COUNTERMEM: WORLD-MODEL-VERIFIED COUNTERFACTUAL MEMORY FOR LANGUAGE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing agent memory frameworks mainly create memory through an agent's interaction with the factual world, e.g., remembering feedback from actions taken to improve performance on future tasks. However, these frameworks seldom ask the "what if" question during memory construction: what if a different action had been taken, would the feedback have changed, and how could this feedback become useful memory? Obtaining such feedback directly in an active environment can be expensive and can alter the state needed for comparison. In this work, we introduce COUNTERMEM, a reinforcement-learning framework for constructing and using verified counterfactual memory across tasks. After a failed action, COUNTERMEM evaluates local alternatives from a copy or reset of the original state using executable world models, such as tests, proof checkers, and solvers. It stores improvements with the original and corrected actions, checked outcomes, and conditions for reuse. A learned memory-use policy selects a retrieved record or skips memory to balance task success and interaction cost, while the base LLM remains fixed. Both memory and policy are frozen during held-out evaluation. We evaluate COUNTERMEM on 12 benchmark settings across six domains. With gpt-oss-120b, COUNTERMEM improves both ReAct and Reflexion on all 12 benchmarks across six domains, averaging a gain of 12.6 percentage points over their unaugmented versions. In the four-domain comparison across two backbones, task-run tokens decrease by 7.7–42.0%, excluding offline selector-training costs. Further analyses show that removing verification or persistent storage weakens the gains, while applying verified corrections to unsuitable decisions can reverse them. Code will be released upon acceptance.

1 INTRODUCTION

Agent memory turns interaction into experience that can guide future tasks. A coding agent remembers test feedback, a theorem-proving agent retains proof attempts, and a database agent learns from query results. For example, Reflexion stores verbal reflections on these outcomes (Shinn et al., 2023); ExpeL extracts reusable insights from trajectories (Zhao et al., 2024). Existing memory mechanisms mainly describe factual trajectories: the actions an agent took and the feedback it received. Retaining this experience can reduce the need to rediscover a useful solution.

The same trajectory leaves another source of experience unexplored: *what if the agent had taken a different action at that decision?* Observing that action A failed does not establish whether action B would have improved the outcome. A reflection may propose B, but without checking it, the agent may miss the opportunity of storing a plausible suggestion as useful experience. The solution is to construct memory from evaluated alternatives, so that later tasks can benefit from corrections absent from the original trajectory.

However, obtaining this counterfactual feedback is not always straightforward. Trying alternatives in an active environment can consume additional interactions or alter the state needed for comparison. Even when actions can be replayed, evaluating every alternative is costly. Many computational tasks offer a more controlled setting: repository tests check patches (Jimenez et al., 2024), proof assistants check proofs (Zheng et al., 2022), and SQL benchmarks check query results (Li et al., 2023). When the decision state can be copied or reset, these tools can evaluate a bounded set of alternatives without

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.

