When Conversations Cannot Resume: An Empirical Study of Persistent-State Failures in LLM Applications

SHIRSHAK JUNG THAPA, Independent Researcher, USA

LLM applications save conversations so that users can reload and resume them. We study what happens when a saved conversation cannot be resumed. A *stuck* conversation's stored record never reaches a finished state. Every later request built from a *bricked* conversation's stored history is rejected, because the history, or the code that replays it, breaks the message rules. Under a protocol written before any issue was classified, we analysed 275 GitHub issues (261 root-cause clusters, 128 repositories), labelled by an LLM pipeline checked against a blind human rater. We reproduced 12 defects in 10 applications, before and, where one exists, after the fix, and confirmed the rejections on the real Anthropic and OpenAI APIs. The main finding is that, in our reproductions, bricked conversations were mostly broken not by the agent loop but by the code around it: in 5 of 8, an extension, middleware, plugin, compaction step or command dispatcher saves a message between a tool call and its result. Fixes tend to repair the history when it is read back rather than prevent the bad write, and repair or fix code itself went wrong in three reproductions. Rules tested against the real API agreed with Anthropic on 120 of 121 held-out histories, and on 80 of 80 after one revision; our OpenAI rules agreed with OpenAI on 161 of 177, and every miss was a duplicate result. The history check in LangGraph's prebuilt agent, which asks only whether each call has a result somewhere, caught only 38–42% of the histories Anthropic rejected. The rules also differ by API: OpenAI's Responses API accepts the interleaving that Anthropic and Chat Completions reject. Because defects on the read path leave no trace in storage, saved conversations should be checked both at rest and on the outgoing request. Finally, we used the mechanism to search 15 current open-source applications and frameworks, and found six previously unreported defects in five of them. The five that brick conversations were each confirmed on the real API; in each, a message is saved while a tool call is still open, and three of these five sit in or next to code added by an earlier fix.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Artificial intelligence*.

Additional Key Words and Phrases: LLM applications, agents, conversation state, persistence, empirical study, bug reproduction, differential testing

1 INTRODUCTION

You come back to a conversation with an AI assistant, and it will not continue. The page shows "Loading…" forever, or every message you send fails with the same error. When the cause is the record the application saved, reloading does not help: the same broken record is read back every time.

LLM applications save conversations so that users can reload, resume and branch them. A chat front-end stores a message tree, a coding agent appends to a session file, and an agent framework saves its state after every step. Unlike most saved data, this record is later *sent back to the model provider*, and the provider checks it against strict rules. The rules that matter here concern tools: on Anthropic's API, for example, a tool call must be answered at the start of the very next user message. A saved record that breaks this rule is rejected every time it is sent.

Saved state can therefore end a conversation in two ways:

- **Stuck (STUCK).** An interaction is cut short (a cancel, crash or disconnect) or races with another action, and the saved record never reaches a finished state. The interface shows a running indicator forever, even after a reload.

Author's address: Shirshak Jung Thapa, Independent Researcher, Orlando, Florida, USA, shirshakjungthapa@gmail.com.

- **Bricked (BRICKED).** Every later request built from the saved history is rejected, usually with HTTP 400, because the history, or the code that replays it, breaks the rules, until one of them is repaired.

Either way, a failure that should have cost one reply costs the whole conversation.

Developers know these failures well: issue trackers hold hundreds of reports quoting the providers' error messages, frameworks ship "history repair" code, and a standalone library now validates and repairs histories [5]. Yet studies of LLM-application bugs file these failures under broad labels such as "Session & State Management" [1, 3, 15, 16, 20]. They do not answer the basic questions: what writes the bad state, why fixes do not stop it, and whether existing checks would catch it. We asked five questions:

RQ1 What goes wrong? Which interruptions and broken rules leave a conversation that cannot resume?
RQ2 Where is the bad state written, and where is it fixed?
RQ3 What mechanism breaks the conversation? When we reproduce the failures on real code, which component writes the bad state, and how does that write break the provider's rules?
RQ4 Can it be caught before the provider rejects it, and where must the check run?
RQ5 Does the mechanism found in RQ3 lead us to previously unreported defects?

The main finding is that, in our reproductions, bricked conversations were mostly broken not by the code that calls the model and runs tools, but by the code around it. In 5 of our 8 bricked reproductions, an extension, middleware, plugin, compaction step or command dispatcher saved a message while a tool call was still open. The message (a status notice, a typed command, a loop warning, an image or a summary) is harmless on its own; placed between a tool call and its result, it breaks the provider's rule, and the conversation cannot continue.

The fixes and the safeguards tell the same story. In the issues we studied, fixes more often repair the history when it is read back than stop the bad write, which leaves the broken record in storage for other readers; in three reproductions the repair or fix code itself went wrong. The history check shipped with LangGraph's prebuilt agent asks only whether each tool call has a result *somewhere*, not whether it comes *next*, and caught only 38–42% of the histories Anthropic rejected. Rules checked against the provider held up: ours agreed with Anthropic on 120 of 121 held-out histories, and on 80 of 80 after one revision. And the rules differ by API: OpenAI's newer Responses API accepts the very interleaving that Anthropic and OpenAI's Chat Completions API reject.

The mechanism also predicts new defects. Searching with it, we found six previously unreported defects, in five of the 15 applications and frameworks. Five of them brick conversations, and in each one a message is saved while a tool call is still open. Three sit in or next to code written to fix an earlier bug. We reported all six, and one fix has already been merged. The lesson we draw is that every component that writes into a conversation history, not only the agent loop, must follow the API's rules, recovery code included.

Contributions.

- **Issue study** of 275 issues (261 distinct root causes) separating stuck from bricked conversations, rated by an LLM pipeline checked against a blind human rater, with every protocol change reported.
- **Reproductions** of 12 defects, before and, where one exists, after the fix, confirmed on the real APIs, with every attempt reported.

- **Mechanisms** seen in the reproductions: messages saved between a call and its result, harmful repair code, readers that hide stuck records, and writers and readers that are different programs.
- **History rules tested on the real APIs**, a measuring instrument rather than a new tool (§9), compared with a deployed framework check and an existing validator.
- **New defects**: six found in 15 current applications and frameworks, all reported upstream; every negative result is recorded.
- **A replication package** with the protocol, data, harnesses and analysis code.

2 BACKGROUND

To see how a conversation breaks for good, follow it through its life: where the application stores it, how it writes to it and reads it back, and what the provider checks when it is sent again.

Two parties are involved. The *application* (a chat front-end such as open-webui, a coding agent such as gptme, or an agent built with a framework such as LangGraph) saves the conversation. The *provider* (such as Anthropic or OpenAI) runs the model; the application sends it the saved conversation with every request, and it rejects any that break its rules.

Saved conversations. We call any durable record from which an application rebuilds a conversation its *persisted conversation state*: a server database (open-webui's `chat` tree and `chat_message` table), a local session file written by a command-line agent (a JSONL transcript), or a framework checkpoint (LangGraph's `SqliteSaver`). The code that adds messages to this record is the *write path*; the code that later loads it and turns it into a request is the *read path* (or replay path). The read path may transform, filter or repair what it loads, so a defect can sit on either path, and a fix can be applied on either.

The provider's rules. Each request carries the whole conversation so far, and the provider checks its structure before answering. The rules matter most around tools. A valid history looks like this (illustrative, in Anthropic's format):

> user: *"What is the weather in Paris?"* → assistant: *calls* `weather(Paris)` → user: *tool result "18 ℃"* → assistant: *"It is 18 ℃."*

Whether anything may be saved between the call and its result depends on the API. For the Anthropic Messages API, we confirmed four rules on the real API (§5, §6): (1) every tool call (`tool_use`) must be answered by a result (`tool_result`) at the start of the very next user message; (2) every result must answer a call in the message directly before it; (3) each call may be answered only once; and (4) consecutive messages from the same role are merged into one before these checks. The last rule has a surprising effect. A user message saved between a call and its result merges with the result message, so the result is no longer first, and the request is rejected. An extra assistant message merges with the call instead, and the history is accepted.

