WHY DO SELF-EVOLVING LLM AGENTS FAIL? A STUDY OF MEMORY-BASED LEARNING AT TEST TIME

Anonymous authors
Paper under double-blind review

ABSTRACT

Memory-based self-evolving LLM agents promise continuous improvement by reusing experience without updating model weights. Yet a failure can originate in earlier learning, memory retrieval, or task execution, and outcomes alone cannot tell which. We introduce WTF (What's the Failure), a human-calibrated framework that diagnoses failures by linking task trajectories to memory artifacts and update histories, and tests selected diagnoses by replaying tasks from saved memory state. Across 9,544 traces spanning four self-evolution methods, five benchmarks, and three base LLMs, we derive a taxonomy of 25 failure classes. Most failures reflect persistent model, environment, or harness limitations, but 10.9% are memory/state-induced: a *memory tax*. This tax is heaviest in methods that inject their entire memory into every task, which show nearly three times the memory-induced share of methods that retrieve a few entries, and the gap widens as memory grows. Lessons from successful attempts can discourage useful actions, and correct knowledge can be missed at retrieval or applied beyond its scope. In controlled reruns, removing harmful guidance restores useful behavior without adding knowledge; across five selected cases, a memory-free agent raises pooled success from 1/18 to 13/18. Yet fixing a targeted behavior does not guarantee task success, and a fix for one task may not transfer. Reliable self-evolution thus requires effective learning, appropriate retrieval, grounded execution, and trustworthy evaluation. We will release code, memory artifacts, and task traces.

1 INTRODUCTION

Learning from experience is becoming a central goal for agents built on large language models (LLMs). Recent systems accumulate reusable workflows (Wang et al., 2024), strategy playbooks (Zhang et al., 2026b), lessons (Ouyang et al., 2025), and skills (Nous Research, 2026) that guide subsequent tasks; related approaches store verbal reflections (Shinn et al., 2023), extracted insights (Zhao et al., 2024), executable skill libraries (Wang et al., 2023), evolving cheatsheets (Suzgun et al., 2026), or optimized prompts (Agrawal et al., 2025). This *memory-based self-evolution* offers a practical route to adaptation at test time: agents update persistent state while model weights stay fixed, so each interaction can supply knowledge that helps avoid repeated mistakes. We focus on the reuse of experience across tasks, where earlier learning shapes later decisions.

Yet experience reuse can also undermine reliability. Repeated memory consolidation can erase earlier gains and fall below a memory-free baseline even when the underlying experiences are useful (Zhang et al., 2026a), and agents may disregard or misinterpret condensed experience (Zhao et al., 2026a). We therefore distinguish failing to benefit from experience from being harmed by it, and call the latter a *memory tax*: accumulated memory misguides or obstructs later task execution. This raises a central question: **Self-evolving LLM agents promise continuous improvement through experience, but do they reliably deliver? When they fail, why and how?**

Answering it requires connecting failures to the process through which experience becomes action. A wrong decision can originate in a lesson extracted from an earlier task, a later memory update, the retrieval of unsuitable guidance, or its application beyond its intended scope, and these mechanisms coexist with failures of the model, environment, and harness. Task scores cannot separate them, and the current trajectory may omit the memory history that explains them.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We introduce **WTF (What's the Failure)**, a human-calibrated framework that connects task trajectories with memory artifacts and updates, organizes the observed mechanisms into a taxonomy, and tests selected diagnoses through controlled reruns from saved state. We apply it to 9,544 task traces from 231 runs, spanning four self-evolution methods, five benchmarks in tool use, coding, and search, and three base LLMs. Our contributions are:

- **A framework for diagnosing failures across tasks and memory updates.** WTF calibrates automated analysts against human diagnoses on a trace subset, scales the analysis to the full corpus, and uses saved memory snapshots to rerun failed tasks with prior experience held fixed.
- **A taxonomy that separates memory, model, and system failures.** We identify 25 failure classes in three families. Of 4,979 classified failures, memory/state-induced mechanisms account for 10.9%, base-model failures for 74.7%, and environment or harness failures for 14.4%: failures of self-evolving agents extend well beyond memory itself.
- **Evidence of a memory tax.** Lessons from successful attempts can suppress useful actions, and stored procedures can exhaust execution budgets. In controlled reruns, removing harmful guidance restores useful behavior without adding knowledge, and across five selected cases a memory-free agent raises pooled full success from $1/18$ to $13/18$. Yet a targeted fix need not restore full task success, and guidance that repairs one task may fail to transfer.

We will release the framework, benchmark adapters, task traces, memory artifacts, and analysis outputs to support reproducible failure analysis and future mitigation research.

2 Related Work

Reliability of self-evolving agents. Recent work questions whether accumulating experience reliably helps: LLM-driven consolidation can corrupt useful memories and fall below a memory-free baseline (Zhang et al., 2026a), self-evolution can erode acquired capabilities as workflows grow longer and more defensive (Yu et al., 2026), agents follow retrieved experiences even when they mislead (Xiong et al., 2026) yet may disregard condensed experience (Zhao et al., 2026a), reflections can transfer negatively across task contexts (Huang et al., 2026), and memory and tool evolution can weaken safety (Shao et al., 2026; Zhao et al., 2026b). Each of these studies isolates one mechanism or measures it in a controlled setting; we instead attribute naturally occurring failures of existing methods to the pipeline stage that produced them, alongside base-model and harness failures, and test these diagnoses through controlled interventions.

Failure analysis of LLM agents. MAST (Cemri et al., 2025) builds a taxonomy of multi-agent failures, Zhu et al. (2025) one that includes memory errors within a trajectory, and other studies examine long-horizon execution (Wang et al., 2026a), agentic reinforcement learning (Wang et al., 2026b), and contextual distractors (Lee et al., 2026); these analyses largely treat each trajectory as a self-contained record of its failure. MemFail (Garg et al., 2026) and MemTrace (Deng et al., 2026) diagnose memory systems directly, but target factual memory, such as user information recalled in question answering. Self-evolving agents instead store procedural experience, so a memory-induced failure appears as a wrong action rather than a wrongly recalled fact and coexists in the same trace with base-model and harness failures. WTF therefore analyzes trajectories together with the memory history that shaped them, and its saved snapshots make counterfactual replay direct, which Deng et al. (2026) consider prohibitively costly.

3 WTF (What's The Failure)

WTF targets memory-based self-evolving agents, which improve across tasks by accumulating experience in persistent memory while keeping model weights fixed; we focus on cross-task memory rather than within-task scratchpads. Figure 1a abstracts the process into four stages: knowledge extraction and state update turn a task's trace and feedback into memory, and context retrieval and grounded action apply it to later tasks. Because a failure can enter at any stage and surface several tasks later, WTF analyzes each failed trajectory together with the memory history that shaped it, scales this analysis with automated analysts calibrated against human diagnoses, and tests selected diagnoses by replaying tasks from saved memory state (Figure 1b).

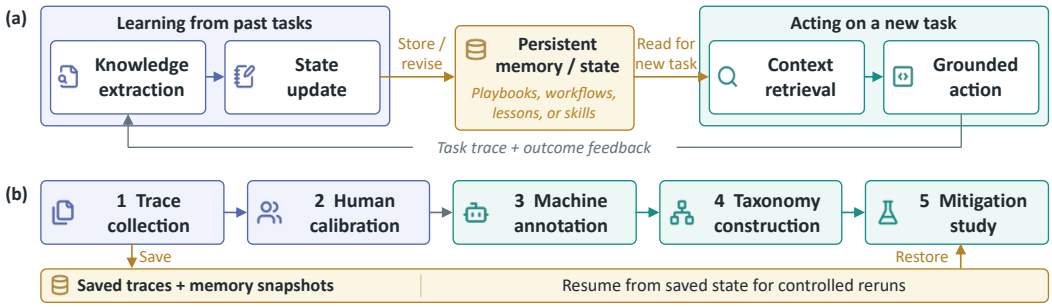

Figure 1: **(a) A common pipeline for memory-based self-evolution; (b) the WTF workflow.** (a) Learning from past tasks turns experience into persistent memory; acting on a new task retrieves and applies it. We use these four stages to locate memory/state-induced failures (Figure 2). (b) Human-agent reconciliation on a calibration subset guides automated annotation of the full corpus; agents aggregate the annotations into a taxonomy reviewed by two researchers; saved snapshots support controlled reruns.

3.1 Trace Collection

Methods, benchmarks, and models. We study ACE (Zhang et al., 2026b), Agent Workflow Memory (AWM) (Wang et al., 2024), ReasoningBank (Ouyang et al., 2025), and Hermes Agent (Nous Research, 2026), which maintain playbooks, workflows, lessons, and skills, respectively. We use the authors' official implementations and add benchmark adapters where a method does not natively support a benchmark. Tasks span tool use (AppWorld (Trivedi et al., 2024) and τ^2-bench (Barres et al., 2025)), coding (SWE-bench Lite (Jimenez et al., 2024) and Terminal-Bench (Merrill et al., 2026)), and search (BrowseComp+ (Chen et al., 2025)), and agents run on GPT-5.4, Claude Sonnet 4.6, and DeepSeek V4 Pro.

Runs, traces, and failures. A *run* is one ordered task stream processed under a fixed configuration, with memory carried across tasks; a *task trace* records one task execution. We collect 9,544 traces from 231 runs (Table 1) in two collections: a smaller collection of 4,526 traces from 60 runs, which supplies the human calibration subset, and a larger collection of 5,018 traces from 171 runs. A trace fails if its recorded outcome is not a success: its benchmark score is below 1.0, so partial credit counts as failure; its coding task is unresolved or could not be evaluated by the harness (260 SWE-bench Lite traces); or the agent exceeded its time limit even though the verifier passed (13 Terminal-Bench traces; §4.3). This yields 5,080 failed traces, all of which the automated analysts diagnose; mapping the diagnoses to the codebook assigns 4,979 to a failure class and the remaining 101 to X1, for diagnoses too inconclusive to code (§3.3). Counts exclude duplicate exports and memory-free runs, which serve only as references in §5; because tasks recur across runs, counts measure executions rather than unique tasks.