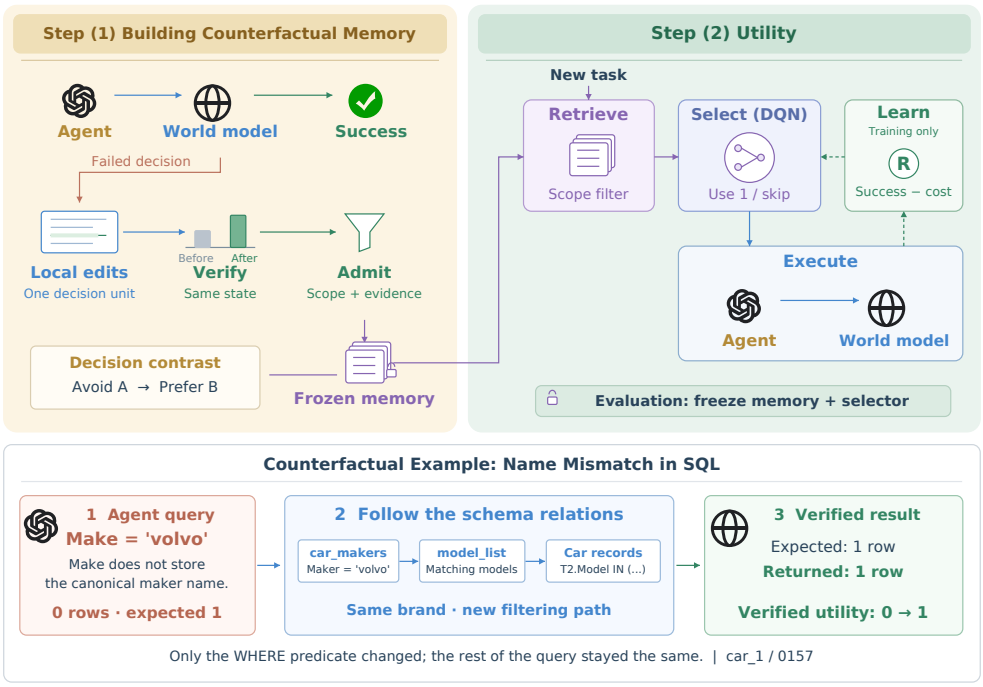


Figure 1: **Constructing and using counterfactual memory.** Local alternatives are checked from a copy or reset of the original state. Verified improvements become records containing the action contrast, evidence, and applicability conditions. On later tasks, retrieval supplies compatible records, and a trained DQN selector chooses one or skips memory. Memory and selector parameters are frozen during held-out evaluation.

committing them to the original trajectory. We use such executable evaluators as world models to enable counterfactual evaluation.

Counterfactual feedback becomes more useful across tasks when the agent preserves what changed and when the correction applies. A correction verified on one task may not apply to a similar task with different conditions. This motivates two decisions: *which evaluated alternatives should become memory, and when should that memory guide a later action?*

Motivated by the above problems, we introduce COUNTERMEM, a reinforcement-learning framework for constructing and using verified counterfactual memory across tasks (Figure 1). After a failed decision, it proposes local edits, evaluates them from the same state, and stores improvements with their action contrasts, checked outcomes, and applicability conditions. A memory-use policy learns to select a retrieved record or skip memory, with rewards balancing task success and interaction cost. This assigns credit to useful reuse rather than assuming that every verified record will help. The base LLM remains fixed. Construction and policy training exclude evaluation tasks, and both memory and policy are frozen for held-out evaluation.

With gpt-oss-120b, COUNTERMEM improves both agents on all 12 benchmarks across six domains, averaging +12.6 points. Across two backbones, it reduces task-run tokens by 7.7–42.0%. Further studies show that effective reuse requires retaining useful records and matching verified corrections to appropriate decisions. Our contributions are:

- **Constructing memory beyond factual trajectories.** We turn checked local alternatives into persistent action contrasts with supporting evidence and conditions for reuse (Section 3).
- **Learning when counterfactual memory is useful.** We formulate record selection and skipping as an RL problem, linking source-task evidence to later success and interaction cost (Section 3.4).
- **Testing memory quality as well as task performance.** Experiments measure gains, accumulation, and transfer; interventions examine the roles of verification, state matching, and applicability conditions (Section 5; Section 6).

2 Related Work

Agent memory, reflection, and counterfactual experience. Agent memory retains reflections, experiences, workflows, and reusable strategies across tasks, as in Reflexion (Shinn et al., 2023), ExpeL (Zhao et al., 2024), Generative Agents (Park et al., 2023), Voyager (Wang et al., 2024), MemGPT (Packer et al., 2023), Agent Workflow Memory (Wang et al., 2025), A-Mem (Xu et al., 2025), and ReasoningBank (Ouyang et al., 2026). Counterfactual reasoning and search instead evaluate alternative outcomes or actions (Vashishtha et al., 2026; Antoniades et al., 2025). CounterMem connects these directions by evaluating local action alternatives at the same decision state and retaining their checked outcome contrasts and applicability conditions for reuse across future tasks.

World models and executable feedback. World models support decision making by predicting or simulating action consequences for planning (Hafner et al., 2019; 2020). Recent agent systems extend this idea through language-based environment models (Chae et al., 2025) and executable simulators (Lehrach et al., 2026), while executable tests provide direct feedback for software agents (Jimenez et al., 2024). CounterMem uses task-specific tests, checkers, and solvers as executable world models to compare local alternatives at the same decision state and preserve verified contrasts as cross-task memory. Unlike current-task planning, these comparisons are retained with conditions for later reuse, while their evidence remains bounded by the tested state and available checks.

3 Methodology

3.1 What the Agent Remembers

CounterMem saves tested corrections from earlier tasks to help an agent solve later ones (Figure 1). The *base LLM* writes actions, such as SQL queries or code edits, and an *evaluator* checks their results. When an edit improves a failed action, the *memory store* can save what changed, the check results, and the conditions for using that correction. On a later task, a *selector* chooses one stored record or skips memory. The LLM uses the selected record to revise its draft action before execution.

We first build memory on *source tasks*, which supply the corrections. We then fix the store and train the selector to choose when to use those records. During testing, both the store and selector remain fixed. The base LLM's parameters remain unchanged throughout.