The rules are not the same everywhere. **OpenAI Chat Completions** also requires each call's result to follow directly, answered once, but does *not* merge consecutive messages, so an assistant message between a call and its result is rejected here although Anthropic accepts it (§6). **OpenAI's newer Responses API** ignores order: each call needs an output somewhere, and no output may carry an unknown `call_id`. Where a rule applies, the penalty is the same: HTTP 400, for *every* request that contains the broken history.

Terms used in the study. A *unit* is one GitHub issue together with its linked pull requests. A *cluster* groups units that share the same root cause. A unit is *confirmed* at one of two levels: C1 means a maintainer acknowledged the bug or a merged pull request references it; C2 means we reproduced it ourselves.

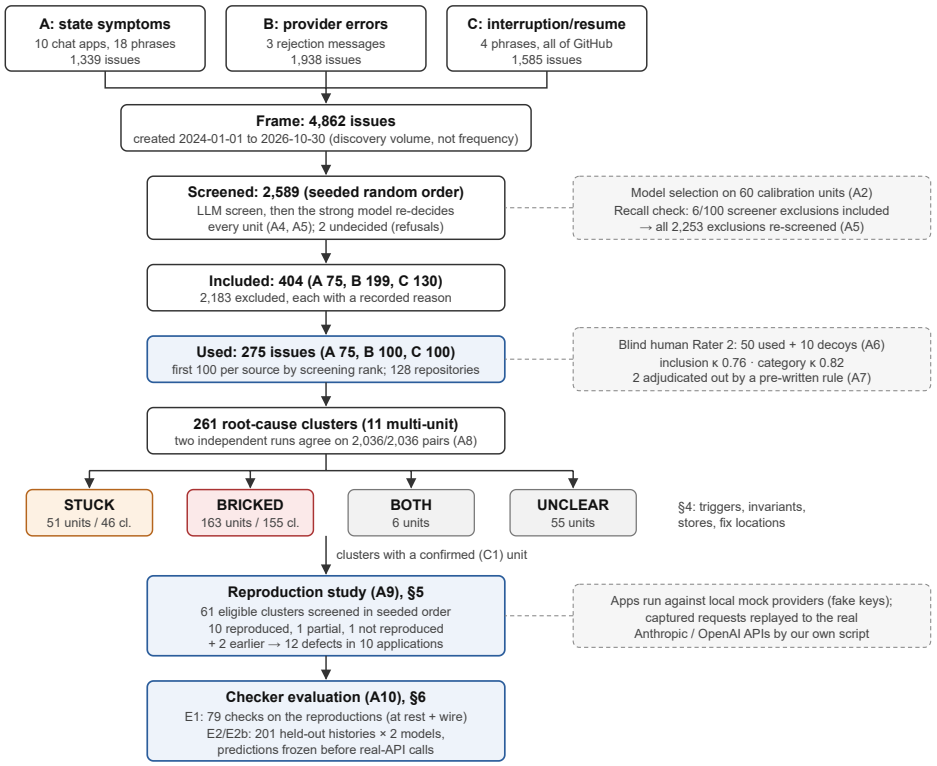

Fig. 1. Study flow. Solid boxes form the main pipeline; dashed boxes are validation steps. A*n* refers to protocol amendments.

3 STUDY DESIGN (RQ1, RQ2)

RQ1 and RQ2 rest on an issue study. We wrote the protocol down before classifying any issue and changed it only through dated amendments (A1–A13). We call a step *registered* if an amendment fixed it before its data existed; every change made after seeing results is marked post hoc (§10). Figure 1 shows the main steps for RQ1–RQ4; the search for new defects (RQ5) is in §7.

3.1 Where the Issues Come From

No single search finds these failures, because users describe them in different ways. We therefore searched GitHub in three ways, for issues created between 2024-01-01 and 2026-10-30:

Source A, symptoms: issues in 10 open-source LLM chat applications matching 18 symptom phrases that users write when a conversation hangs, e.g., “stuck generating”.

Source B, provider errors: issues quoting one of three provider error messages for a broken history: Anthropic’s unanswered `tool_use`, OpenAI’s unanswered `tool_calls`, and Anthropic’s `tool_result` without a call. Every issue here is about a rejected history, so Source B is bricked by construction.

Source C, interruption and resume: issues anywhere on GitHub matching four quoted interruption and resume phrases, e.g., `"INVALID_CHAT_HISTORY"`; most are in agent frameworks and coding agents.

GitHub search returns at most 1,000 results per query; only one Source C query exceeded it (996 of 1,661 retrieved). The searches found 4,862 issues (A 1,339; B 1,938; C 1,585). These counts show how much the searches found, not how often the failures occur.

3.2 Which Issues Count, and What We Record

An issue is included only if it describes both:

(a) a *saved* conversation, message, thread or checkpoint that is inconsistent or unusable afterwards; and
(b) how it got that way: an abnormal ending, a failure or resume path, or a saving defect on the normal path after which the stored history cannot be continued (A3).

We excluded, each with a recorded reason, issues that fail either test, display-only bugs, state that lives only in memory, feature requests, duplicates, and reports too vague to judge.

For each included issue we answer seven questions, D1–D7, and record how firmly the bug is confirmed (§2). Every answer is backed by quotes from the issue. **D1** Which kind of failure: stuck, bricked, both, or unclear (reported as such, never forced)? **D2** What triggered it? **D3** Which rule did the saved state break? **D4** Where was the bad state written: a server store, a local application store, browser storage, or unclear (recoded in A7)? **D5** What was the consequence for the user? **D6** Did anything recover it? **D7** Where was it fixed: write path, read path, both, or not at all?

3.3 Rating Pipeline and Its Validation

Rater 1: an LLM pipeline. A scripted LLM pipeline with fixed, versioned prompts screened and labelled the issues. We chose its models by testing, not by assumption. On 60 calibration issues, we picked, for each task, the cheapest model that met agreement thresholds set in advance against the strongest model. Screening went to `claude-haiku-5-5` (inclusion κ 0.80, one miss). Labelling went to `claude-opus-5-5`, because no cheaper model met the labelling threshold.

A check that failed, and what we did about it. To see what the cheaper screener missed, the strong model re-read 100 issues the screener had excluded. It included 6 of them (Wilson 95% CI 0.028–0.125 [13]), 4 of them stuck, which failed the threshold we had set in advance. We therefore re-screened all 2,253 exclusions with the strong model (A5). Every final decision now comes from one model and one prompt version, except two issues the strong model refused to process, which keep the screener's exclusion.

How many issues we used. Within each source, issues were put in a fixed random order (a seeded shuffle, not a model score), and the screener worked down that order until 100 issues per source were included. The re-screen then found more inclusions earlier in the order, so the 100th inclusion moved forward (B: rank 228; C: rank 663). The study uses the first 100 inclusions per source. Source A ran out at 76 (75 after adjudication, below). This leaves 129 included units beyond the cutoff (B 99, C 30). They were labelled like the others but are not part of the planned sample; we use them only for a robustness check (§4).

Rater 2: a human, working blind. To test the pipeline, the author, acting as a human rater, labelled a random sample without seeing any of Rater 1's answers (A6). The sample was 50 used issues plus 10 excluded ones mixed in as decoys, so the rater could not tell which issues had been included. The rater saw the same text and codebook. Table 1 reports agreement as Cohen's κ [2], where 1 means perfect agreement and 0 means agreement at chance level. On the key label, D1, the raters reached $\kappa = 0.82$, and on the 38 issues where both chose stuck or bricked, they agreed every time. *One label needed fixing:* D4 had a gap, because local CLI session files fit neither store category; we recoded it before changing any label, and report both versions. *Inclusion disagreements* were

Table 1. Agreement between the LLM pipeline (Rater 1) and the blind human rater (Rater 2).

