twenty configurations, fourteen conditions and three repetitions: 600 deterministic and 240 model episodes. The cases were fixed before that challenge ran but were informed by development failures. They test the revision under specified conditions rather than independently validating its generality.

The native-runtime comparisons contain 96 retry episodes, 36 single-effect reversal episodes and 120 episodes reversing the full workflow. Exact RAC and Atomix source revisions are recorded in the supplementary execution details. For the full-workflow comparison, a common test setup creates the forward effects and passes their actual request parameters and recorded results to each runtime. We then test reversal scheduling and callback behavior. This setup does not test native crash recovery.

Schedules, random seeds, worker counts and stage-specific environments are recorded in the supplementary execution details. Controllers and service databases are isolated by episode, while workers share the host. Two incomplete attempts at the primary comparison were excluded after fixing initialization and bookkeeping errors in the harness. The corrected run used the same controller implementations and scenario schedule; all attempts are retained.

The main testbed uses Python HTTP services and SQLite on macOS. Native-runtime tests use a separately documented Python environment. Recorded timings therefore describe these local setups, not comparative production performance.

We report descriptive counts for matched configurations. Deliberate scenario selection and repeated runs do not support estimates of deployment failure rates. Infrastructure errors, calibration runs and corrections are documented separately in the supplementary execution details. The analysis preserves the original scores rather than treating a later diagnostic as a prespecified test.

Evidence Groups

Table 2 maps the evidence to the recovery decisions tested; it is not a pooled sample. The primary comparison measures differences between four recovery policies under common conditions. Supporting experiments test enforcement, dependency checks, rejection guarantees

and runtime adapters. Development pilots and software regression checks are excluded from these episode counts.

Table 2: Experimental stages

Stage	Design	N
Initial main	12 families × 12 conditions × 5 repeats	720
Cross-service stress	4 interventions × 5 conditions × 3 repeats	60
Revised challenge	20 configurations × 14 conditions × 3 repeats	840
Native retry	16 configurations × 6 adapters	96
Native single inverse	9 configurations × 4 adapters	36
Native full graph	20 configurations × 6 adapters	120
Revalidation, matched	20 configurations × 3 conditions × 3 repeats	180
Revalidation, follow-up	10 configurations × 2 conditions × 3 repeats	60
Final model, matched	20 configurations × 2 models × 2 controllers × 3 repeats	240
Final model, targeted	10 configurations × 2 models × 2 controllers × 3 repeats	120
Strong baseline, historical	30 configurations × 4 policies × 3 repeats	360
Strong baseline, prospective	60 configurations × 4 policies × 3 repeats	720

related services; and the revised challenge tests dependency checks and their ablations. Native-runtime studies vary the callbacks supplied to RAC and Atomix. These stages use different questions and conditions, so their episode counts and outcomes are reported separately.

The rejection-contract comparison asks whether rereading after a confirmed version rejection permits more recovery than stopping immediately. Its matched and targeted parts contain 180 and 60 episodes. The corresponding model study adds 240 and 120 episodes. These experiments isolate the effect of allowing another attempt; the primary comparison determines whether that benefit distinguishes the evaluated controller from conventional recovery with equivalent safeguards.

Recovery Policies

The controller cancels the workflow by marking the invoice inactive (voiding it), cancelling the order and returning the reserved stock to available inventory, in that order. Each reversal is also called an inverse or compensation operation. Before a new attempt, the controller reads the related records and checks whether cancellation is still allowed, whether the workflow still owns the reservation, and whether that reversal has already completed. If a change is needed, it sends the current target version as a condition on the update.

Every attempt has an identity and exact request contents saved before transmission in a journal, a record of attempts that survives a process restart. If the response is missing, the controller looks up that attempt or repeats it under the provider's deduplication guarantee. It cannot assume that a missing response permits a fresh attempt. A confirmed version rejection is different: if the provider guarantees that the request ended without changing anything, the controller can reread the records, check whether cancellation is still allowed, and submit a new conditional attempt. We call this revalidation and allow it at most twice per service.

The guarantee that a rejected request has ended without effect is an explicit adapter setting, disabled by default. The test provider checks the version before changing the record, but its public contract endpoint does not advertise this extra guarantee. We audit the implementation and configure the assumption for the comparison. A generic error is insufficient evidence. On restart, the controller resumes a journaled attempt; it rejects reuse of that journal for different records or recovery assumptions.

These checks do not create a transaction or lock across the services. A record can change between separate reads and writes. The experiments test the checks under the fixed dependency graph and specified event schedules; results under other workflows and schedules require further evaluation.

Execution Procedure

The cancellation procedure is as follows. An unconfirmed reversal stops the sequence so that the controller does not continue changing dependent records.