For example, a query for people aged *at least* 18 may incorrectly use `age > 18`. CounterMem tests `age >= 18` from the same database state. If the result checker confirms improvement, the method can save the original and corrected predicates, their results, and the requirement to include the boundary value. A later "at least" query can use this record to correct the same mistake. A query for ages *strictly above* 18 should keep the strict comparison. Modules I–II build these records; Module III retrieves them and decides whether they should guide a new action.

3.2 Module I: Propose Alternatives to a Failed Action

During memory construction, the agent takes action a_t in state s_t, which includes the current task, environment, partial solution, and feedback so far. The evaluator H returns the outcome $o_t = H(s_t, a_t)$. We use the term *world model* for this evaluator because it checks what happens when an action is tried. Depending on the task, H is a test suite, proof checker, SQL result checker, or solver. Its feedback comes from executing these checks.

When an action fails and checking budget remains, a generator proposes at most $K = 4$ local alternatives:

$$\mathcal{A}_{\text{cf}} = G(s_t, a_t, o_t; K). \tag{1}$$

The generator G uses LLM edits or rules for the observed failure to change one part of the action, such as a SQL condition, proof step, or code expression. Each resulting action is a candidate for evaluation. These alternatives ask a *counterfactual* question: what would have happened if the agent had taken this edited action instead?

3.3 Module II: Check Alternatives and Store Corrections

Each alternative a' is evaluated from a copy or reset of s_t. This lets us compare the edited and original actions from the same starting state:

$$\Delta(a') = U(H(s_t, a')) - U(o_t). \tag{2}$$

U converts evaluator feedback to a score in $[0, 1]$, such as correctness or the fraction of tests passed. Thus, Δ measures the improvement due to the edit. A candidate qualifies for storage only if its check completes and $\Delta(a') > \epsilon_{\text{adm}} \geq 0$, where ϵ_{adm} is the minimum required gain. An improved partial score may qualify even if the whole task remains unsolved.

An accepted correction becomes a record $m = (q, a^-, a^+, e, c, u)$. The record describes the situation q, the original action a^-, the improved action a^+, and their check results e. The field c states the task requirements under which the correction applies. In the SQL example, q describes a missed boundary value, while c requires an inclusive request such as "at least" and compatible column types. Retrieval uses q to find related records and checks c against the new task before offering them for use. The evidence e compares query results; successful execution alone does not establish improvement.

The remaining field u summarizes what happened when this record was reused on subsequent source tasks during construction. It starts at zero and is updated when reuse outcomes are observed. These observations precede selector training: once construction ends, the records and their reuse statistics are fixed. We also retain each record's source-task identifier so that retrieval can exclude records from the current task.

The store holds at most $B = 200$ records in the specified configuration. Admission rules check proposed records for duplicate or conflicting advice. When space is needed, retention rules decide which records to remove (Appendix B). Saving a correction makes it available for later tasks; it does not automatically replace the action in its source task.

3.4 Module III: Retrieve a Record and Decide Whether to Use It

On a later task, the LLM first drafts an action $\hat{a}_t$. Retrieval excludes records from the same task and filters out records whose conditions c do not fit the current task and environment. It ranks the remaining records by the relevance of their situation q and their stored reuse statistic u, returning at most $k = 3$ candidates.

The selector decides whether one of these candidates would help revise the current draft. A related record may be unnecessary: for example, the draft may already include the requested boundary value. If the selector chooses a record, the LLM adapts its correction to the draft. If the selector skips memory, the draft proceeds unchanged. With no eligible record, it must skip. The evaluator checks the resulting action in either case.

We train the selector as a deep Q-network (DQN) (Mnih et al., 2015). At each training decision, it receives the current state, draft, candidate records, and remaining budgets. Its choices are to supply one candidate or skip memory. Training rewards task completion and penalizes evaluator calls, failed attempts, and token use, teaching the selector to score these choices by their expected reward.

The three quantities serve different purposes: Δ measures how much the original edit helped on its source task; u summarizes observed reuse during construction; and the selector predicts the reward from using a record at the current decision. Appendix C gives the reward and training rules. During testing, the fixed selector chooses the valid option with the highest predicted reward.

4 Experimental Setup

We compare memory mechanisms, then vary source experience, record retention, and the tasks on which memory is used. We use success and cost to measure whether earlier corrections help solve new tasks.

4.1 Tasks, Agents, and Memory Baselines

Tasks and checks. The suite contains 12 benchmark settings in six domains: mathematics, coding, formal verification, text-to-SQL, SMT, and SAT. A task asks the agent to produce a proof, repair, verified program, query, or solver solution. Lean checks proofs (de Moura & Ullrich, 2021); repository tests check repairs (Jimenez et al., 2024); Dafny and Verus check programs (Leino, 2010; Lattuada et al., 2023); SQL results are compared with reference results (Li et al., 2023); and SMT/SAT solvers check solver inputs and outcomes (Barrett et al., 2016; Hoos & Stützle, 2000). Appendix D.1 maps each benchmark to its checker.