Measure	Agreement	κ (bootstrap 95% CI)
Inclusion (60 units, 10 decoys)	0.933	0.76 (0.48–0.95)
D1 category (48 doubly included)	0.896	**0.82** (0.67–0.95)
D1, both raters definite (STUCK/BRICKED), $n = 38$	1.00	–
D2 trigger / D3 invariant / D7 fix location	–	0.88 / 0.87 / 0.86
D4 store, original / recoded (A7)	– / 0.854	0.41 / 0.75
D5 consequence / D6 recovery (exploratory)	–	0.67 / 0.51

Table 2. Violated invariant (D3), units (C1 units in parentheses).

D3	STUCK	BRICKED
terminal status not reached	33 (6)	0
output item left in progress	3 (1)	0
tool use without result	8 (4)	79 (24)
orphan tool result	0	29 (4)
role or order violation	0	20 (9)
duplicate or wrong id	0	6 (0)
stored content missing or wrong	7 (5)	23 (7)
other	0	6 (3)

settled by a rule written before it was used: the full thread decides, and a maintainer's statement wins. This removed two issues; the next Source B inclusion replaced one, leaving 275. All other labels are Rater 1's. *D5 and D6 are exploratory;* no claim rests on them.

3.4 Counting Root Causes, Not Reports

Several issues can report the same bug, so counting issues alone would give popular bugs extra weight. Within each repository, Rater 1 grouped issues by shared root cause ("one fix would resolve both", preferring to under-merge) in two independent runs. The two runs agreed on all 2,036 pairs of issues within repositories. A second pass proposed merges across repositories only where an issue names a shared upstream defect; we reviewed each of the 4 proposals and accepted 3. The result is 275 units in 261 clusters, 11 of them with more than one unit. We report units and clusters, with C1 counts, throughout.

4 WHAT BREAKS, WHERE, AND HOW IT IS FIXED (RQ1, RQ2)

The 275 units split into 163 bricked (155 clusters), 51 stuck (46 clusters), 6 that are both, and 55 unclear; 83 units are confirmed at level C1. In short: bricked conversations mostly break the tool-call rule, the same interruptions cause both kinds of failure, and fixes land mostly on the read path.

Because Source B was found through the providers' error messages, 99 of its 100 units are bricked. The proportions therefore describe what our searches found, not how common each failure is, and every comparison between the two classes uses sources A and C only (Table 3).

Finding 1: most bricked conversations break the tool-call rule, but not all. Over all sources, 134 of 163 bricked units (82%) break the pairing of a tool call and its result, or the order of messages (Table 2). That share is inflated by Source B, whose queries are the pairing error messages themselves:

Table 3. Triggers (D2), sources A and C only (source B is BRICKED by construction), units.

D2	STUCK ($n = 51$)	BRICKED ($n = 64$)
cancel/stop	9	12
tool interruption	7	12
restart/resume	9	9
normal-path serialization	3	7
truncation/malformed	2	6
upstream disconnect	6	1
provider switch/proxy	0	5
app internal error	4	0
timeout	2	0
trim/compact	1	3
other / provider error status / unclear	8	9

97 of its 99 bricked units are pairing or order violations. In sources A and C, the share is 37 of 64 (58%). The second-largest group there is missing or wrong stored content (21 of 64): the history is well formed, but what it holds is wrong. Stuck conversations are mostly records that never reach a finished status (33 of 51), plus 3 output items left in progress.

Finding 2: the same interruptions cause both kinds of failure. Cancel/stop, tool interruption and restart/resume are the most common triggers of both stuck and bricked conversations (Table 3). The same event can leave either a record that never finishes or a history the provider rejects. A few triggers lean towards one class: upstream disconnects (6 stuck vs. 1 bricked), internal errors (4 vs. 0) and timeouts (2 vs. 0) towards stuck, and provider switching and proxy translation, which re-encode the history into another provider's format (0 vs. 5), towards bricked. These counts are small, and no contrast survives correction for multiple comparisons (two-sided Fisher exact tests over all 13 triggers; smallest unadjusted $p = 0.036$, Holm-adjusted $p \geq 0.47$). We report them as hypotheses, not findings. Seven bricked units need no interruption at all: a serialization defect corrupts the history on the normal, successful path.

Where the bad state is written. Bricked state is written to server stores (55) and local application stores (54) about equally, with the store unclear in another 54 because Rater 1 did not infer it from the application type. Stuck state is written mostly to server stores (31 vs. 16). Most bricked units have no linked fix (116 of 163).

Finding 3: bricked conversations are mostly fixed when the history is read back, not when it is written. Among bricked units whose fix touches only one path, read-path fixes outnumber write-path fixes 32 to 13 (two-sided exact binomial test, $p = 0.007$). Source B does not explain the skew: in sources A and C it is 14 to 2 ($p = 0.004$). Among C1 units alone, the direction holds but the counts are too small for significance (17 vs. 9, $p = 0.17$). For stuck units the split is even (5 vs. 6). In these issues, fixes more often tolerate or repair a corrupted history when it is loaded than prevent it from being saved; §5 shows the consequences.

The exploratory labels (κ 0.67 and 0.51) add the user's view: bricked units most often report that the next message is rejected (119) and that recovery needed manual clearing (55). Automatic repair on replay is rarely reported (7).

A third shape of failure. The 55 unclear units (20%) are not noise. Of these, 41 concern stored content that is missing or wrong, and 39 report silent data loss (exploratory label). In these cases

the saved state is damaged, but the record is neither left unfinished nor rejected by the provider: a third shape of failure that our two-class codebook does not name. A single coder open-coded all 55 from their titles and evidence quotes, with AI assistance; this coding is not independently validated. It yields six groups: output lost on an abnormal ending, i.e., a reply stored empty or partial after a stop, disconnect or error (17); malformed or misplaced content that later breaks display or replay (12); history structure lost on reload or under concurrency (9); interrupted tool calls whose consequence the report does not state (7); compaction or summaries that corrupt what is kept (6); and accounting errors or work executed twice (4). One of the seven interrupted-tool-call units says that every later turn fails; it is probably a bricked unit that Rater 1 missed, and we report it rather than relabel it (A7). We keep the unclear units separate and leave a codebook for silent loss of saved state to future work.

The results hold on the larger set. Applied to all 404 included units, including the 129 beyond the planned cutoff, the main results do not change. The pairing/order share among bricked units in sources A and C is 48 of 80 (60%, vs. 58%), and the read/write fix skew is 16 vs. 3 ($p = 0.004$, vs. 14 vs. 2).

5 HOW THE BAD STATE GETS WRITTEN (RQ3)

Issue text rarely says which code wrote the bad record, so we reproduced failures on the applications' own code, before and after the fix, and inspected what each version saved.

5.1 Choosing and Running the Reproductions

Which failures we tried. Candidates were clusters with at least one confirmed (C1) unit labelled stuck, bricked or both, excluding the positive control. A fixed-seed shuffle set the order, and we screened in that order without skipping, so the choice of cases was not ours. A candidate was attempted only if it met five feasibility rules set in advance: runnable without Docker or a new native toolchain, a deterministic trigger, an identifiable pre-fix version, a scriptable model side, and no paid closed service. We took at most two per repository, aimed for at least three stuck cases, and stopped at ten reproduced.

What counts as reproduced. A defect is REPRODUCED only if the pre-fix version saves the bad state *and* shows the reported consequence, and, where a fix exists, the fixed version shows neither. At level *L1* we trigger the failure through the application's own code; at *L2* we place the issue's bad state into the real store, and the application's own code shows the consequence. An L1 reproduction runs at one of three scopes, marked in Table 4: the *whole application* (the default, e.g., the open-webui server driven through its API or UI); a *component*, where the defective middleware or plugin, copied unchanged from the pre-fix or fixed commit, runs inside a real framework graph with a persistent checkpointer; or a *library* called through its public API. At every scope the bad state is written by the project's own code; only the host around it is reduced. Each version ran twice.