1. Read the three service contracts. Stop for manual resolution if a service cannot enforce a required version condition on an update.
2. Read the related records. Check that they are available, whether any instruction protects them from cancellation, and who owns the reserved stock.
3. Void the invoice, cancel the order and release the reserved stock, in that order.
4. Before a new attempt, reread the records and check whether the reversal is still allowed. Skip it if already

completed. Include the target record's current version as a condition on the request.

5. Save the attempt identifier and exact request contents before sending them. After a restart, resolve the recorded attempt first; reject reuse for different records or recovery assumptions.

6. If the outcome is unknown, check the operation history. Repeat the same request only while the service guarantees that it will not repeat the effect. Stop replaying two seconds before that guarantee expires.

7. After a confirmed version rejection, read and check again only if the configured guarantee establishes that the request ended without effect and fewer than two revalidations have occurred for that service. Do not continue to later reversals while an earlier one remains unconfirmed.

8. Read the final records. Report completion only when cancellation has reached its required final state; otherwise request manual resolution.

The stop-on-conflict policy, labelled dependency in the supporting comparisons, uses the same ordered checks but requests manual resolution after a version rejection. The contract-disabled condition runs the evaluated implementation with the extra rejection guarantee switched off. It tests whether allowing a fresh attempt depends on that guarantee.

Why Individual Record Checks Are Insufficient

Suppose the order is cancelled before the invoice is voided, and billing then rejects the void request. The result is an active invoice without its required order. The order's version check cannot prevent this failure in another service. Voiding the invoice first avoids that particular ordering problem, but the later order cancellation may still fail. In that case, part of the workflow remains reversed and requires manual resolution.

A note edit presents a different decision. If the note changes after the controller reads the invoice, the provider rejects an update carrying the old version. A guarantee that this rejection ended without effect lets the controller read again and check whether the invoice can still be voided while preserving the note. A lost response provides no such assurance. The distinction is between evidence that the attempted change did not occur and uncertainty about whether it did.

Comparison Policies and Prospective Scenarios

The primary comparison includes a separately implemented conventional reference. It uses the same HTTP client and journal but neither calls nor inherits the evaluated controller. Both implementations use the same reversal order, service guarantees, two-revalidation limit and two-second margin before replay protection expires. Both check all related records before a new update, skip completed reversals and verify the final state. This reference tests whether the evaluated implementation contributes beyond those shared safeguards.

Table 3 labels the conventional reference Graph retry and the evaluated controller Final. Target retry checks all records initially, then only the record it is about to change; this isolates the scope of repeated checks. Stop requests manual resolution after a version conflict. Graph retry includes an additional initial check, so we do not use timing to claim an efficiency advantage. It passed five development regression checks before its implementation was frozen.

The sixty prospective configurations include thirty-six combinations of note edits and connection failures: three services, zero to three note conflicts, and no response loss or loss before or after the action takes effect. Twelve configurations combine API capabilities with connection failures. Six change protections on another record, and six have another actor complete a reversal. Events occur at specified update attempts. A policy that stops earlier may encounter fewer events; the log records those that occurred. An external reversal can itself break a dependency, so an inconsistent final state does not necessarily mean the controller caused it.

The prospective cases and the engineering reference were authored within this project. The frozen protocol specified that matching results would rule out a claim of controller superiority in this comparison. Provider behavior, the scoring rules and the evaluated controller were held unchanged. These controls separate the question being tested from later interpretation of the outcome.

RESULTS & DISCUSSION

We first compare recovery policies under common service guarantees, then examine the meaning of their completion counts. Supporting results isolate rejection handling, model action selection, dependency checks and external adapters. Each comparison reports its own group of configurations; counts should not be combined across tables.

Equivalent Safeguards and Recovery Outcomes

The conventional reference and the evaluated controller matched in all 270 paired episodes (Table 3; Figure 2). Each pair had the same scored safety, completion, handoff, unresolved-state, false-completion and write counts. The measured benefit of bounded revalidation is therefore achievable through conventional conditional retry with the same safeguards. These results attribute the benefit to that shared combination of protections; they do not establish a distinct algorithmic advantage or equivalence under every possible schedule.

N is the number of episodes; Goal is safe completion under the prespecified scoring rules; Handoff is a request for manual resolution; Writes counts requests to change records during recovery. Historical and targeted configurations were reused from earlier stages; prospective configurations were specified after the controller freeze. Each configuration has three repetitions. Unsafe includes inconsistent states left by external reversals: the nine unsafe Stop episodes do not imply that Stop created those effects.