State-aware logging and restartable checkpoints. Each trace record stores the retrieved context, execution trajectory, evaluation result, reflection, and a serialized snapshot of the memory state. Pairing each trajectory with the state that shaped it lets us trace a failure back to extraction, update, or retrieval on earlier tasks. Snapshots also serve as restart points: an intervention can edit memory and resume the stream without regenerating the preceding history (§5). These records are long: the median failed trace spans 12 agent steps and roughly 20K tokens (90th percentile: 123K), and by the end of a run the median memory state holds roughly 8K tokens, with the largest ACE and Hermes states approaching the 50K-token limit (Appendix G). A failure's cause can thus lie far from its symptom in a record no single context window holds, motivating the agentic inspection in §3.2.

3.2 Human-Calibrated Automated Failure Analysis

Human diagnosis. Four annotators independently diagnose a calibration subset of 677 failed traces from 17 runs of the smaller collection, stratified by method, benchmark, and base model. For each trace, they describe the outcome, identify the failure mechanism, and judge whether experience reuse caused or amplified it, including whether the behavior recurs across the task stream. Each diagnosis must cite concrete evidence, such as action steps or memory entries, and state un-

Table 1: **Collected traces and failures.** Each trace is one task execution; a trace fails if its outcome is not a success, and coded failures exclude the 101 uncodable (X1) cases. Memory-free runs and duplicate exports are excluded; Appendix B gives the full coverage matrix.

		Runs	Traces	Failed	Coded failures
Method	ACE (playbook)	56	2,106	1,107	1,093
	AWM (workflows)	65	2,828	1,505	1,486
	ReasoningBank (lessons)	62	2,738	1,465	1,425
	Hermes Agent (skills)	48	1,872	1,003	975
Benchmark	AppWorld	51	1,856	305	304
	τ^2-bench	60	2,640	1,343	1,305
	SWE-bench Lite	44	2,048	1,488	1,470
	Terminal-Bench	24	1,080	492	468
	BrowseComp+	52	1,920	1,452	1,432
Total		**231**	**9,544**	**5,080**	**4,979**

certainty where the record is inconclusive. Annotators may reason about how a memory-free agent would behave; §5.1 tests this judgment experimentally for selected cases. Annotators do not use LLMs to draft diagnoses; Appendix H gives the full instructions.

Reconciliation. Automated analysts, Claude Code agents running Claude Sonnet 5, then diagnose the same traces. Annotators compare the two sets of diagnoses and revisit each disagreement against the trace evidence until it is resolved. This calibrates evidence standards and causal interpretation, not just labels.

Automated analysis at scale. The reconciled diagnoses then serve as in-context examples for the same analysts, which inspect each remaining failed trace with file access and programmatic tools. Because a failure's cause may lie in earlier tasks and records often exceed a single context window, analysts navigate the records rather than reading them whole, following retrieved context, reflections, and memory snapshots back through the stream. Each analysis produces a structured record: a free-text diagnosis, cited evidence, a causal attribution (base model, memory/state, environment, mixed, or uncertain), and a confidence level (Appendix H).

3.3 Taxonomy, Reliability, and Controlled Interventions

Taxonomy construction. Claude Sonnet 5 (via Claude Code) clusters 787 free-form failure-mode descriptions from 84 run-level analysis reports, together with the human calibration annotations and five open-coded hold-out runs, into recurring mechanisms grouped by pipeline stage. Two researchers review each candidate against its supporting evidence, merging, splitting, or removing classes the evidence does not support. The final codebook contains 25 classes in three families and ten subcategories: memory/state-induced, base-model, and environment or harness failures (Appendix A). Claude Sonnet 5 then maps each trace-level diagnosis to one primary class from this fixed codebook, without re-deriving it from the trace.

Evidence requirements. A method's general vulnerability is not evidence that it caused a particular failure. Irrelevant retrieved memory, for example, does not justify a memory-induced label unless the trace links it to the failing behavior. Causal attribution is recorded separately from the primary class, so mixed or uncertain explanations remain visible: 67% of memory/state-induced and 6% of base-model failures carry a mixed or uncertain attribution (§6).

Reliability. We check the automated labels at two levels. Re-coding every diagnosis against the same codebook with Claude Opus 5.5 reproduces family shares within 0.2 and class shares within 1.4 percentage points (Spearman $\rho = 0.99$; Appendix C), which tests the coding step. For selected memory/state-induced failures, memory-free reruns test the label experimentally (§5.1; Appendix D). Trace-level diagnoses are not individually audited, and Claude Sonnet 4.6 traces share a model family with the analysts (§6).

Controlled interventions. To test selected diagnoses, we restore the pre-task memory snapshot, disable memory updates, and rerun the failed task under a control condition, which keeps the original memory, and an intervention condition, which edits memory, changes retrieval, or adds a warning. Model, task, and harness are held fixed, and each condition is run three to five times; we interpret interventions only relative to their control reruns. For memory/state-induced failures, a memory-free condition replaces the memory section with the empty playbook template (section headings only) used by our historical memory-free baselines. This makes the memory tax measurable for a given task: memory imposes a tax when the memory-free agent succeeds more often than the agent with its accumulated memory. We record both whether the targeted behavior occurs, judged by reading the rerun transcripts, and whether the task succeeds. Interventions may use hindsight to identify harmful memory or supply guidance, so they test mechanisms in selected cases; they do not evaluate an autonomous mitigation method or estimate effectiveness across the corpus.

4 The Failure Taxonomy for Self-Evolving LLM Agents

We organize the 4,979 coded failures into three families; Appendix A gives full class definitions. Base-model failures dominate at 74.7%, followed by environment and harness failures at 14.4% and memory/state-induced failures at 10.9% (Figures 2 and 3). Measured per execution rather than per failure, memory/state-induced mechanisms account for the failure of 5.7% of all 9,544 task executions. These shares describe which mechanisms produce failures; they do not measure how self-evolution changes the overall failure rate, which §5 probes with memory-free reruns.

Memory-induced failures concentrate on τ^2-bench. The memory/state-induced share varies widely across benchmarks, from 2.4% of coded failures on SWE-bench Lite to 27.7% on τ^2-bench (Figure 4). Shares of failures can mislead, however, because benchmarks fail at very different rates: 16% of AppWorld executions fail, compared with 76% on BrowseComp+. Per execution, AppWorld and BrowseComp+ lose similar fractions to memory/state-induced failures (3.3% and 3.8%), whereas τ^2-bench stands apart at 13.7% and alone contributes about two-thirds of all memory/state-induced failures. Its tasks follow domain policies through multi-turn procedures under a 30-step budget, so stored procedures are often applicable and costly when wrong. Consistent with this, misleading memory and procedural bloat dominate its memory/state-induced failures (Appendix F), and §5.1 finds retrieved workflows pushing agents toward unnecessary escalation. Across all benchmarks, these two classes account for 79.8% of memory/state-induced failures.

Base-model failures dominate everywhere. Base-model failures form the largest share on every benchmark. Three classes alone, evidence retrieval failure (B11), wrong localization or fix (B9), and loop or stall (B12), account for 42.5% of all coded failures. Appendix F breaks down the distribution by method and base model.

4.1 Memory/State-Induced Failures

Stored experience imposes a memory tax when it suppresses useful actions, consumes the budget needed to finish, or overrides current task requirements.

> 🔍 **Finding 1: Learning can backfire.** Lessons extracted from successful attempts can discourage useful actions, while stored procedures can consume the budget needed to finish.

In an ACE payment task on AppWorld, the agent succeeded after trying three saved cards; reflection criticized the retries, and the playbook came to recommend a single best card. On the next task, the agent stopped after the first decline with two cards untried (A1), whereas without memory it keeps trying and pays in all three reruns (§5.1; Appendix E.2). Stored procedures can also harm through their cost: in a Hermes telecom task, a skill required a second diagnostic check although the user's status already revealed the problem, and the agent requested the correct fix only after the 30-turn limit (A5; Appendix E.5). Each step is reasonable on its own; together they leave no budget to finish, echoing the drift toward longer, more defensive workflows observed by Yu et al. (2026).

> 🔍 **Finding 2: Correct memory is not enough.** Useful lessons may not be retrieved, and valid procedures can cause errors when applied beyond their intended scope.

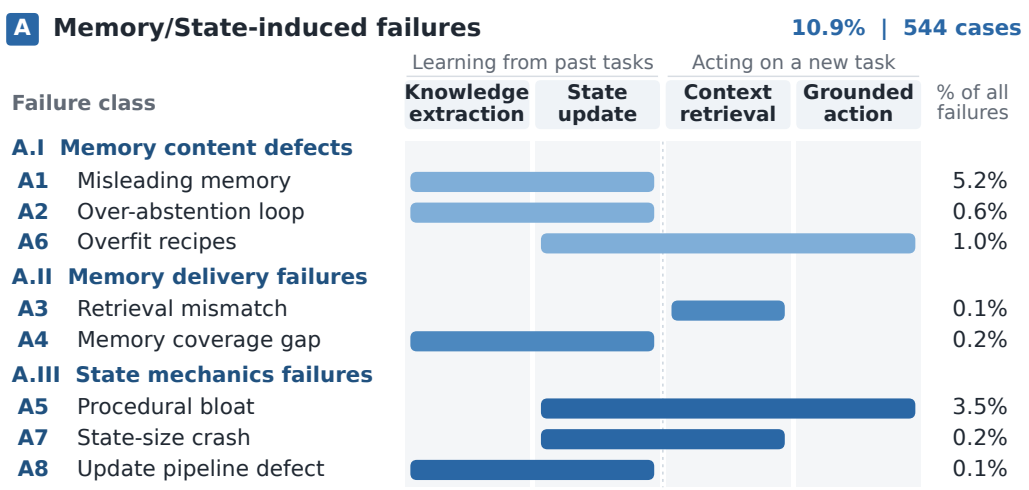

Figure 2: **Memory/state-induced failures.** Shaded spans indicate the pipeline stages where each failure mechanism operates. Percentages are shares of all 4,979 coded failures. A4, A7, and A8 are rare and vanish under an alternative coding model (Appendix C), so we do not interpret them.