The model side. Applications talked to local mock providers, compatible with OpenAI and Anthropic, using fake keys. The mocks enforce the pairing rules we observed on the real API. To confirm a rejection, our own script, built on the provider's official SDK, replayed the captured requests to the real API, so no third-party code ever held a real key.

Table 4. Reproductions (all L1 unless noted). The first two rows are the earlier positive control and pilot reproduction.

Issue	Class	Mechanism (persisted state)	Pre-fix → fixed
open-webui#29281 (control)	STUCK	cancel persists `done=true` with an output item `in_progress`; the UI masks it	0.11.3 stuck 4/4 → 0.11.4 `incomplete`
langgraph#6726	BRICKED	`CancelledError` bypasses `except Exception`; call persisted without result; next turn rejected (real API 400)	open; latest also affected
open-webui#29323	STUCK	a tool-argument crash leaves the reply `done=false` and loses output	0.11.3 → 0.11.4 completes
open-webui#19225	STUCK	a send-then-switch race stores the head message in the other chat without id/role; the chat shows "Loading…" forever	0.8.12 → 0.9.0 (fix commit unidentified)
kolu#1017 (L2)	STUCK	a dangling `tool_use` at the transcript tail pins the derived state at "running"	fix decays it to `waiting`
pi-cohort#7 (simulated event)	BRICKED	a control notice is persisted between call and result; pi-ai's synthetic result becomes a duplicate	real API 400 → 200
OpenHands-CLI#163	BRICKED	the `/resume <id>` text is persisted between `tool_use` and the reject observation	real API 400; reordered control 200; no fix exists
acp-kernel#286 (library)	BRICKED	the compaction fold re-projects call → summary → result	real API 400 → 200
deer-flow#2724 (component)	BRICKED	loop-detection middleware injects a warning between call and result	rejected on run and resume → clean
miaoli_bot#1 (component)	BRICKED	an attachment-fixer plugin flushes an image message between two results	rejected on run and resume → clean
preloop#1113 (L2)	BRICKED	the gateway rejects a client-stored cross-chat item (`function_call_output` with a `name` but no `call_id`) on every retry, although OpenAI's Responses API accepts it	gateway accepts after the fix
vercel/ai#9929	BRICKED	the stored web-fetch error result is dropped on replay	real API 400 → fix release throws client-side → latest OK
opencode#218	PARTIAL	a crash leaves a tool part `running` forever; replay already inserts a synthetic result, so no rejection	–
LinkX#119 (component)	NOT REPR.	the issue predicts a history slice that splits tool pairs, but the persisted transcript is replayed as text only in every version, so there is no pair to split; real OpenAI API 200	–

5.2 What We Reproduced

Table 4 lists every attempt. All 12 reproduced defects behaved the same in both runs. For langgraph#6726, pi-cohort#7, OpenHands-CLI#163, acp-kernel#286 and vercel/ai#9929, the real Anthropic API rejected the saved history. The two OpenAI cases (deer-flow, miaoli_bot) ran against the strict OpenAI-compatible mock, and their captured requests were then replayed to the real OpenAI API (`gpt-5-mini`): the pre-fix request failed with the issues' pairing error (400), the fixed one succeeded (200). For miaoli_bot, the replay replaces our harness's truncated PNG placeholder, which the real API rejects as an invalid image before any pairing check, with a valid 1×1 PNG; this is a post-hoc change.

5.3 What the Reproductions Show

Finding 4: most bricked conversations in our reproductions are broken by a message saved between a tool call and its result, and the writer is not the agent loop. In 5 of the 8 bricked reproductions, something is saved *between* a tool call and its result: a control notice, a user command, a loop warning, an attachment or a compaction summary (Figure 2). Each time, the writer is an extension, middleware, plugin, compaction kernel or command dispatcher, not the core agent loop, and none of them waited for an open call to close before appending. The Anthropic and Chat Completions protocols

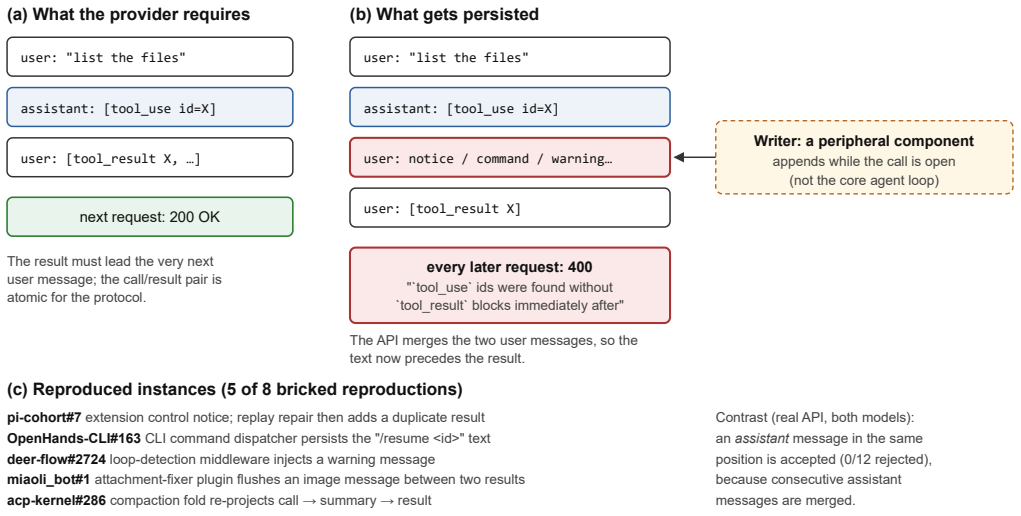

Fig. 2. The most frequent bricked-conversation mechanism among our reproductions, shown in Anthropic's format. (a) The provider requires a call's results to lead the next user message. (b) A peripheral component persists a message while the call is open; Anthropic merges consecutive user messages and rejects every later request (Chat Completions rejects it without merging). (c) The five reproduced instances, and a contrast case the real Anthropic API accepts (Chat Completions rejects it, §6.5).

treat a call and its result as one unit; OpenAI's Responses API does not (§6.5). These five were drawn in seeded order under feasibility rules, so this identifies a mechanism, not how common it is. The other three bricked cases are an error path that skips writing the result (langgraph#6726), a replay conversion that drops a stored result (vercel/ai#9929), and a reader stricter than the provider (preloop#1113, Finding 7).

Finding 5: code meant to repair the history can make it worse. In pi-cohort#7, pi-ai's replay-time repair adds a synthetic result for a call that looks orphaned; when the real result is merged in, the request holds two results for one call and is rejected. In miaoli_bot#1, the plugin that corrupts the history is one meant to *fix* tool attachments. In vercel/ai#9929, the first release with the fix throws an error in the client on the SDK's own saved error object, so stored conversations stay unusable on that release; the latest, `@ai-sdk/anthropic` 4.0.94, works, and we did not identify the release in between that fixed it. Issues outside our sample report the same pattern, e.g., openclaw#8264, "Synthetic error repair creates malformed tool_use/tool_result pairs".

Finding 6: stuck records are hidden when read, but stay wrong in the store. In open-webui#29281, the UI shows the reply as finished because of the message-level done flag, while one output item is still `in_progress`. In opencode#218, the replay path makes up a result for a tool part that stays `running`. In kolu#1017, the fix lets a derived "running" indicator decay, but the transcript keeps the dangling call. A fix on the read side clears the symptom for one reader and leaves the bad record for any other program that reads the same store without the same repair. This matches the read-path skew of fixes in Finding 3.

Finding 7: the program that writes the history and the program that reads it can be different. In preloop#1113, the Codex client stores and re-sends an item, and a gateway rejects it, although

OpenAI's Responses API accepts the same item (verified on both models, §6.5): the reader is stricter than the provider. In kolu#1017, Claude Code writes the transcript and kolu derives a stuck state from it. Neither side's tests exercise the other side's assumptions.