Table 3: Strong engineering baseline comparison

Cohort	Policy	N	Unsafe	Goal	Handoff	Writes
Historical	Stop	60	0	18	42	96
Historical	Target retry	60	0	24	36	114
Historical	Graph retry	60	0	24	36	114
Historical	Final	60	0	24	36	114
Targeted	Stop	30	0	0	30	42
Targeted	Target retry	30	0	9	21	81
Targeted	Graph retry	30	0	9	21	81
Targeted	Final	30	0	9	21	81
Prospective	Stop	180	9	27	153	360
Prospective	Target retry	180	0	117	63	645
Prospective	Graph retry	180	0	108	72	621
Prospective	Final	180	0	108	72	621

On the sixty new configurations, Graph retry and Final each completed 108 of 180 episodes, handed off 72 and issued 621 recovery writes. Target retry appeared to complete nine more episodes. Those nine results arose when another actor added a keep-active instruction, or hold, to a record that was already cancelled. The prespecified scorer checks whether a later mutation violates an edit; it does not require an already cancelled record to become active again. All four policies left those nine episodes with a hold on an inactive record.

We examined this mismatch in a separate analysis after seeing the results. The additional check disqualifies a goal state that leaves such a hold unsatisfied. Under that check, Target retry has 108 completions rather than 117, eliminating its apparent advantage. The original scores remain in Table 3. The example shows why avoiding a new harmful write and fulfilling the latest human instruction are different requirements; it does not determine how an operational system should resolve a late hold.

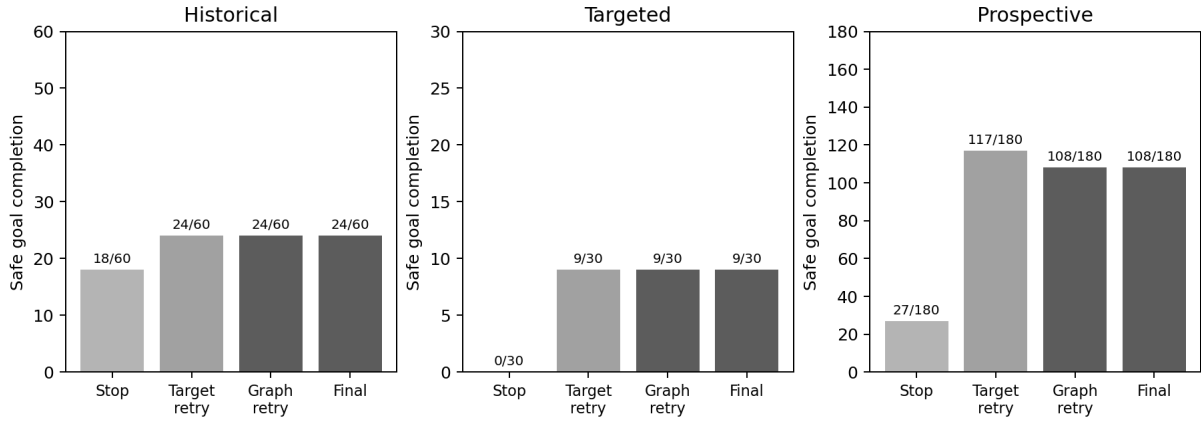


Figure 2: Safe completion under the prespecified scoring rules. Graph retry and Final have identical counts in every group. Target retry has nine additional prospective completions only under the original edit-protection rule; the separate check for unsatisfied holds removes that difference.

Confirmed Rejection and Model-Triggered Recovery

The rejection-contract comparisons isolate a specific decision: whether to stop after a version conflict or reread the records and attempt recovery again. The model comparison tests that recovery layer when a language model selects the initiating action. The conventional reference in Table 3 provides the separate test of whether the measured completion gain is specific to the evaluated implementation.

Table 4 reports the deterministic comparison. On the matched configurations, safe completion increased from 18/60 to 24/60 and unresolved handoffs decreased from 42/60 to 36/60, with zero observed unsafe episodes. The gains occurred when an invoice note changed and when its keep-active flag changed and then returned to its original value (the ABA case). Each case was repeated three times. Disabling the guarantee that rejection ended without effect removed the gain. Recovery write requests increased from

Table 4: Effect of the rejection guarantee on deterministic recovery

Cohort / controller	N	Unsafe	Goal	Handoff	Writes
Matched / dependency	60	0	18	42	96
Matched / contract off	60	0	18	42	96
Matched / revalidation	60	0	24	36	114
Follow-up / dependency	30	0	0	30	42
Follow-up / revalidation	30	0	9	21	81

96 to 114 across the sixty episodes.

N is the number of episodes; Goal is safe completion; Handoff requests manual resolution; Writes counts requests to change records during recovery. Dependency stops after a version rejection. Revalidation permits up to two fresh checks and attempts under the required rejection guarantee. Contract off disables that guarantee. Matched uses the twenty dependency-challenge configurations; follow-up uses ten targeted configurations.