In an AppWorld purchase task, ReasoningBank held a lesson about clearing unwanted cart items before checkout. Retrieval instead selected an entry with a nearly identical source-task description whose guidance concerned payment-card declines. The agent followed that guidance and ordered the pre-existing cart contents along with the requested items (A3). The safeguard had been learned but was not delivered when needed.

Delivery alone is also insufficient. In another ReasoningBank task, a recipe correctly described how to choose the cheaper of native gift wrapping and separately purchased wrapping paper. The current task explicitly requested gift-wrapped items. After correctly applying native wrapping, the agent followed the cost-saving recipe, removed the wrapping, and bought paper instead (A6). The recipe stated its scope, yet the agent applied it beyond that scope and undid work that already satisfied the request. Both mechanisms are rare as coded (0.1% and 1.0% of failures), but they show that topical similarity and correctness on a source task do not make a lesson appropriate for the current decision.

4.2 Base-Model Failures

Finding 3: Execution bottlenecks persist. Memory-based agents can fail to gather or use task evidence even before relevant lessons are available to guide them.

On its first BrowseComp+ task, before any memory existed, Hermes issued 83 searches but largely ignored the birth-year clues that would have narrowed the candidate set (B11). Evidence retrieval failure is the largest single class (17.0%), and it reflects a weakness that self-evolution must overcome rather than one it introduces.

Attribution must also respect update timing: in an ACE code-repair task, the playbook rule naming the correct repair file was created from that very failure (B9), so reading the trace against the final playbook would misattribute it to ignored memory.

4.3 Environment and Harness Failures

Finding 4: Failure feedback can be ambiguous. An episode can fail despite producing a correct artifact, so its outcome alone does not identify which behavior should change.

In an AWM Terminal-Bench task, the artifact passed the verifier, but the episode was recorded as a failure because the agent process exceeded its time limit (C3). The log therefore contains both `reward=1.0` and `outcome=failure`. The same disagreement appears in 13 of the 1,080 Terminal-Bench traces (AWM 6, ReasoningBank 4, ACE 3), each an agent timeout recorded after

the verifier had passed. Evaluation faults also arise in parsing: on BrowseComp+, the harness recorded failure whenever the judge formatted a correct verdict as JSON, and rescoring the 1,213 stored judge replies flips 14 task scores from failure to success and none in reverse (Appendix E.1).

This ambiguity matters because failures are learning signals: environment and harness failures make up one in seven coded failures (14.4%), and a method that learns from them risks discouraging behavior that was not at fault, such as the procedure that produced a correct artifact here. Reliable lesson extraction therefore requires identifying what failed before deciding what to change.

5 Understanding Failures Through Controlled Interventions

We now apply the protocol of §3.3 to selected failed tasks, asking which failures change with memory or context and which persist. §5.1 edits memory content and retrieval for Family A failures; §5.2 tests whether failure-aware warnings prevent recurrence and transfer to a new task. Throughout, we separate recovery of the targeted behavior from full task success, which distinguishes limits on the information an agent receives from failures to act on it.

5.1 Mitigating Memory/State-Induced Failures

Table 2 reports five Family A cases, each replayed from its pre-task snapshot with a targeted intervention and a memory-free reference. In all five, the memory-free agent succeeds at least as often as the control: pooled full success rises from $1/18$ to $13/18$ (one-sided exact test stratified by task, $p \approx 3 \times 10^{-5}$). Individual comparisons are not significant at three to six reruns per condition, so we rely on the consistent direction together with the mechanisms visible in the transcripts; Appendix D reports all 14 memory-free comparisons, including one reversal.

Table 2: **Behavioral recovery and task success under memory/state interventions, with a memory-free reference.** Entries report successful reruns as control → intervention; the *No mem.* columns replace the memory section with the empty playbook template used by our memory-free baselines, holding the model route, prompt template, and harness fixed. Each task has three reruns per condition; the telecom row pools two tasks. Behavior achieved refers to the listed target; full task success requires a benchmark score of 1.0. Memory updates are disabled throughout. †All memory-free reruns clear the user's pre-existing cart without restoring it; the grader does not check cart state. Only the corrective-lesson reruns restore it ($3/3$). ‡The one memory-free failure ships orders to the invitees' addresses rather than the user's home, an error unrelated to the removed recipe.

			Behavior achieved		**Full task success**	
Failure class	**Intervention**	**Targeted behavior**	**Control → Interv.**	**No mem.**	**Control → Interv.**	**No mem.**
A1: Misleading memory	Remove misleading card-selection rule	Try another card and send the payment	$0/3 \rightarrow \mathbf{3/3}$	$3/3$	$0/3 \rightarrow \mathbf{2/3}$	$3/3$
A2: Over-abstention loop	Soften the skill's cautionary footer	Provide an answer instead of abstaining	$1/3 \rightarrow 0/3$	$3/3$	$1/3 \rightarrow 0/3$	$3/3$
A3: Retrieval mismatch (AppWorld)	Select an existing corrective lesson	Preserve existing wish-list state	$0/3 \rightarrow \mathbf{3/3}$	$3/3$	$0/3 \rightarrow \mathbf{3/3}$	$3/3^{\dagger}$
A3: Retrieval mismatch (τ^2-bench telecom)	Include the stored Data Saver workflow	Turn off Data Saver	$2/6 \rightarrow \mathbf{4/6}$	$6/6$	$0/6 \rightarrow 0/6$	$2/6$
A6: Overfit recipes	Remove the overfit RSVP recipe	Identify gift recipients from phone contacts	$0/3 \rightarrow \mathbf{3/3}$	$3/3$	$0/3 \rightarrow \mathbf{3/3}$	$2/3^{\ddagger}$

Insight 1: Recovery need not require new knowledge. Removing harmful memory can restore memory-free behavior without adding a corrective lesson or updating model weights.

In the ACE payment case (A1), deleting the rule that discourages card retries restores payment in all three reruns, and the memory-free agent behaves the same way; the single removed rule therefore accounts for the failure. A retrieval intervention shows the same effect in ReasoningBank (A3): retrieved cart-handling instructions cause the agent to alter unrelated wish-list state. Selecting an existing corrective lesson restores full success, but deleting the harmful instructions and removing memory entirely also succeed in all three reruns, so removing harmful guidance is sufficient here.

The corrective lesson's one benefit, restoring the user's pre-existing cart, is invisible to the grader (Table 2†), echoing Finding 4.

> **Insight 2: Small edits may leave harmful assumptions unchanged.** Repair may require removing a whole procedure when several instructions reinforce the same mistaken assumption.

In the ReasoningBank gift-purchasing case (A6), removing an example email subject changes the search terms but leaves the agent searching for nonexistent RSVPs, whereas removing the entire RSVP recipe lets it identify recipients from phone contacts and complete the task, matching the memory-free agent's behavior, which never searches for RSVPs. The error is a shared assumption that invitees must be inferred from email replies. A second A6 case with ACE shows the same pattern (Appendix D): removing one of three reinforcing playbook bullets leaves success at $1/3$, whereas removing all three or all memory reaches $3/3$. In Hermes over-abstention (A2), softening the skill's final cautionary sentence yields no correct answers, whereas the memory-free agent answers correctly in all three reruns (up from $1/3$) and retrieves the same key interview with roughly half as many searches; memory thus affects answer commitment as well as evidence gathering, although this condition does not isolate a single responsible rule.

> **Insight 3: Memory-induced harms can stack.** Correcting one harmful behavior can leave another in place, and execution limits can still prevent task success.

In two AWM telecom tasks (A3), retrieving a stored Data Saver workflow raises Data Saver deactivation from $2/6$ to $4/6$ reruns, yet full task success stays at $0/6$. The memory-free condition locates a second memory tax: without memory, the agent turns off Data Saver in all six reruns and never escalates, whereas 7 of the 12 reruns with retrieved roaming, VPN, and network workflows transfer the user to a human agent. The targeted fix repairs the missing diagnosis but not the escalation, and even the memory-free agent succeeds in only $2/6$ reruns under the 30-step limit (Appendix D). An unchanged score can hide both a behavioral improvement and a second memory-induced failure.

These cases are selected for reproducible failures: of six A3 cases considered, only one showed improved task scores, and in others the original failure did not recur under control, so the pooled memory-free comparison is conditional on this selection. Memory is also not uniformly harmful: in an ACE A2 case, the playbook succeeds in $3/3$ reruns versus $1/3$ without memory (Appendix D). Across Table 2, no targeted intervention outperforms the memory-free agent except in A6 ($3/3$ vs. $2/3$), where the single memory-free failure is unrelated to the removed recipe (Table 2, ‡).

5.2 Failure-Aware Mitigation and Its Limits

We next study **whether explicit guidance derived from a diagnosed failure can prevent its recurrence and transfer to another task**. For four failure classes, we append a warning describing the error and corrective rules to the original context. Each warning is derived from a failed attempt on a source task and evaluated both on that task and, unchanged, on a second task with the same failure code. Controls replay the original context without the warning. Source-task reruns therefore probe recovery with hindsight, while the second task probes transfer. Table 3 reports full task success.

> **Insight 4: Concrete corrective instructions can restore task success.** Rules about actions or output formats can correct selected failures attributed to either memory or the base model.

Some failures respond to an explicit rule about a required action or output format. In two AppWorld tasks (B10), agents submit numerical amounts as currency-formatted strings. A warning to pass a raw number restores full success in all three reruns of each task, transferring from an AWM task to a ReasoningBank task. In the AppWorld retrieval-mismatch case (A3) studied in §5.1, a warning about cart handling, state restoration, and verification restores full success in all five source-task reruns.