A rule only the real API revealed. In OpenHands-CLI#163, our Anthropic mock first accepted the captured history, because LiteLLM had merged the user text and the tool result into one message. The real API rejected it ("`tool_use` ids were found without `tool_result` blocks immediately after"), and accepted the same history with the text moved after the result. The result must therefore come *first* in the user message. We added this rule to the mock only after this evidence; a re-run then failed during the run, as on the real API.

6 CATCHING BAD STATE BEFORE IT IS SENT (RQ4)

We wrote the rules from §4 and §5 into a checker and asked two questions: does it flag the defects we reproduced, and does it agree with the real providers on histories it has never seen?

6.1 The Checker

The checker is a small offline program (Python standard library only) that looks in two places: *on the wire*, at the request body the application builds, and *at rest*, at the store as saved, through adapters for 7 store formats (open-webui, pi, Claude Code, OpenHands, LangGraph checkpoints, Vercel AI SDK and opencode; preloop already stores items in wire format). Each adapter is 7–22 lines of code (median 18, excluding comments and docstrings); the LangGraph adapter also needs a 17-line helper that runs in the application's environment. The rule engine is 166 lines. It checks three groups of rules:

Anthropic, AN1–AN6. After consecutive same-role messages are merged: each call is answered by the results at the start of the next message; each result matches a call in the message before; no result appears twice; results come first; each server-tool call is closed in the same message; error results have content.

OpenAI, OC1–OC3 and OR1–OR2. Pairing rules for Chat Completions and for Responses-API items. OC3 (no call answered twice) and the revised OR1–OR2 came out of the OpenAI tests below (§6.4, §6.5).

Lifecycle, LC1–LC3 (at rest only). A record that never finished; an unanswered call at the end; a broken pointer in a message tree.

The rules, the expected verdicts and the evaluation design were registered (A10) before the checker was written. Similar validators that check requests exist (§9); what we add is the evaluation against real providers.

6.2 E1: Does It Flag the Defects We Reproduced?

Across all runs of the 12 reproductions, 79 of 79 checks contain the registered expected rule, and 61 match exactly; the other 18 also fire a rule with the same cause, e.g., AN4 alongside AN1 when text comes before the result. Because these defects shaped the rules, E1 is a sanity check, not evidence that the checker generalizes. It does show where a check has to run. *Bad writes are visible at rest,* before any request is sent: in 5 of the 7 bricked cases with a message store, and in all stuck cases; the exceptions are vercel/ai and preloop, whose stored item the provider accepts (Finding 7). *Bad reads are not:* vercel/ai#9929 stores a valid history and corrupts it during replay, so only the wire check sees it. *Read-side fixes leave the store invalid* (kolu, opencode), and the checker rightly still flags these stores, because other readers are still exposed. Our registered expectation for preloop was wrong, because the provider accepts that item (§6.5); we corrected it and the OR rules post hoc, and E1 stays at 79/79.

Table 5. Agreement with the real Anthropic API (identical on both models). All checks had specificity 1.0. Wilson 95% CIs in brackets.

	E2 (frozen checker)	E2b (revised checker)
Accuracy	120/121 = 0.992 [0.955, 0.999]	80/80 = 1.00 [0.954, 1.00]
Sensitivity (API rejected)	81/82 = 0.988 [0.934, 0.998]	58/58 = 1.00 [0.938, 1.00]
Specificity (API accepted)	39/39 = 1.00 [0.910, 1.00]	22/22 = 1.00 [0.851, 1.00]
Baselines (sensitivity)		
LangGraph `_validate_chat_history` 1.2.18	34/82 = 0.415	22/58 = 0.379
Naive pairing (result anywhere)	35/82 = 0.427	22/58 = 0.379
turnsafe 0.1.0 (post hoc)	79/82 = 0.963 [0.898, 0.987]	57/58 = 0.983 [0.909, 0.997]

6.3 E2: Does It Agree With the Real Provider on New Histories?

Design (registered in advance). E2 asks the provider itself whether the rules are right. We built six valid seed histories (single call, parallel calls, sequential calls, error result, result plus text, server tool) and changed them with fifteen mutation operators. Some copy the mechanisms we observed, e.g., a user message between call and result, text before the result, a duplicated result. Others make shapes we expected to be *valid*, e.g., an inserted assistant message (which the API merges) or parallel results in reverse order. *E2* has 121 cases: the seeds, all 75 single mutations and 40 random double mutations. The checker's predictions, and a SHA-256 hash of the checker, were fixed before any API call. Every case went to `claude-haiku-4-5`, the model of the reproduction replays, and to the newer `claude-opus-5-5`, so that a model-specific quirk could not pass as a rule (`max_tokens` 16, no retries); a 200 response counts as valid, a 400 `invalid_request_error` as invalid. Any rule revised after E2 had to be re-tested on a fresh sample: *E2b*, 80 new random double mutations with a new seed and no overlap with E2.

Results. The checker agreed with the API on 120 of 121 E2 cases and all 80 E2b cases (Table 5), and the two models gave identical answers on all 201. The one E2 miss is a content rule, not a pairing rule: the API rejects an `is_error` result with empty content. It became AN6, which E2b tested three times, correctly each time. For every rejection, the API's error class matches the rule that fired.

Finding 8: the history check shipped with a major framework misses most rejected histories in our tests. LangGraph's prebuilt `create_react_agent` checks only that every tool call has a result *somewhere* (`_validate_chat_history`). That check passes interleaved messages, misplaced results and duplicates, exactly the mechanisms of Finding 4, and it passes the corrupted stored histories of deer-flow#2724, miaoli_bot#1, pi-cohort#7 and OpenHands-CLI#163. This agent API has been deprecated since LangGraph 1.0, and its successor, LangChain's `create_agent` (inspected at 1.2.10 and 1.2.15), has no comparable history check in its agent factory, so applications that migrate lose even this check. Whether an order-free check is wrong depends on the API, however: for OpenAI's Responses API it is nearly right (178 of 180, §6.5).

Finding 9: a validator that checks order closes most of the gap, but rules copied from documentation miss rules that only the provider's behaviour reveals. We added turnsafe [5], which our literature check found, as a baseline (not registered in advance). It encodes ordering, uniqueness and the same-role merge, and agrees with our frozen checker on every client-tool case. But it has no rule for server-executed tools or for empty `is_error` content, so it passes vercel/ai#9929's pre-fix request, which the real API rejects. We found our own E2 miss (empty `is_error` content) only by testing against the API, so our advantage on that rule in E2b is post hoc; on E2, the two checkers differ

only on server tools. History rules should therefore be tested against what the provider does, not only transcribed from its documentation and error messages.

6.4 E2-OAI: The Same Test Against OpenAI

We translated every E2/E2b case except the server-tool seed into Chat Completions format: 177 distinct histories. Consecutive same-role messages were left unmerged, because merging is one of the things under test. Predictions from the frozen OC rules were recorded before any call (A12). The models mirror the Anthropic pair: `gpt-5-mini`, a small previous-generation model, and `gpt-5.5`, the current flagship (a third chosen model, `gpt-4.1-nano`, had been retired and returned no data). The two models returned identical outcomes on all 177 cases. The frozen OC rules were correct on 161 of 177 (accuracy 0.91 [0.858, 0.944]), with sensitivity 124/140 = 0.886 [0.822, 0.928] and specificity 37/37 = 1.00 [0.906, 1.00]. All 16 misses are duplicate tool results: unlike AN3 on the Anthropic side, our OpenAI rules had no uniqueness rule. We added it as OC3 and tested it on a fresh registered sample (A13) of 68 new random double mutations, 13 of them with a duplicate result. OC1–OC3 matched the API on all 68 cases on both models, and the models agreed on all 68. Only four of these cases are valid, so the sample tests sensitivity far more than specificity. On E2-OAI, the order-free check (naive pairing, which equals LangGraph's check on these cases) has sensitivity 0.393.