On the ten targeted configurations, safe completion increased from 0/30 to 9/30, with zero observed violations under either controller. Gains occurred for an order-note conflict, two consecutive note conflicts, and a lost successful response after revalidation. Repeated conflicts beyond the bound, protected changes, missing rejection semantics, business rejection and uncertain effects without replay still led to handoff. The delayed-commit case was checked at controller return and after the pending request settled.

The separate matched model evaluation contains 360 completed episodes and 360 recorded model calls, with no execution errors. For each model, the baseline and final controller reproduce the deterministic comparison: safe completion rises from 18/60 to 24/60 on the matched set and from 0/30 to 9/30 on the targeted

Table 5: Model-triggered recovery with and without bounded revalidation

Cohort	Model / policy	N	Unsafe	Goal	Handoff	Writes
Matched	Claude / base	60	0	18	42	96
Matched	Claude / final	60	0	24	36	114
Matched	Gemini / base	60	0	18	42	96
Matched	Gemini / final	60	0	24	36	114
Targeted	Claude / base	30	0	0	30	42
Targeted	Claude / final	30	0	9	21	81
Targeted	Gemini / base	30	0	0	30	42
Targeted	Gemini / final	30	0	9	21	81

set. All eight combinations of model, controller and configuration group have zero observed unsafe episodes and zero false completion reports (Table 5; Figure 3).

Base is the policy that stops on a version conflict; final permits bounded revalidation. Matched uses twenty configurations and targeted uses ten, each repeated

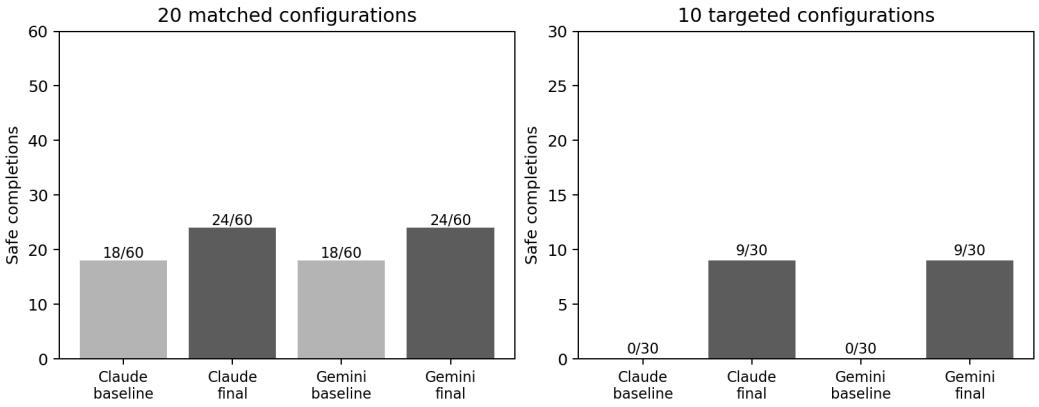


Figure 3: Safe completion under model-triggered recovery. Both controllers recorded zero unsafe episodes in these tested configurations. Remaining outcomes were unresolved handoffs.

three times for each model and policy. The models select an action; the assigned controller executes the service operations.

Both models triggered recovery that produced the same counts as deterministic execution. Each made one recorded action choice per episode in this comparison, while the controller executed the recovery sequence. The result shows that these safeguards also operated through the tested model interface. It does not establish an advantage from model reasoning. Raw responses, matched configurations, token accounting and recovery times are supplied in the supplement.

Enforced Checks Compared with Prompt Instructions

The initial 720-episode study compared asking a model to recover carefully with enforcing checks in the execution layer (Table 6). Both guarded policies recorded zero violations for each model. The deterministic engineering control also recorded zero and matched the highest guarded completion count, 40/60. In the matched verification conditions, adding enforcement reduced unsafe episodes from 30 to zero for Claude and from 29 to zero for Gemini. These results concern the initial scenarios; the cross-service stress test below exposed failures outside them.

Table 6: Initial main study

Provider	Policy	N	Unsafe	Safe goal	Handoff
Claude	Guarded model	60	0	40	20
Claude	Guarded verify	60	0	40	20
Claude	Retry prompt	60	35	25	0
Claude	Saga prompt	60	30	28	0
Claude	Verify prompt	60	30	30	0
Fixed	Abstain	60	0	10	60
Fixed	Engineering	60	0	40	25
Gemini	Guarded model	60	0	40	20
Gemini	Guarded verify	60	0	35	25
Gemini	Retry prompt	60	35	25	0
Gemini	Saga prompt	60	35	23	2
Gemini	Verify prompt	60	29	23	8

N is the number of episodes. Fixed denotes a deterministic control. Safe goal measures stored workflow state; Handoff records the controller's request for manual resolution. They can overlap, for example when an action has completed but its response is lost. The always-abstain policy can therefore encounter a safe goal state while still handing off every episode.