> **Insight 5: Recovery on one task does not establish transfer.** A warning may correct a known mistake while leaving task-specific reasoning unresolved on a new task.

Transfer is less consistent when guidance leaves task-specific decisions unresolved. Applied unchanged to an AWM telecom task, the A3 warning raises full success from $0/3$ to $2/3$, plausibly

Table 3: **Recovery and transfer under failure-aware warnings.** Entries report successful reruns as control → warning. Each warning is derived from a failure on the source task and applied unchanged to a second task with the same annotated failure code. For A3, transfer is from an AppWorld purchase task to a τ^2-bench telecom task. Full task success requires a benchmark score of 1.0 or a resolved coding task.

			Full task success	
Failure class	**Intervention**	**Targeted behavior on source task**	**Source task**	**Transfer task**
A3: Retrieval mismatch	Warn against using the wish list as scratch space	Clear the cart and restore unrelated state	$0/5 \rightarrow \mathbf{5/5}$	$0/3 \rightarrow \mathbf{2/3}$
B9: Wrong localization or fix	Warn against symptom-based repair localization	Trace the call path and identify the repair location	$0/3 \rightarrow 0/3$	$0/3 \rightarrow 0/3$
B10: Malformed output artifact	Warn against formatted numerical answers	Pass a raw number to the completion tool	$0/3 \rightarrow \mathbf{3/3}$	$0/3 \rightarrow \mathbf{3/3}$
B11: Evidence retrieval failure	Warn against overly constrained search queries	Search a discriminative clue and retrieve relevant evidence	$0/3 \rightarrow \mathbf{2/3}$	$0/3 \rightarrow 0/3$

through its general completion and verification rules. In BrowseComp+ (B11), a warning against combining all clues in one query restores two of three source-task reruns but none on the transfer task, and in SWE-bench (B9), guidance to trace the call path yields no successful repair on either task. Unlike the B10 rule, these warnings leave the decisive search or localization choice to the agent, suggesting that transfer depends on whether guidance specifies an actionable correction.

6 Discussion

Memory delivery shapes the memory tax. Failure rates are nearly identical across methods (52.6–53.6%), yet memory/state-induced failures make up 17.3% and 17.5% of coded failures for ACE and Hermes, which inject their whole store into every task, against 7.7% and 5.0% for ReasoningBank and AWM, which retrieve a few entries (Appendix F). Wholesale injection keeps stored lessons in view, but it also carries cross-domain rules and accumulated checklists into every task, and its tax grows with the store: in single-domain streams, memory/state-induced failures rise from about 5% of executions in the first third to 9–11% in the last for ACE and Hermes, while ReasoningBank and AWM stay below 4% (Appendix F.1). Deciding how much experience reaches the agent thus matters as much as deciding what to store.

Memory is usually one cause among several. Of memory/state-induced failures, 67% carry a mixed or uncertain attribution, against 6% of base-model failures; the 10.9% share therefore counts failures in which trace evidence implicates memory rather than failures it causes alone, and the memory-free reruns of §5.1 provide the stronger test.

Designing against the memory tax. Our cases suggest four checks: verify the outcome signal before extracting a lesson (Finding 4), store each lesson with its scope and check it at use time (A6), budget stored procedures against the step limit (A5), and treat removal as a first-class update, using memory-free reruns to test whether stored guidance still earns its place (Table 2).

Limitations. Our shares describe one corpus and shift with its benchmark mix, and the method comparison above is observational. Memory updates can fail silently: with DeepSeek V4 Pro, 74% of AWM induction calls and 51% of Hermes curator updates failed to parse, and 8 of 21 AWM runs on this model never stored a workflow, so A8 understates how often the update pipeline breaks. Diagnoses come from a single analyst family, calibrated on a human subset rather than audited trace by trace, and Claude Sonnet 4.6 traces share that family. Interventions cover selected cases (14 memory-free comparisons and four warning experiments) and are sensitive to the model route, and our historical memory-free runs differ in tool budget and task subset, so we do not estimate the net effect of memory on task success at corpus scale.

AI USE STATEMENT

In this work, generative AI tools were used for text editing, figure refinement, implementing methods designed by the authors, and, as a core part of the methodology, analyzing experiment traces (§3). Claude Code agents running Claude Sonnet 5 diagnosed failed traces, proposed candidate failure classes from these diagnoses, and mapped each diagnosis to the final codebook; Claude Opus 5.5 re-coded all diagnoses as a robustness check (Appendix C). Human authors designed the study and analysis protocol, independently annotated a calibration subset of 677 traces and reconciled every disagreement with the automated diagnoses before large-scale analysis, and reviewed each candidate failure class against its supporting evidence. Diagnoses beyond the calibration subset were not individually audited; we test selected diagnoses through controlled reruns (§5) and discuss this limitation in §6. Generative AI was not used to produce original text from scratch or to design the methodology or experiments. Polished text was manually proofread for factual correctness, and generated code was independently verified and tested by several authors. The authors assume full responsibility for all content, claims, and artifacts presented in this paper.

ETHICS STATEMENT

This work does not raise specific ethical concerns. Our contributions focus on analyzing and mitigating failures in self-evolving language model agents. All experiments are conducted on publicly available benchmarks and datasets, without involving human subjects, sensitive data, or privacy-related information. No potential conflicts of interest are present.

REPRODUCIBILITY STATEMENT

We will release code, benchmark adapters, task traces, memory snapshots, and analysis outputs upon publication. Section 3 describes trace collection, annotation, taxonomy construction, and the intervention protocol; Appendix A gives the full codebook, Appendix B the coverage of methods, models, and benchmarks, Appendix G the run configurations, and Appendix H the instructions given to human annotators. Every intervention restarts from a saved memory snapshot, so each reported rerun can be reproduced from the released artifacts.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. *arXiv preprint arXiv:2506.07982*, 2025.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, et al. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025.

Xinle Deng, Ruobin Zhong, Hujin Peng, Xiaoben Lu, Yanzhe Wu, Guang Li, Buqiang Xu, Yunzhi Yao, Jizhan Fang, Haoliang Cao, et al. Memtrace: Tracing and attributing errors in large language model memory systems. *arXiv preprint arXiv:2605.28732*, 2026.

Ishir Garg, Neel Kolhe, Dawn Song, and Xuandong Zhao. Memfail: Stress-testing failure modes of llm memory systems. *arXiv preprint arXiv:2605.26667*, 2026.

Jiahao Huang, Fei Cheng, Junfeng Jiang, Zefan Yu, and Akiko Aizawa. Benchtrace: A benchmark for testing reflection ability and controlled evolution in llm agents. *arXiv preprint arXiv:2605.29225*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, volume 2024, pp. 54107–54157, 2024.

Seongyun Lee, Yongrae Jo, Minju Seo, Moontae Lee, and Minjoon Seo. Lost in the Noise: How Reasoning Models Fail with Contextual Distractors. *arXiv preprint arXiv:2601.07226*, 2026.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Nous Research. Hermes Agent. `https://github.com/NousResearch/hermes-agent`, 2026.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025.

Shuai Shao, Qihan Ren, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, Dongrui Liu, et al. Your agent may misevolve: Emergent risks in self-evolving llm agents. *International Conference on Learning Representations (ICLR)*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Xinyu Jessica Wang, Haoyue Bai, Yiyou Sun, Haorui Wang, Shuibai Zhang, Wenjie Hu, Mya Schroder, Bilge Mutlu, Dawn Song, and Robert D Nowak. The long-horizon task mirage? diagnosing where and why agentic systems break. *arXiv preprint arXiv:2604.11978*, 2026a.

Zihan Wang, Chi Gui, Xing Jin, Qineng Wang, Licheng Liu, Kangrui Wang, Shiqi Chen, Linjie Li, Zhengyuan Yang, Pingyue Zhang, et al. RAGEN-2: Reasoning Collapse in Agentic RL. *arXiv preprint arXiv:2604.06268*, 2026b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts llm agents: An empirical study of experience-following behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 623–645, 2026.

Ye Yu, Xiaopeng Yuan, Haibo Jin, Heming Liu, Yaoning Yu, and Haohan Wang. Do self-evolving agents forget? capability degradation and preservation in lifelong llm agent adaptation. *arXiv preprint arXiv:2605.09315*, 2026.

Dylan Zhang, Yanshan Lin, Zhengkun Wu, Yihang Sun, Bingxuan Li, Dianqi Li, and Hao Peng. Useful Memories Become Faulty When Continuously Updated by LLMs. *arXiv preprint arXiv:2605.12978*, 2026a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *International Conference on Learning Representations (ICLR)*, 2026b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Weixiang Zhao, Yingshuo Wang, Yichen Zhang, Yang Deng, Yanyan Zhao, Wanxiang Che, Bing Qin, and Ting Liu. Large Language Model Agents Are Not Always Faithful Self-Evolvers. *arXiv preprint arXiv:2601.22436*, 2026a.

Weixiang Zhao, Yichen Zhang, Yingshuo Wang, Yang Deng, Yanyan Zhao, Xuda Zhi, Yongbo Huang, Hao He, Wanxiang Che, Bing Qin, and Ting Liu. On Safety Risks in Experience-Driven Self-Evolving Agents. *arXiv preprint arXiv:2604.16968*, 2026b.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

A FINE-GRAINED FAILURE CATEGORIES

Our taxonomy contains 25 failure classes organized into three families and ten subcategories. Each coded failure receives one class label describing its mechanism. Causal attribution is recorded separately to distinguish contributions from the base model, accumulated state, and environment, including mixed or uncertain cases. The family label alone therefore does not establish the cause of an individual failure. Below, we define each class and clarify its boundaries with related classes. We use an additional code, X1, when the available evidence is insufficient for classification.