6.5 E2-RESP: OpenAI's Responses API Plays by Different Rules

We re-expressed all 177 E2-OAI histories as Responses-API input items and added three cases shaped like preloop's (A13). The frozen OR rules did poorly: 113/180 correct on both models, with sensitivity 58/58 but specificity only 55/122 (0.45). The reason is that this API does not care about order: it rejects only a call that is never answered and an output for an unknown or empty `call_id`, and *accepts* a user message in between (0/6 rejected), a late output (0/6), an assistant message in between (0/6) and duplicate outputs (0/7). Our registered OR2 rule, "output before the next message", was therefore wrong for this API; a naive order-free check scores 178/180, missing only the two `call_id` cases. One registered preloop case left out the `name` field of the issue's item; the issue's exact item, sent post hoc, is accepted on both models. We revised OR1 and OR2 to match. They fit all 360 outcomes in-sample but have not been tested on a fresh sample. The two models agreed on all 180 cases.

What the providers accept and reject. The Anthropic API (both models) rejects a user message between call and result (12/12), text before the result (12/12) and a duplicate result (14/14). Because it merges consecutive same-role messages, it accepts three related shapes: an assistant message between call and result (0/12 rejected), results split across consecutive user messages (0/2), and a user message after the result message (0/12). It also accepts `[tool_use, text]` within one assistant message (0/14). The providers differ. OpenAI does not merge consecutive messages, so an assistant message between call and result is *accepted* by Anthropic (0/12 rejected) but *rejected* by OpenAI (6/6). OpenAI's Responses API goes further and accepts messages in between altogether, so the mechanism of Finding 4 bricks Anthropic and Chat Completions histories but not Responses-API histories. A history is valid only for a given provider and API, and a store replayed to more than one provider must meet the stricter rules. This may explain why provider switching appears as a trigger in §4.

7 USING THE MECHANISM TO FIND NEW DEFECTS (RQ5)

If Finding 4 describes a real mechanism, it should lead to previously unreported defects. We tested this with targets, probes and criteria fixed in advance.

7.1 Design (Registered in Advance as A11)

- *Where we looked.* The 128 repositories of the issue study plus the 30 most-starred results for each of the GitHub topics `llm-agent`, `ai-agent` and `llm-chat`: 212 repositories after deduplication, in a fixed-seed random order.
- *Which targets we took.* We screened in that order until 15 targets were accepted, at most two per organisation. A target had to be active (a commit in the last six months), keep conversations across restarts, support tool calls, offer at least one place to probe, be feasible under the reproduction rules of §5 (now also allowing PostgreSQL and Redis), and work with the Anthropic Messages API or Chat Completions. Targets that support only OpenAI's Responses API were left out, because that API accepts messages between call and result (§6.5). We screened 55 repositories, and every rejection is logged with its reason.
- *How we probed.* Each probe tries to put something into the history while a tool call is open, chosen per target before any run: a user message during a tool call (P1), a cancel during the tool followed by a reload (P2), an extension, hook or plugin message while a call is open (P3), compaction across a tool pair (P4), a SIGKILL during the tool followed by a resume (P5), and a provider switch (P6). Target-specific probes, such as a malformed tool call, were written into the target's plan before it ran.
- *How we decided.* The oracle is the checker of §6, frozen at the start (SHA-256 recorded), applied at rest and on the wire through our mocks, plus the application's own reload path. A defect is CONFIRMED only if it occurs in two of two runs on the latest release or default branch, and either (bricked) the captured request, replayed to the real provider by our own script, returns 400 while a minimal corrected version returns 200, or (stuck) an unfinished record survives a reload.
- *What counts as new.* A read-only search of the target's issues and pull requests for the mechanism, with the search terms recorded. Any match makes a defect KNOWN.

Where a target's UI could not be scripted, the harness drives the application's own session and loop code against our mocks (the component scope of §5), and each target's record states its scope. Runs spoiled by harness errors are listed and excluded.

7.2 What We Found

All 15 targets were completed; none was abandoned. We found six new defects in five targets (Table 6). The gptme defect needs gptme's documented but non-default native tool format for Claude models. We reported each defect upstream after confirming it[1]. At the time of writing, gptme has merged a fix, and the other reports are awaiting a maintainer response. We re-ran the three gptme triggers on the merged branch (2/2 each), and the real API accepted every replayed request, including resumes. As Finding 3 would predict, the fix is on the read side: the store still keeps the notice between the results, and the converter moves it later when building the request. Old conversations therefore resume, but their stored history stays invalid for readers without this repair.

The other ten targets had no new defect. Five known defects recurred: langgraph#6726 on LangGraph 1.2.19; three open aegra issues (a cancel during a tool call bricks the thread, a concurrent run ignores the multitask strategy, and a kill leaves the thread busy and bricked); and Vibe-Trading#2115, where system messages sent mid-run make the Anthropic client library raise an error, so the turn fails. Several targets resist by design, for example by writing a stand-in result

[1] https://github.com/langchain-ai/langgraph/issues/9556, https://github.com/gptme/gptme/issues/4763, https://github.com/chrisrothwell/thinkarr/issues/557, https://github.com/Miso2233/SonettoHere/issues/325, https://github.com/Miso2233/SonettoHere/issues/326, https://github.com/thetombrider/coding_agent/issues/422.

Table 6. New defects found in A11 (2/2 runs each). Real API: captured request (Orin: stored history) 400, corrected control 200, on the providers listed.

Target (version)	Mechanism (persisted state)	Class	Real API
LangGraph 1.2.19 (library)	when parallel tools return state updates, `ToolNode` appends a message added by one tool between the results of the same batch; its own command check tests only that a result exists	BRICKED	Anthropic, OpenAI
gptme 3950aeb (native tool format)	system notices (a directory change, interrupted output, lint warnings) stored inside a tool round are sent between the results by the Anthropic converter; an earlier fix covered only OpenAI	BRICKED	Anthropic
thinkarr 695575f (component)	the orphan repair persists its synthetic result *after* the new user message, so the conversation fails one turn later; a concurrent message yields a duplicate result	BRICKED	OpenAI
SonettoHere f428f81 (component)	parallel image tools: the LangGraph mechanism above, triggered by the application's own tools	BRICKED	OpenAI
SonettoHere f428f81 (component)	"undo last round" treats a tool's image message as the start of the round, so the question, call and result survive the undo	state	n/a
Orin 758f527 (component)	the argument-validation retry stores the tool call and a user correction but no result; the SDK rejects every later request; that a clean restart does not repair it was already reported upstream	BRICKED	Anthropic, OpenAI

on cancel (QwenPaw, gajae-code) or by holding the user's steering messages until the tool batch is done (R-Dson/pi).

Finding 10: the mechanism of Finding 4 leads to new defects, and some of the writers are recovery code. In all five new bricked defects, a message is saved while a tool call is still open. The writers are a framework's merge of updates returned by tools (LangGraph, and SonettoHere through it), notice and lint hooks (gptme), and two recovery paths: an orphan repair (thinkarr) and a retry after argument validation fails (Orin). The Orin retry sits inside the agent loop, so the "component outside the agent loop" of Finding 4 is better read as *any path other than the normal step from call to result.* Three of the five defects are in or next to an earlier fix: gptme's mechanism had been fixed for the OpenAI converter only (gptme#3421), thinkarr's defect is in the repair added for its issue #151, and Orin's retry path was widened by the fix for its issue #377, whose test uses a stub provider that skips message conversion. This extends Finding 5: code that repairs or fixes the history keeps producing new corruption.