Failures Across Related Services

Record-level enforcement did not protect the complete workflow under cross-service interventions. Each guarded model and the record-level deterministic controller had 9/12 unsafe episodes in the stress suite; prompt-only verification had 12/12. Recovery either voided a protected invoice or left an active invoice attached to a cancelled order. The individual record checks had passed, but the workflow violated its declared business policy. This distinguishes the correctness of a conditional API update from the correctness of recovery across related records.

Which Safeguards Changed the Outcomes

The revised challenge separates avoidance of violations from successful recovery (Table 7). The table reports all fourteen conditions. For each model, the revision reduced unsafe episodes from 21/60 to zero. Its safe completion count stayed at 18/60, while handoffs increased from 21/60 to 42/60. The full deterministic controller matched the revised models. Relative to the original deterministic controller, however, completion fell from 27/60 to 18/60 as unsafe episodes fell from 18/60 to zero. The revision prevented the observed violations by accepting more unresolved work, rather than by completing more episodes.

N is the number of episodes; V counts unsafe episodes; Goal counts safe completion; H counts handoffs; F counts false reports of completion. Fixed denotes a deterministic control. Each No condition removes one check from Full. Full is the dependency controller that stops at version conflicts, before bounded revalidation was added. Eligibility asks whether a reversal is allowed now; preflight checks all records before starting; inverse skip avoids repeating a completed reversal. A guard is the code enforcing these checks.

Removing outcome confirmation produced 21 unsafe episodes; removing reversal ordering produced 12; and removing version checks produced 9. Removing contract enforcement, checks that a reversal is still allowed or checks for an already completed reversal produced 3 each.

Table 7: Revised challenge and ablations

Provider	Policy	N	V	Goal	H	F
Claude	Dependency guard	60	0	18	42	0
Claude	Original guard	60	21	18	21	21
Fixed	Abstain	60	0	3	60	0
Fixed	Full	60	0	18	42	0
Fixed	No confirmation	60	21	18	42	0
Fixed	No contract check	60	3	21	36	3
Fixed	No eligibility	60	3	18	39	3
Fixed	No ordering	60	12	18	42	0
Fixed	No preflight	60	0	18	42	0
Fixed	No inverse skip	60	3	15	42	3
Fixed	No version check	60	9	24	27	9
Fixed	Original guard	60	18	27	12	21
Gemini	Dependency guard	60	0	18	42	0
Gemini	Original guard	60	21	18	21	21

Removing the initial check of all records (preflight) had no measured effect. These one-at-a-time tests show which checks changed outcomes in this challenge. They do not establish that every check is necessary or that this is the smallest sufficient set.

Figure 4 compares violations and safe completion across the revised conditions. Figure 5 shows which test configurations fail when one safeguard is removed, showing which injected cases distinguish the safeguards.

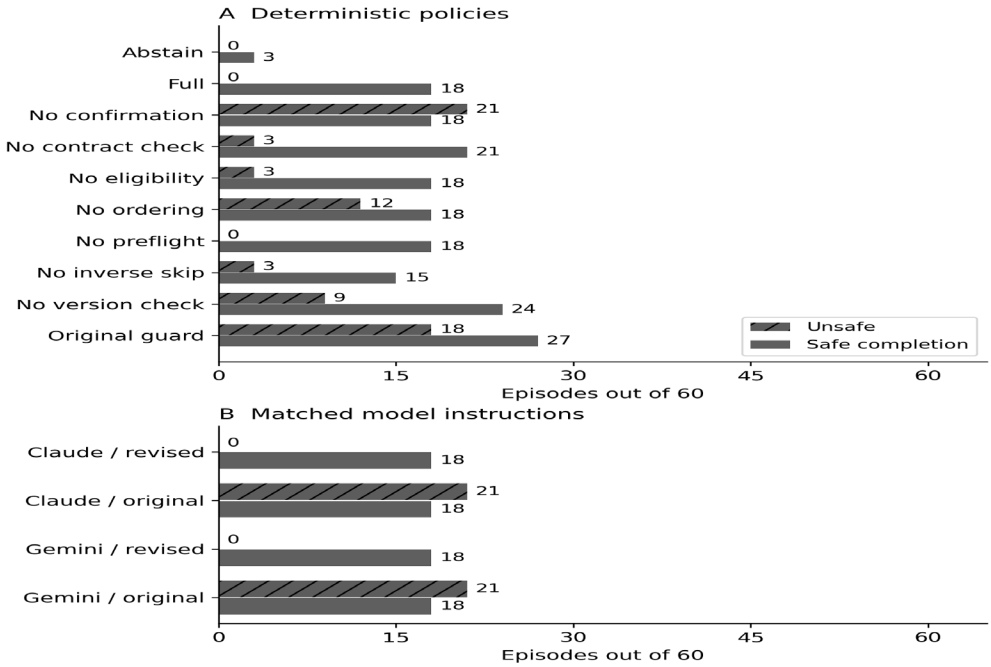


Figure 4: Observed violations and safe completions in the revised challenge. Each condition has twenty configurations repeated three times. Bars are descriptive counts, not estimates of enterprise failure rates. Table 7 reports the handoffs that the completion bars alone cannot show.