A.1 MEMORY/STATE-INDUCED FAILURES (FAMILY A)

Family A covers failures involving the content, availability, size, or update mechanisms of persistent state, such as playbooks, workflows, and skills. Assigning an A code requires evidence linking the failure to the agent's use of memory or to a constraint imposed by the state-management pipeline. The presence of memory in the context is not sufficient by itself.

A.1.1 MEMORY CONTENT DEFECTS (A.I)

The agent receives and follows memory whose guidance is incorrect or does not transfer to the current task.

- **A1: Misleading memory.** Retrieved memory is relevant to the task but contains incorrect, overgeneralized, or inappropriate cross-domain guidance, which the agent follows into failure. Examples include unsupported "always" or "never" rules, invented restrictions, and escalation advice transferred between domains with different policies. This code requires evidence that the agent acted on the guidance, such as an explicit reference to the memory or an action sequence that follows it. A2 identifies the specific case of memory-reinforced abstention, while A6 identifies recipes that were correct for their source tasks but fail to transfer. A policy error without a memory source belongs to B6.
- **A2: Over-abstention loop.** Cautionary lessons are stored and later retrieved, repeatedly reinforcing the decision to withhold an answer. The resulting non-answers fail the benchmark's answer criterion, including exact-match evaluation. Evidence includes abstention rationales that echo retrieved lessons or increased abstention after these lessons enter memory. Unlike B8, this class requires evidence of reinforcement through accumulated state.
- **A6: Overfit recipes.** A workflow or skill learned from one or a few tasks preserves assumptions that are valid for those tasks but inappropriate for a similar-looking new task. The retained details may include specific entities, data sources, action orderings, or task templates. The defining feature is failed transfer of a previously valid procedure. This distinguishes A6 from A1, where the stored guidance is itself incorrect or overgeneralized, and from A3, where retrieval supplies memory for an unrelated task or topic.

A.1.2 MEMORY DELIVERY FAILURES (A.II)

The agent does not receive the lesson needed for the current task because it is absent from state or the retrieval process selects an unsuitable entry.

- **A3: Retrieval mismatch.** Retrieval supplies memory for an unrelated task or topic, and the agent spends effort applying it or is otherwise diverted by it. Possible contributing conditions include a sparse memory collection, a small retrieval budget, and a similarity measure that selects the wrong entry. A mismatch is not coded as a failure when the agent explicitly dismisses the irrelevant memory and it does not explain the unsuccessful outcome. A3 concerns the selection of memory, whereas A4 concerns the absence of the needed lesson from state.
- **A4: Memory coverage gap.** A useful lesson is systematically absent because of how the method produces or retains knowledge. Examples include learning only from successful tasks, recording a corrective rule only after a repeated failure, and evicting useful lessons from a capped memory collection. This code requires evidence that the method had an opportunity to retain the lesson, such as an earlier occurrence of the same failure. Ordinary lack of task knowledge is insufficient. If a lesson was generated but lost through a mechanical update defect, the appropriate code is A8.

A.1.3 STATE MECHANICS FAILURES (A.III)

Failures arise from the volume or execution cost of accumulated state, or from defects in the machinery that updates it.

- **A5: Procedural bloat.** Accumulated checklists and verification rules require so many actions that the agent exhausts its turn or token budget before completing the task or reporting the result. Individual instructions may be reasonable; the problem is their combined execution cost. The trace must show the agent following an extended sequence of memory-driven checks. Unlike B12, the agent progresses through distinct checks rather than repeatedly attempting the same failing action. Unlike A7, the episode runs but exhausts its budget rather than failing through a hard technical error.
- **A7: State-size crash.** Accumulated state exceeds a hard technical limit and causes a model call or the harness to fail. Examples include exceeding the model's context limit or an operating-system argument-size limit when injecting a large playbook. The error must be traceable to state size. Such failures belong to A7 rather than C2, even when their immediate manifestation is an infrastructure error.
- **A8: Update pipeline defect.** A mechanical fault in the memory update process drops, corrupts, or incorrectly modifies useful state. Examples include failures to parse reflection or curation outputs, updates applied to the wrong entry, and unintentionally skipped writes. The defect must demonstrably affect a task outcome or prevent the method from operating for the failed task. We assign the code to the affected task. This differs from A4, where the pipeline operates as designed but fails to produce or retain the needed lesson.

A.2 BASE MODEL FAILURES (FAMILY B)

Family B describes failures in task understanding, information gathering, action execution, answer commitment, and control flow. These mechanisms can occur when memory is absent, benign, or ignored. Here, the agent includes the base model and its task-execution scaffold. Within this family, we identify the earliest stage at which the failure mechanism begins, rather than coding only its downstream symptom.

A.2.1 TASK GROUNDING ERRORS (B.I)

The agent adopts an incorrect interpretation of the task, the relevant data, or the temporal reference before acting.

- **B1: Task misinterpretation.** The agent misreads the task's target, constraints, or selection rule. For example, it may interpret a request about the user's preferences as a question about global popularity. Unlike B2, the error lies in understanding the request itself rather than choosing a data source for a correctly understood goal. Errors limited to the temporal reference belong to B4.
- **B2: Wrong source of truth.** The agent understands the goal but selects an endpoint, field, or record type whose meaning does not match the required information. Examples include using global popularity counts to infer personal likes or treating a friends-only search as evidence of whether an account exists. The distinction from B3 is that B2 queries the wrong source, whereas B3 queries an appropriate source incompletely.
- **B4: Temporal grounding error.** The agent uses an incorrect reference for the current date or time, leading it to apply the wrong time window, expiry condition, or threshold. The reference may be fabricated or inferred from unrelated metadata. The task interpretation and data source can otherwise be correct; the decisive error is the temporal anchor.

A.2.2 INFORMATION GATHERING ERRORS (B.II)

The agent understands what information is needed but does not obtain enough of it to complete the task correctly.

- **B3: Incomplete enumeration.** The agent acts on an incomplete set of relevant entities or records. Examples include reading only the first API page, omitting one source from an aggregation, and stopping candidate enumeration too early. Unlike B11, the agent has reached an appropriate

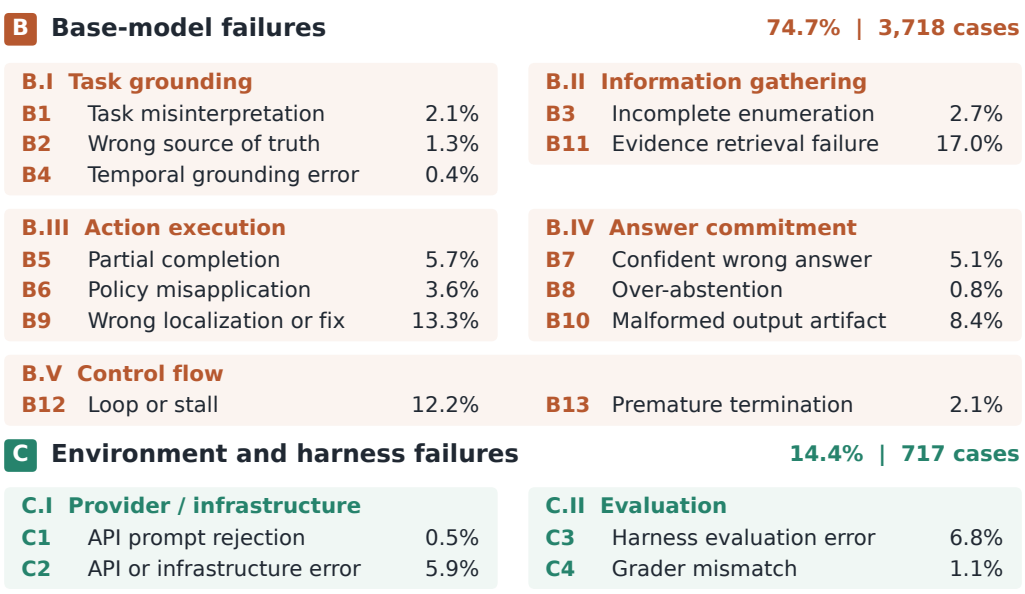

Figure 3: **Base-model, environment, and harness failures.** Classes are grouped by the affected component or behavior. Percentages are shares of all 4,979 coded failures.

source but reads too little of it. Unlike B5, the missing work concerns gathering information rather than executing required actions.

- **B11: Evidence retrieval failure.** The search or browsing process does not surface the evidence needed to answer the task, leaving the agent to guess or choose an unsupported candidate. When the retrieval tools function correctly, the failure may arise from the query strategy or search process. If the required document is demonstrably absent from the searchable collection, the instance receives an environment attribution. This class concerns task evidence, rather than retrieval of stored experience as in A3.

A.2.3 Action Execution Errors (B.III)

The agent has sufficient understanding and information, but its actions are incorrect, policy-violating, or incomplete.

- **B5: Partial completion.** The agent performs only part of the required work and treats the task as complete. Examples include updating some but not all relevant records, completing only one layer of a multi-layer repair, or omitting a required verification step. The actions it performs may be correct, distinguishing B5 from B9. Unlike B13, the agent does not explicitly recognize that it is abandoning an unfinished task.
- **B6: Policy misapplication.** The agent violates or invents a domain policy without being directed to do so by retrieved memory. Examples include imposing unsupported restrictions, escalating an action it could perform, incorrectly inferring eligibility, and violating an action's state preconditions. If the incorrect rule comes from memory and the agent follows it, the failure belongs to A1. Unlike B13, the agent's refusal or escalation follows an incorrect policy judgment rather than simple abandonment.
- **B9: Wrong localization or fix.** In a code-repair task, the agent modifies the wrong file, location, or abstraction layer, or implements a semantically incorrect repair. A patch may be syntactically valid and apply successfully while failing to address the underlying problem. An incomplete patch whose implemented changes are correct belongs to B5. A patch that cannot be applied because its format is invalid belongs to B10.

A.2.4 ANSWER COMMITMENT ERRORS (B.IV)

The failure occurs when the agent commits to an incorrect answer, withholds an answer, or produces an invalid final artifact.