Finding 11: the check has to run before the history reaches the SDK. On the wire, the frozen checker flagged exactly the failing requests for gptme (12 of 42 requests), thinkarr (12 of 26) and SonettoHere (6 of 24), and at rest it flagged the corrupted LangGraph checkpoints. In Orin, the Vercel AI SDK rejects the history inside the client (`MissingToolResultsError`), so no request is ever sent (for the real-API check we replayed the stored history instead); only a check at rest, or on the history handed to the SDK, can see it.

Silent loss is common. In 8 of the 15 targets, a killed or cancelled turn, or a queued message, left no trace in the stored history. The next session is valid, but the work is gone: the third shape of failure seen in the unclear issues (§4).

8 DISCUSSION: WHAT TO DO ABOUT IT

For application developers.

(1) *Save a tool call and its result as one unit.* Any component that adds to the history (notices, warnings, attachments, summaries, command echoes) must wait until open calls are closed, or add its message after the results (Findings 4 and 10).
(2) *Close the record on every exit path,* including exceptions that slip past generic handlers, such as Python's `CancelledError` (langgraph#6726).

(3) *Check when writing rather than repair when reading.* A repair on read hides the damage from one reader, can collide with other repairs, and leaves the store invalid for everyone else (Findings 3, 5 and 6).
(4) *Check in both places.* A check at rest catches a bad write before the next request; only a check on the outgoing history, before it reaches the SDK, catches damage done during replay (Finding 11).

For framework and provider designers. A framework's history check should follow the rules of the API it targets: order and uniqueness for Anthropic and Chat Completions, and, for OpenAI's Responses API, only that every call has an output and no output has an unknown `call_id` (Finding 8, §6.5). If providers published their history rules in machine-readable form, developers would not have to probe for what we had to test: merging, results first, error content, OpenAI duplicates and the Responses API's indifference to order.

For research. Bug taxonomies of LLM applications should separate failures of saved state, which make one failed request permanent, from transient ones.

9 RELATED WORK

Bug studies of LLM applications and agents. Zhang et al. [15] study Claude Code, Codex and Gemini CLI; their root-cause category "State & Context Management" covers 6.6% of bugs, and their symptom category "Session & State Management" 5.8%. Chen et al. [1] study agent-reactive bugs at the boundary between model and harness, including one case where a mismatch between calls and responses makes every later prompt fail. Studies of frameworks put related failures into buckets such as "Message Fault", "Serialization Error", "Memory Persistence Bug" and "Hang" [3, 16, 20]. In the two studies that report shares, state and session buckets hold about 3–7% of bugs [15, 20]. We study saved-state failures on their own, separate the record that never finishes from the history the provider rejects, and reproduce the mechanisms on real providers.

Gateways and serving infrastructure. FailureAtlas [9] catalogues failures of gateways that route requests to several LLM providers. Two of its entries touch conversation state: a race that silently drops turns, and malformed tool-call arguments saved into the client's history. Neither concerns the pairing of calls and results or records that never finish. ContinuityBench [8] forwards the full history when one provider fails over to another, without testing whether the history is valid for the new provider, although provider switching is a bricking trigger in our data.

Resume, checkpoint and deployment. The closest precedent for our method is *Built to Remember, Shipped to Forget* [7]. It derives contracts between LangGraph and container platforms and tests them with induced events. One contract is a run-level cousin of our stuck class ("the interrupted run remains 'in progress' and is not resumed"), but the work never looks at tool calls or at whether saved histories are valid for the provider. *Resume Means Resume* [4] model-checks checkpoint, interrupt and resume contracts for the persistence layers of workflow frameworks, including whether a checkpoint is valid, but against the framework's state schema, not the provider's history rules. ACRFence [18], recoverability primitives [17] and LIMBO [6] deal with tool side effects that are duplicated or rolled back. These works complement ours: a resumed conversation must be both *valid to send*, our concern, and *safe to re-execute*, theirs. AgentR [10] proposes a recovery-aware workflow architecture whose detection of orphaned jobs is the workflow counterpart of our stuck records.

Fault injection, exceptions and contracts. AgentChaos [11] injects faults into individual LLM API responses and measures task success, but does not ask whether the faulty content ends up in stored

state or in a resumed conversation. SHIELDA [19] handles agent exceptions with a taxonomy and a runtime framework. AgentFootprint [14] measures how much agents store and whether their history can be reconstructed, but not failures or validity for the provider. Weng et al. [12] find that "the presence of checks says little about their adequacy"; Finding 8 is a case of this for history checks.

History validators and repairers. turnsafe [5] validates and repairs OpenAI and Anthropic message lists with rules taken from documentation and error messages, and several frameworks ship their own repair logic. We do not claim a new validator. What we add is a registered evaluation of such rules against the real providers' answers, with turnsafe as a post-hoc baseline; evidence that checks need to run both on the saved store and on the outgoing request; lifecycle checks over application stores; and a measurement of how much a check shipped in a major framework misses.

Concurrent defect reporting. Shortly before our search began, the GitHub account swf1996120 had filed more than 250 issues across dozens of repositories. Three of them, all filed on 30 October 2026, concern two of our targets (gptme#4601 and #4602, Vibe-Trading#2115). Each says an automated tool found and reproduced it; we found no published description of the method. Our results overlap in one defect: Vibe-Trading#2115 is the turn-fatal error we observed, so we classify it as KNOWN. The two gptme reports concern other bugs on the Anthropic path (tool output sent as a system message; tool input written as a Python repr), and the fix for the first was already in the version we tested. Our search differs in three ways: it was registered in advance, its probes come from a mechanism found in an issue study, and each defect is confirmed against the real provider with a corrected control.

10 THREATS TO VALIDITY

Construct validity (do labels measure what we claim?). An issue is included only if its text describes saved state, and every label comes from that text, so we miss issues whose reporters never described the stored record. Ambiguous cases are reported as unclear rather than forced into a class; §4 shows that most of them describe a third shape of failure, silent loss of stored content. Issue text can also describe failures that never happen: one C1 unit (LinkX#119) is a planning ticket that predicts a split tool pair, and our reproduction showed that the split cannot occur on that code path (§5). The consequence (D5) and recovery (D6) labels reached only moderate agreement and are treated as exploratory.

Internal validity (other explanations). Rater 1 is an LLM. Five measures guard against its errors: models chosen by testing; a recall check whose failure led to a full re-screen; a blind human rater on a random sample with decoys; adjudication rules written before they were used; and agreement reported both before and after the D4 recode. The human rater was the author. Blinding to Rater 1's answers and the decoys reduce, but do not remove, the risk of assumptions shared with the codebook. Inclusion was adjudicated only for the 50 sampled units; outside the sample there is no second judgement, so we report the estimated false-inclusion rate instead of correcting for it: 2 of 50 (Wilson 95% CI 0.011–0.135). Component- and L2-level reproductions show the consequence of the reported state, not the application's full write path; Table 4 marks them. Mocks may differ from real providers, so every new or reproduced bricked consequence we claim for Anthropic was confirmed on the real API (the known aegra recurrences rely on the same shape, confirmed earlier), and one mock rule was added only after real-API evidence. In the other direction, the mock rejects results split across consecutive user messages, which the real API accepts, so the gptme re-check rests on the real API.

External validity (how far results generalize). Source B is bricked by construction, so class proportions are not estimates of how common each failure is, and comparisons between the classes use sources A and C only. Popular repositories and English-language issues dominate. The feasibility rules favoured applications that run without Docker, a new native toolchain or paid services, which tilts the reproductions towards Python and JavaScript applications with local file stores and in-process checkpointers. Failures that need a database or cache server, including concurrency across processes or machines, are under-represented: our one such candidate (LinkX#119) needs PostgreSQL and Redis as a full application, so we tested it only at component level, where it did not reproduce. The provider evaluation covers two providers with two models each. The revised Responses-API rules are verified only in-sample, and we did not test Bedrock, Vertex or other providers. E2/E2b are synthetic mutations of six seeds: they show that the rules match the provider on these shapes, not how often such histories occur in the field. Finally, the A11 search covers 15 targets screened from one seeded pool under the same feasibility rules, so its yield (new defects in 5 of 15) is not an estimate of how common such defects are. Two of its bricked defects share one mechanism (SonettoHere triggers the LangGraph defect), five targets were tested at component scope, and novelty was judged only from public issues and pull requests, so NEW means not publicly reported.