How External Adapters Affect Existing Runtimes

Retrying through a native runtime did not by itself prevent duplication. Direct RAC retry and Atomix with application retry each produced duplicates in 4/16 episodes; Atomix without caller retry handed off all sixteen. Adding the shared contract adapter gave both native systems and the deterministic contract controller zero unsafe episodes and ten confirmed completions out of sixteen. The comparison reversing one effect showed the same dependence on callback behavior. Unconditional callbacks caused six unsafe episodes out of nine for each runtime. Conditional callbacks caused zero out of nine, with three completed reversals and six retained effects handed off for reconciliation.

Table 8: Native full graph comparison

Runtime	Adapter	N	Unsafe	Safe goal	Handoff
Atomix	Dependency guard	20	0	6	14
Atomix	Direct	20	10	8	12
Atomix	Record guard	20	10	5	15
RAC	Dependency guard	20	0	6	14
RAC	Direct	20	7	7	13
RAC	Record guard	20	3	4	16

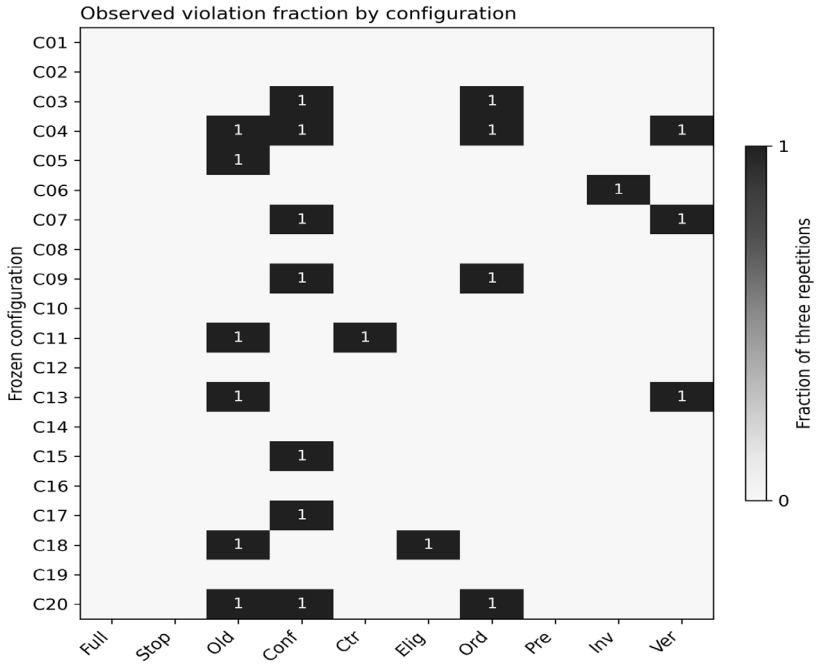


Figure 5: Violation fractions in the deterministic challenge. Cases C01–C20 follow the supplementary case key. Full = revised; Stop = abstain; Old = original. Other columns remove confirmation, contract, eligibility, ordering, preflight, inverse-skip or version checks, respectively. Three repetitions per cell are not independent scenario families.

The full-graph results in Table 8 also depend on the adapter. Both runtimes achieved zero unsafe episodes and six safe completions out of twenty with dependency-aware callbacks, matching the revised controller's configuration-level result. Direct and record-local adapters yielded different counts. These comparisons describe the tested runtime/adapter/provider combinations. They neither rank the complete published methods nor establish that the added dependency checks are native runtime capabilities.

Reproducibility and Validity

A fresh extraction of the reviewer archive reproduces the main comparison tables and checks the records behind them. For the 1,080-episode strong-baseline comparison, verification covers the frozen sources and schedule, 3,240 service databases, operation histories, journal transitions and paired counts. The model comparison additionally checks stored responses and token accounting. The later check for unsatisfied holds remains separate from the prespecified scoring rules.

The evidence archive supports inspection of both execution and scoring. It contains source snapshots, frozen schedules, raw model interactions, HTTP histories, provider databases, outcome tables and SHA-256 manifests. Offline checks recompute results against stored state. The revised challenge and native graph comparison together contain 960 episodes and 2,880 provider databases.