- **B7: Confident wrong answer.** The agent has the relevant evidence or computation inputs but produces an incorrect final answer without recognizing the error. Examples include arithmetic or aggregation mistakes, selecting the wrong candidate despite available evidence, and extracting the wrong value from a document. Unlike B11, the necessary evidence has been obtained. Unlike B10, the answer's content is incorrect rather than merely its format.
- **B8: Over-abstention.** The agent returns a refusal or an inability-to-determine response that fails the benchmark's answer criterion, without evidence that accumulated memory reinforced the decision. Memory-reinforced abstention belongs to A2. This class concerns withholding an answer to a question; abandoning an action task with viable options remaining belongs to B13.
- **B10: Malformed output artifact.** The final deliverable violates an explicit output or interface requirement. Examples include malformed unified diffs, prose in place of a required patch, missing final-call arguments, and incorrect formatting of an otherwise correct answer. The defining problem is the deliverable's form. If substantively correct work is rejected because of an unstated or overly strict grading requirement, the failure belongs to C4.

A.2.5 CONTROL FLOW ERRORS (B.V)

The episode fails because the agent becomes stuck or stops while useful actions remain available.

- **B12: Loop or stall.** The agent repeatedly attempts the same failing action, oscillates between unproductive actions, or restates intentions without executing them until its budget is exhausted. Unlike A5, this is not the execution of an excessively long sequence of distinct memory-driven checks. Unlike B13, the agent continues without making progress rather than choosing to terminate.
- **B13: Premature termination.** The agent explicitly gives up while viable, untried options remain. For example, it may stop after one funding source is rejected even though other valid sources are available. Unlike B5, the agent recognizes that the task is unfinished. Unlike B8, the failure concerns abandoning an action task rather than withholding a factual answer.

A.3 ENVIRONMENT AND HARNESS FAILURES (FAMILY C)

Family C covers failures in external services, execution infrastructure, and evaluation. We distinguish errors that prevent or interrupt execution from errors in assessing the work produced.

A.3.1 PROVIDER AND INFRASTRUCTURE FAILURES (C.I)

The model service or execution environment prevents meaningful action or interrupts an otherwise ongoing episode.

- **C1: API prompt rejection.** The model provider rejects a prompt through a content or safety filter before meaningful agent action occurs. The resulting trace is empty or nearly empty. Other provider and infrastructure errors belong to C2.
- **C2: API or infrastructure error.** Execution is interrupted by a provider or infrastructure fault, such as a server error, exhausted rate-limit retries, a sandbox failure, or a timeout without an identified agent-side cause. Timeouts caused by repeated unproductive agent actions belong to B12. Hard errors caused by accumulated state exceeding a technical limit belong to A7.

A.3.2 EVALUATION FAILURES (C.II)

The agent produces work, but the evaluation process fails to assess it correctly.

- **C3: Harness evaluation error.** The harness does not produce a valid assessment because of an evaluation exception, a logging or parsing fault, or a service-lifecycle mismatch. The resulting outcome is missing, unknown, or otherwise invalid. Unlike C4, the issue is the absence of a valid verdict rather than an incorrect failure verdict on substantively correct work.

- **C4: Grader mismatch.** The agent's work is substantively correct but is rejected because the verifier enforces an unstated or overly strict expectation. Examples include rejecting equivalent answer forms or failing a natural-language assertion despite correctly executed actions. If the agent instead violates an explicitly stated output contract, the failure belongs to B10.

A.4 Uncodable Cases

- **X1: Uncodable.** The available record does not provide enough usable evidence to assign a failure to any of the classes above. Each X1 assignment includes a brief explanation of the missing or insufficient evidence. X1 is not used merely to avoid choosing between two plausible classes; in such cases, the better-supported class is selected with lower confidence. The 101 X1 cases are excluded from the 4,979 coded failures used to compute the reported taxonomy percentages.

B Coverage Matrix

Table 4: **Coverage by method, base model, and benchmark.** Each cell reports runs / traces. The final row per method reports runs / traces / failed traces. GPT refers to GPT-5.4; Son refers to Claude Sonnet 4.6; DS refers to DeepSeek V4 Pro.

Method	Model	AppWorld	τ^2-bench	SWE-bench Lite	Terminal-Bench	BrowseComp+
ACE	GPT	5 / 177	7 / 280	5 / 206	3 / 120	6 / 210
	Son	5 / 177	1 / 100	1 / 100	1 / 60	1 / 60
	DS	4 / 120	6 / 180	4 / 106	2 / 60	5 / 150
	All	14 / 474 / 77	14 / 560 / 286	10 / 412 / 302	6 / 240 / 128	12 / 420 / 314
AWM	GPT	6 / 233	8 / 380	6 / 306	4 / 180	7 / 270
	Son	5 / 203	2 / 200	2 / 200	2 / 120	2 / 120
	DS	4 / 120	6 / 180	4 / 106	2 / 60	5 / 150
	All	15 / 556 / 106	16 / 760 / 396	12 / 612 / 449	8 / 360 / 155	14 / 540 / 399
ReasoningBank	GPT	6 / 233	8 / 380	6 / 306	4 / 180	7 / 270
	Son	2 / 113	2 / 200	2 / 200	2 / 120	2 / 120
	DS	4 / 120	6 / 180	4 / 106	2 / 60	5 / 150
	All	12 / 466 / 67	16 / 760 / 381	12 / 612 / 446	8 / 360 / 169	14 / 540 / 402
Hermes	GPT	5 / 180	7 / 280	5 / 206	1 / 60	6 / 210
	Son	1 / 60	1 / 100	1 / 100	1 / 60	1 / 60
	DS	4 / 120	6 / 180	4 / 106	–	5 / 150
	All	10 / 360 / 55	14 / 560 / 280	10 / 412 / 291	2 / 120 / 40	12 / 420 / 337

Failed traces range from a single patch-generation call over retrieved source files on SWE-bench Lite to a median of 55 agent steps on AppWorld. By the end of a run, the median memory state holds roughly 8K tokens and contains 87 playbook bullets for ACE, 90 memory items for ReasoningBank, and 15 workflows for AWM, while Hermes typically concentrates its experience in a single, repeatedly patched skill; the largest ACE and Hermes states approach the 50K-token cap (Appendix G).

C Robustness of Class Assignment

To test whether class assignment depends on the coding model, we re-code all 5,080 diagnoses against the fixed codebook with Claude Opus 5.5, keeping the diagnoses themselves unchanged. Table 5 compares the resulting distribution with the primary Claude Sonnet 5 coding used throughout the paper.

Family shares differ by at most 0.2 percentage points and class shares by at most 1.4 points; class shares are rank-correlated at Spearman $\rho = 0.99$, and the total variation distance between the two class distributions is 5.9 points. Because the comparison is over aggregate shares, it tests whether the coding model changes the distribution, not how often the two codings agree on individual diagnoses. The differences concentrate in three places. First, the boundary between misleading memory and overfit recipes shifts: Opus 5.5 assigns fewer failures to A1 (5.2% $\rightarrow$ 4.5%) and more to A6 (1.0% $\rightarrow$ 1.9%), reading some stored guidance as valid for its source task but misapplied rather than

Table 5: **Class shares under two coding models.** Shares of coded failures (%). Both codings cover the same 5,080 diagnoses; X1 counts differ, so the coded totals differ.

Class	Sonnet 5	Opus 5.5	Class	Sonnet 5	Opus 5.5	Class	Sonnet 5	Opus 5.5
A1	5.2	4.5	B1	2.1	2.6	C1	0.5	0.7
A2	0.6	0.7	B2	1.3	1.0	C2	5.9	4.7
A3	0.1	0.1	B3	2.7	2.6	C3	6.8	7.1
A4	0.2	0.0	B4	0.4	0.4	C4	1.1	2.0
A5	3.5	3.4	B5	5.7	5.8			
A6	1.0	1.9	B6	3.6	3.5			
A7	0.2	0.0	B7	5.1	3.7			
A8	0.1	0.0	B8	0.8	1.3			
			B9	13.3	14.5			
			B10	8.4	8.9			
			B11	17.0	17.1			
			B12	12.2	10.8			
			B13	2.1	2.7			
A	**10.9**	**10.7**	**B**	**74.7**	**74.8**	**C**	**14.4**	**14.4**
Coded failures: 4,979 (Sonnet 5) vs. 5,047 (Opus 5.5); uncodable (X1): 101 vs. 33.								

incorrect. The Family A total changes by only 0.2 points, so this ambiguity does not move failures across families. Second, within Family B, shares move between adjacent classes whose boundaries Appendix A defines explicitly: B7 to B9, and B12 to B13 and B8. Third, three rare memory/state classes (A4, A7, and A8; 24 failures combined under Sonnet 5) nearly vanish under Opus 5.5 (0.0% each at one decimal). Their existence rests on few cases and one coding model, so we report them for completeness but do not interpret them individually. A3 is equally rare (six cases) but keeps its 0.1% share under both coders; we discuss it through individual traces and controlled reruns (§5.1) rather than through its share.

D Memory-Free Reference Comparisons

Table 6 reports every memory/state intervention case with a memory-free condition, including the five cases in Table 2. Entries report full task success (benchmark score 1.0). Each memory-free condition uses the same model route, prompt template, harness, and number of reruns as its control; the memory section is replaced by the empty playbook template (section headings only). Cases whose control reruns never reproduced the original failure and that were not given a memory-free condition are omitted. The pooled one-sided exact test stratified by task gives $p \approx 2 \times 10^{-6}$ across all 14 comparisons (control $15/46$, memory-free $35/46$).

In 13 of 14 comparisons the memory-free agent succeeds at least as often as the control. The exception is an ACE case of A2, where the playbook improves success over the memory-free agent. In the remaining A2 cases, the harmful effect is concentrated in Hermes skills, while the ReasoningBank memory has at most a small effect. In the second A6 case, a narrow edit to the payment-derived playbook bullets leaves success at $1/3$, whereas removing all three bullets and removing memory entirely both reach $3/3$.