Conclusion validity (soundness of the statistics). E1 is circular, because the same defects shaped the rules. Small classes have wide intervals; for example, E2b's specificity of 22/22 has a lower bound of 0.85. The issue-study comparisons are descriptive: the trigger contrasts do not survive correction for multiple comparisons, and Finding 3 is significant over all units with a fix but not over C1 units alone. Seven elements were added post hoc: the turnsafe baseline, OC3 (later tested on a fresh sample), the revised OR rules, the corrected preloop expectation, the mock's results-first rule, the miaoli_bot image-placeholder replay and the stored-history replay for Orin. In A11, a bricked defect counts as confirmed only after two runs and a real-API replay, but the oracle cannot see defects our probes do not trigger, so a target without findings is not shown to be free of them. A11 also found one defect (2/2 runs), the SonettoHere undo defect, that breaks no provider rule, so the real-API criterion does not apply; we found it in the stored state, not with the checker.

11 CONCLUSION

Saving conversation state can turn a passing failure into a permanent one: a record that goes wrong once is read back wrong every time after. In our reproductions, three of the four stuck conversations came from a cancel, crash or interruption path that skips closing the record; the fourth came from a race between sending a message and switching chats. In five of the eight bricked reproductions, a component other than the agent loop saved a message while a tool call was still open, and in three, code meant to repair or fix the history itself went wrong. Across the issue study, bricked fixes lean towards the read side, which hides the problem from one reader and leaves it for the others.

On Anthropic, rules frozen before testing agreed with the API on 120 of 121 new histories, and on 80 of 80 after one revision; on OpenAI, rules taken from documentation missed duplicate results and the Responses API's indifference to order until tested against the API. A widely used framework check missed most of the rejected histories in our tests. Checks should follow the target API's rules and run both on the saved store and on the outgoing history before the SDK. The mechanism also led to five new defects that brick conversations, three in or next to earlier fixes. Recovery code needs the same care about order as the agent loop itself.

DATA AVAILABILITY

The replication package is available on Zenodo at https://doi.org/10.5281/zenodo.24268936. It contains the protocol with all amendments; the issue frame, screening and labelling outputs with request ids, the Rater 2 sheets and the clustering; the analysis scripts; the mock providers and harnesses; the checker code and the E1/E2/E2b, E2-OAI, A13 fresh-sample, E2-RESP and turnsafe-baseline data; and, for A11, the pool, screening log, per-target results and real-API replay scripts.

REFERENCES

[1] Jingyi Chen, Songqiang Chen, Hengcheng Zhu, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. 2026. Understanding Agent-Reactive Bugs at the Model-Harness Boundary: An Empirical Study of LLM Agent Issue Reports. arXiv:2607.15684

[2] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[3] Niful Islam, Ragib Shahriar Ayon, Deepak George Thomas, Shibbir Ahmed, and Mohammad Wardat. 2026. When Agents Fail: A Comprehensive Study of Bugs in LLM Agents with Automated Labeling. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2026)*. IEEE, Piscataway, NJ, USA. arXiv:2601.15232

[4] Sajjad Khan. 2026. Resume Means Resume: A Machine-Checked Conformance Contract for Checkpoint, Interrupt, and Resume Semantics in Workflow Persistence Layers. arXiv:2608.03836

[5] Yasir Khan. 2026. turnsafe 0.1.0: Validate, repair and trim LLM chat history without orphaning tool calls. https://pypi.org/project/turnsafe/. Released 2026-09-28.

[6] Jiapeng Li. 2026. Where Does Exactly-Once Live? Model, Harness, and Tool-Contract Effects on Duplicate Side Effects in LLM Agents. arXiv:2609.29095

[7] Meriem Mastouri, Emna Ksontini, and Amine Barrak. 2026. Built to Remember, Shipped to Forget: Operational Contracts, Failures, and Repairs in Containerized LLM Agent Deployments. arXiv:2610.34620

[8] Vishal Pandey and Gopal Singh. 2026. ContinuityBench: A Benchmark and Systems Study of Stateful Failover in Multi-Provider LLM Routing. arXiv:2607.15899

[9] Vishal Pandey and Gopal Singh. 2026. FailureAtlas: A Taxonomy of Failure Modes in Multi-Provider LLM Serving Infrastructure. arXiv:2607.17525

[10] Riya Samanta, Bidyut Saha, Soumya Kanti Ghosh, and Rajkumar Buyya. 2026. AgentR: A Stateful and Recovery-Aware Software Architecture for LLM-based Auditable Workflows. arXiv:2608.15264

[11] Gou Tan, Zhensu Sun, Jieke Shi, Ting Zhang, Zilong He, Qingfu Wu, Shuai Liang, Weifeng Sun, Junda He, Pengfei Chen, Chuanfu Zhang, Lwin Khin Shar, and David Lo. 2026. AgentChaos: Chaos Engineering for Agent Systems via Programmatic Fault Injection. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*. ACM, New York, NY, USA, 1780–1792. https://doi.org/10.1145/3832783.3837437 arXiv:2608.06790

[12] Shihao Weng, Longtian Wang, Yang Feng, Mingfei Cheng, Jia Liu, Chao Shen, He Ye, Federica Sarro, and Xiaofei Xie. 2026. When Code Becomes an LLM Call: An Empirical Study of Call-Site Contracts in Agentic Systems. arXiv:2610.24545

[13] Edwin B. Wilson. 1927. Probable Inference, the Law of Succession, and Statistical Inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212.

[14] Chenglin Yu, Hongquan Gui, Ying Yu, Tao Zeng, and Ming Li. 2026. The Hidden Footprint: Making Storage a First-Class Metric for LLM Agent Evaluation. arXiv:2607.11149

[15] Ruixin Zhang, Wuyang Dai, Hung Viet Pham, Gias Uddin, Jinqiu Yang, and Song Wang. 2026. Engineering Pitfalls in AI Coding Tools: An Empirical Study of Bugs in Claude Code, Codex, and Gemini CLI. In *Companion Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering (FSE Companion '26)*. ACM, New York, NY, USA, 393–403. https://doi.org/10.1145/3803437.3805213

[16] Xiaowen Zhang, Hannuo Zhang, and Shin Hwei Tan. 2026. Understanding Bugs in Modern Agentic Frameworks: A Study of Symptoms, Root Causes, and Triggering Conditions. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*. ACM, New York, NY, USA, 601–612. https://doi.org/10.1145/3832783.3834378 arXiv:2604.08906

[17] Zhihui Zhang and Wei Liu. 2026. Recoverability as a System Primitive for Long-Horizon AI Agents. arXiv:2609.13672

[18] Yusheng Zheng, Yiwei Yang, Wei Zhang, and Andi Quinn. 2026. ACRFence: Preventing Semantic Rollback Attacks in Agent Checkpoint-Restore. arXiv:2603.20625 CoDAIM Workshop 2026.

[19] Jingwen Zhou, Jieshan Chen, Qinghua Lu, Dehai Zhao, and Liming Zhu. 2025. SHIELDA: Structured Handling of Exceptions in LLM-Driven Agentic Workflows. arXiv:2508.07935

[20] Xinxue Zhu, Jiacong Wu, Xiaoyu Zhang, Tianlin Li, Yanzhou Mu, Juan Zhai, Chao Shen, Chunrong Fang, and Yang Liu. 2026. Bugs in Modern LLM Agent Frameworks: An Empirical Study. In *Companion Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering (FSE Companion '26)*. ACM, New York, NY, USA, 1588–1592. https://doi.org/10.1145/3803437.3805536