The local regression suite has 61 passing tests: 51 original checks, five for the revalidation controller and five for the engineering reference. Selected upstream checks also passed for Atomix (31) and RAC (57). Software tests check implementation behavior and are not counted as additional experimental episodes.

Median recovery times in the initial main study range from 0.0069 to 2.4889 seconds across conditions. They include fixed continuation and local provider lifecycle work. The supplementary CSV reports these times together with component counts, model calls and token-based cost

estimates. Because workers share a host and provider response latency varies, these measurements should not be read as production performance.

The largest limitation is the workload. Earlier scenarios informed controller development, and the prospective scenarios combine events within the same three-service workflow. They were specified after the controller freeze by the same implementation effort. The results therefore describe these cases, without establishing transfer to independently specified operational workflows. The literature audit is focused rather than exhaustive, and the runtime comparisons cover selected adapters rather than complete published evaluations.

Scoring also depends on business rules. The testbed prohibits keeping an active invoice after cancelling its order, although another business may allow that state. Its original edit rule checks damage caused by later updates, which is weaker than requiring every later instruction to be fulfilled. We expose that difference through the separate hold diagnostic instead of changing the original scores. Different business rules require a separately specified evaluation.

One practitioner provided formative feedback on scenario realism. This feedback informed clarification of the scenarios but was not used to validate the benchmark or calculate experimental results. Details are provided in the supplementary materials.

Implications for Recovery Evaluation

The comparison shows why the reference implementation matters. Bounded revalidation raised completion from 18/60 to 24/60 relative to stopping on conflicts, and conventional graph-wide retry achieved the same result. An evaluation against stopping alone would attribute too much to the particular controller. The supported conclusion concerns the provider's rejection guarantee and the checks used before a fresh attempt. An integration team can implement those protections using established recovery techniques.

The hold diagnostic identifies a different evaluation requirement. Target-only checking appeared to finish more prospective episodes, yet the extra completions left late holds unsatisfied. Checking only that recovery did not overwrite an edit can miss an instruction that conflicts with an already completed action. An evaluator must specify whether success means avoiding further harm, fulfilling the latest instruction, or leaving an explicit unresolved case. The diagnostic was performed after observing the results, so it is reported separately from the prespecified scores.

Unresolved work remains substantial: the two policies checking all records completed 108/180 prospective episodes and handed off 72/180. A handoff can prevent an unjustified action while still transferring work to a person. In the dependency challenge, the always-stop control encountered three already completed goals, while the dependency guard achieved eighteen. These counts distinguish automatic progress from abstention. They do not measure the time or correctness of subsequent human repairs, and the deliberately difficult workload does not estimate a production handoff rate.

The model and native-runtime experiments locate the source of the measured protections. Agreement with deterministic execution points to the controller in the tested model interface. Changing RAC and Atomix callbacks shows that an external adapter can determine whether a reversal preserves the workflow. These results support inspecting the full path from recovery decision to stored business state when evaluating an agent integration.

The next external-validity test should use a workflow specified independently of this testbed and audit the actual provider contracts. It should include different business policies, such as allowing an invoice to remain after cancellation, and evaluate what happens after manual escalation. Repeating the current configurations can check reproducibility but cannot answer those transfer questions.

CONCLUSION

Safe recovery depends on evidence about earlier effects and checks on the current workflow. In our controlled comparison, a conventional implementation with equivalent safeguards matched the evaluated controller in all 270 paired episodes. Both completed 108/180 prospective episodes without scored violations and handed off the rest. A separate diagnostic showed that the apparent gain from checking only the target record depended on a scoring rule that missed unsatisfied holds added after cancellation. The testbed, failure histories and comparison code provide a reproducible basis for examining these recovery decisions. Broader claims require independently specified workflows and business rules.

Data and Software Availability

The supplementary package provides the source code, synthetic data, experimental configurations, recorded results, and instructions for reproducing the analysis.

REFERENCES

Barres, V., Dong, H., Ray, S., Si, X., & Narasimhan, K. (2025). *τ²-Bench: Evaluating conversational agents in a dual-control environment.* arXiv. https://arxiv.org/abs/2506.07982v1

Birrell, A. D., & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems, 2*(1), 39–59. https://doi.org/10.1145/2080.357392

Chang, E. Y. (2026). TRW: TRACE-RealWorld—*An auditable consistency contract for world models as materialized views.* arXiv. https://arxiv.org/abs/2607.21910v2

Chang, E. Y., & Geng, L. (2025). *SagaLLM: Context management, validation, and transaction guarantees for multi-agent LLM planning.* arXiv. https://arxiv.org/abs/2503.11951v3