Checking attribution with memory-free reruns. The memory-free condition also tests whether a Family A label is warranted: if the same model, prompt, and harness succeed without memory, the failure cannot be attributed to task difficulty alone. This check resolves an ambiguous case in the ACE messaging task, where the agent's friends-only account check could be read as a base-model source error (B2) or as an overfit recipe (A6). The memory-free agent uses a global user search and succeeds in $3/3$ reruns, matching historical memory-free baselines with GPT-5.4 and Claude Sonnet 4.6, which supports the A6 label. Conversely, the one memory-free failure in the A1 album-genre case, summing public like counts rather than the user's liked songs, occurs without memory and is a B2 error. These checks apply only to the intervention subset; they do not estimate label accuracy across the corpus.

Table 6: **All memory-free comparisons.** Full task success as control / memory-free. *Case in Table 2.

Class	Method	Model	Task	Control	No mem.
A1	ACE	GPT-5.4	Venmo payment, card declines*	$0/3$	$3/3$
A1	ReasoningBank	GPT-5.4	Least-liked genre in album library	$0/3$	$2/3$
A1	ACE	GPT-5.4	Accept Venmo request	$3/5$	$5/5$
A2	Hermes	GPT-5.4	BrowseComp+ 844*	$1/3$	$3/3$
A2	Hermes	GPT-5.4	BrowseComp+ 822	$0/3$	$2/3$
A2	ReasoningBank	GPT-5.4	BrowseComp+ 537	$0/3$	$1/3$
A2	ACE	GPT-5.4	BrowseComp+ 822	$3/3$	$1/3$
A3	ReasoningBank	Sonnet 4.6	AppWorld purchase, wish list*	$0/3$	$3/3$
A3	ReasoningBank	Sonnet 4.6	AppWorld purchase, same stream	$3/3$	$3/3$
A3	AWM	GPT-5.4	τ^2 telecom, task 1*	$0/3$	$1/3$
A3	AWM	GPT-5.4	τ^2 telecom, task 2*	$0/3$	$1/3$
A6	ReasoningBank	DeepSeek V4 Pro	Gift purchase, RSVP recipe*	$0/3$	$2/3$
A6	ACE	GPT-5.4	Message relatives without Venmo	$1/3$	$3/3$
A6	Hermes	DeepSeek V4 Pro	Update standup alarm	$4/5$	$5/5$

Telecom case details. In the two AWM telecom tasks of Table 2, transfers to a human agent occur in $4/6$ control and $3/6$ intervention reruns but in no memory-free rerun. The memory-free agent is the only condition to achieve any full success ($2/6$); its remaining four reruns reach the 30-step limit, and a historical memory-free run on a different model route fails both tasks. One intervention rerun fixes both the Data Saver and roaming faults and reaches "Excellent" connection speed, but hits the step limit before finishing the interaction.

E Case Studies

E.1 Repairing Evaluation Feedback

Some recorded failures require changes outside the agent's context. In BrowseComp+, the judge marks an answer correct, but the parser records failure when the verdict is formatted as JSON. Fixing the parser corrects both task scores selected for intervention by rescoring stored judge responses, without rerunning the agent. Across 1,213 stored judge replies, the correction changes 14 task scores from failure to success, with no changes in the reverse direction. This repairs the recorded feedback without changing the agent's answer; its effect on subsequent memory updates is not evaluated.

E.2 Misleading memory from a successful episode (A1)

ACE, AppWorld, GPT-5.4, task `37a8675_2`, step 22. At step 21, the agent had to send money on Venmo, found the balance short, and tried saved cards one at a time until the third succeeded, scoring 1.0. Reflection objected to the method rather than the result, arguing that the agent should have selected the best card deterministically, and the curator recorded bullet `psw-00068`: exclude expired cards, then choose a single best candidate rather than retrying cards one by one. One step later, the agent filtered four cards to the non-expired ones, picked the latest expiry, received a decline for insufficient funds, and ended the task with two cards untried. The memory-free agent tried four cards and the fourth paid. Because card balances are invisible until a charge is attempted, trial and error was the only available strategy; reflection removed the one behavior that worked, and the rule stayed in the playbook for the rest of the run.

E.3 A cautionary footer that suppresses answers (A2)

Hermes, BrowseComp+, GPT-5.4, task 772, step 3. After the run's first failure, Hermes wrote a skill whose closing checklist told the agent to continue searching or give a bounded uncertainty response without speculation; the footer then appeared, unchanged, in 59 of the run's 60 contexts. On task 772 the agent retrieved the right documents (retrieval recall and citation precision both 1.0) and still

declined to answer, stating that it could not verify the role; the gold answer was *Secretary*. The memory-free agent, on the same task and corpus, committed to "secretary/administrator," which was marked wrong for hedging but is a near miss rather than a refusal. Memory-free agents on BrowseComp+ already abstain on roughly 62% of their substantive failures, so an A2 label is a claim about what memory adds, and a baseline comparison is part of the evidence.

E.4 A CORRECT LESSON OUTRANKED AT RETRIEVAL (A3)

ReasoningBank, AppWorld, DeepSeek V4 Pro, task `9871968_2`, step 4. Three tasks earlier, the bank had stored the lesson this task needed: purge unwanted items from the cart before ordering, or the entire cart will be ordered. ReasoningBank retrieves one entry ranked by similarity to the task text, and a previous entry whose source task differed by one word ("work" instead of "home" address) won; its single item concerned payment-card declines. The agent never inspected the cart, added seven wish-list items to a cart that already held other shopping, and applied the card-decline advice correctly at checkout, paying $2,900.09 for a $425.90 wish list. The same method on GPT-5.4, at the same task and step, retrieved the cart lesson and scored 1.0. An A3 label requires showing that the better lesson existed in the logged state at that step; otherwise the failure is indistinguishable from never having learned it.

E.5 PROCEDURAL BLOAT UNDER A TURN LIMIT (A5)

Hermes, τ^2-bench telecom, Claude Sonnet 4.6, step 80. The agent's context held 72,587 characters across three skills, including the airline and retail skills on a telecom task; the word "mandatory" appears eleven times, and the telecom skill requires running both diagnostic tools in order even if the first already reveals the cause. The user's status bar already showed `Poor | 2G`, which is the diagnosis, yet the agent requested the second check because the rule forbade skipping it. Its next message requests the correct fix, `set_network_mode_preference(mode='4g_5g_preferred')`, at the 30-turn limit, and the user never replies. Reaching the limit is not evidence by itself, since memory-free agents on τ^2-bench reach it about as often; the A5 label rests on a specific turn spent because memory required it.

E.6 A CORRECT RECIPE APPLIED BEYOND ITS SCOPE (A6)

ReasoningBank, AppWorld, GPT-5.4, task `ec655df_1`, step 6. Four steps earlier, on a task that asked for whatever was cheaper, the agent found that buying wrapping paper beat the platform's gift-wrap fee, and the finding became item `rb-00007`, scoped in its own description to tasks that ask for the cheaper option. The current task asked for a gift-wrapped tracker for each coworker who had RSVPed. The agent added the five trackers and applied native gift wrapping, at which point the cart was exactly correct, then followed the recipe: it removed the wrapping, bought a roll of paper, compared totals, and ordered. The grader checks gift-wrap quantity, which was now zero. Eight of the ten runs of this task never buy paper, and all eight score 1.0.

E.7 MISATTRIBUTION WITHOUT UPDATE TIMING (B9)

ACE, SWE-bench Lite, GPT-5.4, `django__django-12184`, step 25. A `re_path` with an optional named group raises a positional-argument error, and the fix belongs in `django/urls/resolvers.py`. The retriever supplied four files, none of them `resolvers.py`, and in this single-shot setting the model patched the deepest layer it was shown: a well-formed change to the request handler that repairs the crash where the callback is invoked but leaves the faulty argument construction untouched. An initial annotation recorded the failure as the agent ignoring a playbook rule naming `resolvers.py`; that rule was written from this very failure and was absent from the step-25 context. Checking which files were retrieved, and when each memory item was created, is therefore part of coding B9 instances.

E.8 SEARCHING THE WRONG CLUES (B11)

Hermes, BrowseComp+, DeepSeek V4 Pro, task 528, step 0. The question gives an October debut album with 11 tracks and two musicians with birth-year windows of 1967–1970 and 1961–1964; the

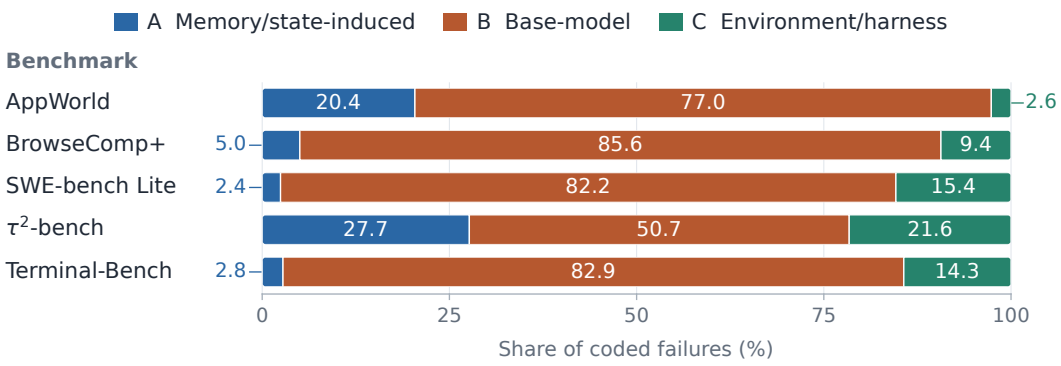

Figure 4: **Failure distribution across benchmarks.** Bars show each family's share of coded failures within a benchmark, excluding uncodable cases. Failure rate / share of executions lost to memory/state-induced failures: AppWorld 16% / 3.3%, BrowseComp+ 76% / 3.8%, SWE-bench Lite 73% / 1.8%, τ^2-bench 51% / 13.7%, Terminal-Bench 46% / 1.2%.

birth years are the only clues that narrow the field. The agent issued 83 searches, 82 of them distinct, so the failure is neither a loop nor a lack of effort: "October" appears in 76 queries, "1967" in none, and "1961" in one, and the first query stacks five clues into a single string. The episode ends without a final answer. The search tool returned documents every time; what failed was the choice of what to search for, before any memory existed.