Chen, Z., Liu, H., Xu, D., Dong, D., Li, J., Pu, B., & Zhai, J. (2026). *Cordon: Semantic transactions for tool-*

using LLM agents. arXiv. https://doi.org/10.48550/arXiv.2606.17573

Featonby, M. (n.d.). *Making retries safe with idempotent APIs. Amazon Builders' Library*. https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

Fielding, R., Nottingham, M., & Reschke, J. (2022). *HTTP semantics* (RFC 9110). Internet Engineering Task Force. https://doi.org/10.17487/RFC9110

Garcia-Molina, H., & Salem, K. (1987). Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on management of data* (pp. 249–259). ACM Press. https://doi.org/10.1145/38713.38742

Gray, J., & Lamport, L. (2006). Consensus on transaction commit. *ACM Transactions on Database Systems, 31*(1), 133–160. https://doi.org/10.1145/1132863.1132867

Gunawi, H. S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Sen, K., & Borthakur, D. (2011). FATE and DESTINI: A framework for cloud recovery testing. In *8th USENIX Symposium on Networked Systems Design and Implementation* (NSDI 11). USENIX Association. https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

Herlihy, M. P., & Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems, 12*(3), 463–492. https://www.cs.cmu.edu/~wing/publications/HerlihyWing90.pdf

Korth, H. F., Levy, E., & Silberschatz, A. (1990). A formal approach to recovery by compensating transactions. In D. McLeod, R. Sacks-Davis, & H. Schek (Eds.), *Proceedings of the 16th international conference on very large data bases* (pp. 95–106). Morgan Kaufmann. https://www.vldb.org/conf/1990/P095.PDF

Kung, H. T., & Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems, 6*(2), 213–226. https://doi.org/10.1145/319566.319567

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., … Tang, J. (2024). AgentBench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*. https://proceedings.iclr.cc/paper_files/paper/2024/hash/e9df36b21ff4ee211a8b71ee8b7e9f57-Abstract-Conference.html

Lyu, H., Zhang, D., Wu, M., Wei, X., & Chen, H. (2026). CoAgent: Concurrency control for multi-agent systems. arXiv. https://arxiv.org/abs/2606.15376v1

Mansoor, I. K., Phadke, A., & Rana, P. (2026). *Verified tool calls improve LLM agent reliability under non-atomic failures*. arXiv. https://arxiv.org/abs/2608.02645v1

Microsoft. (n.d.). *Compensating transaction pattern*. Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

Microsoft. (2024). *Perform conditional operations using the Web API*. Microsoft Dataverse documentation. https://learn.microsoft.com/en-us/power-apps/developer/data-platform/webapi/perform-conditional-operations-using-web-api

Mohammadi, B., Potamitis, N., Klein, L., Arora, A., & Bindschaedler, L. (2026). *Atomix: Timely, transactional tool use for reliable agentic workflows*. arXiv. https://arxiv.org/abs/2602.14849v2

National Institute of Standards and Technology. (2026, February 17). *Announcing the "AI Agent Standards Initiative" for interoperable and secure innovation*. https://www.nist.gov/news-events/news/2026/02/announcing-ai-agent-standards-initiative-interoperable-and-secure

Perera, S., Hapuarachchi, K., Leymann, F., & Khalaf, R. (2026). Robust agent compensation (RAC): Teaching AI agents to compensate. In *Proceedings of the ACM Conference on AI and Agentic Systems* (pp. 253–262). Association for Computing Machinery. https://doi.org/10.1145/3786335.3813141

Stripe. (n.d.). *Idempotent requests*. https://docs.stripe.com/api/idempotent_requests

Sun, Z., Wang, X., & Li, G. (2026). *Agentic transaction: Towards ACID-compliant agent systems*. arXiv. https://arxiv.org/abs/2608.13900v1

Tong, Y., Dai, L., & Guo, S. (2026). *AID-Guard: Stateful authorization for delegated agent effects*. arXiv. https://arxiv.org/abs/2608.21159v1

Wang, A., Hager, S., Asija, A., Khashabi, D., & Andrews, N. (2025). Hell or high water: Evaluating agentic recovery from external failures. In *Second Conference on Language Modeling*. https://doi.org/10.48550/arXiv.2508.11027

Wu, J., Chen, X., & Shi, R. (2026). A proactive fault tolerance approach for Saga transactions in microservices using a health prediction model. *Journal of Universal Computer Science, 32*(8), 1150–1172. https://doi.org/10.3897/jucs.175609

Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. (2024). τ-bench: *A benchmark for tool-agent-user interaction in real-world domains*. arXiv. https://arxiv.org/abs/2406.12045v1

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.2210.03629

Yuvin Labs. (n.d.). *ConsequenceBench* [Computer software]. GitHub. https://github.com/yuvin-labs/consequencebench