E.9 A CORRECT ARTIFACT RECORDED AS A FAILURE (C3)

AWM, Terminal-Bench, Claude Sonnet 4.6, `password-recovery`*, step 56.* The logged result reads `reward: 1.0` and `outcome: failure`: the verifier ran and the tests passed. The Terminal-Bench adapter records success only when the reward meets the threshold *and* no exception occurred, and the exception flag is set by any trial exception, including the agent process exceeding its wall-clock limit. An agent that produced a correct artifact but did not exit cleanly is therefore recorded as having failed. Two other tasks in the same run carry the same signature, and the pattern appears in 13 of the 1,080 Terminal-Bench traces in our corpus. For a method that learns from failure feedback, each such record risks teaching the agent to avoid a procedure that worked.

F FAILURE DISTRIBUTION

Table 7 breaks down coded failures by method and base model, and Table 8 breaks down memory/state-induced failures by class. Failure rates vary little across methods (52.6–53.6%) and models (49.8–54.7%), but the memory/state-induced share varies threefold across methods, from 5.0% for AWM to 17.5% for Hermes (§6). Hermes's memory/state-induced failures are dominated by procedural bloat (94 of 171), ACE's by misleading memory (118 of 189), and state-size crashes occur only under ACE, whose playbook is injected whole. All 29 over-abstention loops (A2) occur with GPT-5.4, 28 of them on BrowseComp+, and misleading memory and procedural bloat account for 335 of the 361 memory/state-induced failures on τ^2-bench. DeepSeek V4 Pro shows the largest environment and harness share (19.6%). Within base-model failures, the dominant classes follow the benchmark: evidence retrieval failure and loops on BrowseComp+ (625 and 349), wrong localization and malformed patches on SWE-bench Lite (588 and 360), and loops, partial completion, and wrong fixes on Terminal-Bench (91, 78, and 71). Attribution and class remain distinct: 321 of the 846 evidence-retrieval failures (B11) carry an environment attribution, which the codebook reserves for cases where the required document is absent from the searchable collection.

F.1 MEMORY/STATE-INDUCED FAILURES OVER THE TASK STREAM

Table 9 splits each run into thirds by task position and reports memory/state-induced failures as a share of executions. We use the larger collection, where each run covers a single domain, benchmark subset, or repository; in the smaller collection, τ^2-bench streams run through the airline, retail, and

Table 7: **Failures by method and base model.** Family columns give shares of coded failures (%); *A per exec.* is memory/state-induced failures as a share of all executions.

	Coded	Fail rate	A	B	C	A per exec.
ACE	1,093	52.6	17.3	72.0	10.7	9.0
AWM	1,486	53.2	5.0	82.9	12.1	2.6
ReasoningBank	1,425	53.5	7.7	73.9	18.4	4.0
Hermes Agent	975	53.6	17.5	66.3	16.2	9.1
GPT-5.4	2,520	54.7	12.4	74.4	13.2	6.7
Claude Sonnet 4.6	1,206	49.8	10.8	77.8	11.4	5.3
DeepSeek V4 Pro	1,253	53.9	8.1	72.2	19.6	4.2

Table 8: **Memory/state-induced failures by class** (counts), by benchmark and by method.

	A1	A2	A3	A4	A5	A6	A7	A8	All
AppWorld	32	0	2	1	8	16	0	3	62
BrowseComp+	30	28	1	0	4	1	8	0	72
SWE-bench Lite	19	0	1	0	0	16	0	0	36
τ^2-bench	174	1	1	7	161	14	0	3	361
Terminal-Bench	6	0	1	0	0	4	2	0	13
ACE	118	7	1	1	41	9	10	2	189
AWM	30	2	2	7	10	22	0	1	74
ReasoningBank	56	8	2	0	28	13	0	3	110
Hermes Agent	57	12	1	0	94	7	0	0	171
Total	261	29	6	8	173	51	10	6	544

telecom domains in sequence, which confounds position with domain. Task order is fixed within each subset (by task index or name) and identical across methods in 45 of 46 subset-model groups, so position is confounded with task identity but not with method. We therefore compare trends across methods: from the first to the last third, the rate for ACE and Hermes grows by 4.8 percentage points more than for ReasoningBank and AWM (95% run-level bootstrap interval: 1.4–8.6). The median pre-task state grows from about 0.8K to 4.6K tokens over the same positions. Methods that inject their whole store thus accumulate a memory tax as the store grows, whereas methods that retrieve a few entries hold it roughly constant.

Table 9: **Memory/state-induced failures by stream position** (larger collection). Entries give memory/state-induced failures as a share of executions (%), with counts in parentheses.

	First third	Middle third	Last third
ACE	4.4 (20)	8.5 (38)	11.1 (50)
Hermes Agent	5.4 (20)	8.4 (31)	9.4 (35)
ReasoningBank	3.2 (13)	3.9 (16)	3.2 (13)
AWM	1.4 (6)	3.9 (17)	2.7 (12)

G Extended Experiment Setup

Shared settings. All runs decode at temperature 0.0, and memory updates follow each method's official implementation after every task. Persistent state is capped at 50K tokens, except in five Terminal-Bench runs (one AWM, four ReasoningBank) capped at 20K. Runs in the smaller collection mostly call provider APIs directly, whereas all runs in the larger collection route through OpenRouter; eight AppWorld runs (five ACE, three AWM) use low reasoning effort and low verbosity.

Memory delivery. ACE and Hermes Agent insert their full playbook or skill store into the context of every task, except in four ACE runs (two each on SWE-bench Lite and Terminal-Bench), which

retrieve the top 100 playbook bullets once the playbook exceeds 12K tokens (capped at 20K and 10K retrieved tokens, respectively). ACE merges playbook updates with an LLM curator. ReasoningBank retrieves the single most similar memory entry ($k = 1$), except in twelve τ^2-bench runs that use $k = 5$, and AWM retrieves up to five workflows.

Benchmarks. τ^2-bench episodes are limited to 30 steps with a simulated user. SWE-bench Lite runs are single-shot: a retriever supplies source files, and the model emits one patch, which the harness applies and tests. Terminal-Bench tasks run inside a command-line coding agent (OpenCode for all ACE runs and for the larger collection; Codex CLI for GPT-5.4 and Claude Code for Claude Sonnet 4.6 in the smaller collection) under per-task wall-clock limits ranging from 6 to 60 minutes. BrowseComp+ answers are graded by an LLM judge. Task order is fixed within each subset and shared across methods (Appendix F.1).

H Instructions for Human Annotators

Annotation Instructions: Why Do Self-Evolving LLM Agents Fail?

What this is. You're reading agent traces and writing short failure analyses. Your annotations serve as **calibration data**—we'll compare them against an independent LLM analysis of the same traces to find where humans and LLMs agree or diverge. You're not building a codebook; you're providing expert judgment that grounds an automated pipeline.

The setup. Each trace comes from an **inter-task self-evolving agent**—an agent that accumulates experience across separate tasks and reuses it on future ones (e.g., ACE, AWM, ReasoningBank). A trace consists of the agent's actions on one task plus the experience memory it carried in. Each trace is labeled with its method, benchmark, and outcome.

The core question for every trace: **did the self-evolution mechanism help, hurt, or do nothing?**

What to write. For each trace, write a short analysis (roughly half a page to a page) covering the following. You don't need to use headers or follow a rigid template—just make sure your write-up addresses these points:

- **What happened.** What did the agent try, and how did it end? Where did things go wrong (or right)?
- **Why it failed.** What's the root cause? Be mechanistic—point to the specific component (retrieval, experience selection, experience application, reflection) and reference concrete moments in the trace (iteration numbers, action steps, experience IDs).
- **Baseline vs. method-induced.** For each failure mode you identify, judge whether it's a baseline LM error (would happen without the self-evolution loop), a method-induced error (caused or amplified by the loop), or a mix of both. This is the most important analytical distinction in the annotation.
- **Dynamics.** Is this a one-time error, or does it accumulate / get amplified across the self-evolution loop?
- **Counterfactual.** Would a vanilla single-shot agent (no experience memory) have done better, worse, or about the same on this task? Why?
- **Evidence pointers.** Cite specific locations in the trace (action steps, experience entries, code diffs) so another reader can verify your claims. A few key references are enough.

Feel free to note uncertainty, flag ambiguities, or add observations that don't fit the categories above. Surprises and edge cases are especially valuable.

Procedure.

1. Read the full trace before writing. Skim for the overall arc, then revisit key moments.
2. Form your own diagnosis before consulting anything external.
3. Write your analysis. **Target: 20–30 minutes per trace.** If a trace takes over 45 minutes, write what you have and flag it.

LLM usage policy. You may use LLMs to understand unfamiliar APIs, check whether a code snippet has a bug, or look up background on the benchmark. You may **not** use LLMs to generate or draft your failure analysis—don't paste the trace and ask "what went wrong." The whole point is independent human judgment for calibration.

What makes a good annotation.

- **Specificity:** "The agent retrieved `exp_042` (a Django aggregation fix) for a lazy evaluation bug because both mention 'queryset' and 'filter"' beats "the agent retrieved a bad experience."

- **Mechanism:** Don't just describe what happened—explain *why* the system produced that behavior. Which component is responsible?
- **Honest uncertainty:** If you can't determine the root cause, say so. A flagged ambiguity is more useful than a forced answer.