Main comparison. Table 1 crosses ReAct (Yao et al., 2023) and Reflexion (Shinn et al., 2023) as base agents with gpt-oss-120b (OpenAI, 2025) and Gemini 3.1 Flash-Lite (Google, 2026) as base LLMs. Each group compares the following memory mechanisms: no augmentation, ExpeL, Generative Agents, Voyager, MemGPT, and CF Memory; Best-of-N provides a separate compute control. Within each group, comparing a row with "None" measures its gain over the base agent; comparing augmented rows tests the different added memory mechanisms. Table 1 reports four-domain results for Math, Coding, Text-to-SQL, and SAT; the full 12-benchmark evaluation across six domains, including formal verification and SMT, is reported in Appendix D.7.

The base agent is the procedure for using the LLM to solve tasks. ReAct asks the LLM to propose actions and then respond to tool feedback. Reflexion also asks it to write reflections on earlier attempts and use them in later attempts. "None" adds no memory mechanism and keeps these reflections. CF Memory adds tested corrections, even when reflections are already available.

What the added mechanisms provide. ExpeL supplies lessons from past attempts (Zhao et al., 2024); Generative Agents supplies stored experiences and reflections (Park et al., 2023); Voyager supplies reusable procedures (Wang et al., 2024); MemGPT manages which information stays in context or external storage (Packer et al., 2023). We adapt these memory components to a common task interface. CF Memory supplies checked corrections through the full CounterMem process: build records, retrieve candidates, and choose one or skip. Best-of-N generates extra candidates for the current task without keeping memory, asking whether extra trials can match reuse. ReasoningBank extracts strategies from past attempts (Ouyang et al., 2026); it is the memory baseline in the accumulation, transfer, and fixed-budget studies.

4.2 Build Memory, Train Selection, Then Test

Construction tasks produce the stored records. On source tasks, the agent tries actions; after a failure, CounterMem tests local edits and saves accepted corrections, as in Modules I–II. The environment and temporary working state reset between tasks; saved records carry corrections forward, and their reuse statistics can be updated during this stage. Each benchmark, LLM, agent, configuration, and seed has its own store.

Training tasks teach the selector to use that store. The store, including reuse statistics, is now fixed. The DQN tries record choices or skipping, observes task success and cost, and updates only its selection parameters. Training uses 2,000 decisions. Appendix C gives the training settings and task-set separation.

Held-out tasks measure reuse of earlier experience. Held-out tasks appear in neither construction nor selector training. The agent can use feedback to revise its answer within a test task, but records, reuse statistics, and selector parameters stay fixed across test tasks. Success therefore measures use of experience collected before testing. Appendix D.2 gives the SQL split example.

4.3 What the Additional Experiments Change

Accumulation and retention. We compare CounterMem and ReasoningBank after 0, 25, 50, 100, and 200 source tasks, evaluating each saved store on held-out tasks. Checkpoints count processed tasks rather than stored records: one task may yield zero or multiple records, and records can be replaced once capacity is reached. We separately compare recent, random, similarity-ranked, and utility-ranked retention at equal record capacity. Appendix D.3 gives further details.

Table 1: **Four-domain comparison of memory mechanisms across agents and backbones.** Scores (0–100, higher is better) average benchmarks within each domain; tokens report total task-run consumption in millions. Bold marks the best value within each agent–backbone group.

Base Agent	Augmentation	Math ↑	Coding ↑	Text-to-SQL ↑	SAT ↑	Tokens(M) ↓
Backbone: gpt-oss-120b						
ReAct	None	29.8	48.7	26.8	50.7	6.0
	Best-of-N (compute control)	34.5	38.1	36.2	41.1	9.4
	ExpeL (Zhao et al., 2024)	27.4	56.5	17.2	49.0	5.4
	Generative Agents (Park et al., 2023)	22.6	37.1	17.6	41.6	6.2
	Voyager (Wang et al., 2024)	21.9	60.6	26.3	51.4	5.2
	MemGPT (Packer et al., 2023)	27.3	39.5	**48.9**	47.0	5.8
	CF Memory	**57.5**	**66.6**	47.0	**58.9**	**4.4**
Reflexion	None	40.8	34.6	54.6	61.2	5.2
	Best-of-N (compute control)	31.8	21.7	62.5	53.6	10.0
	ExpeL	38.0	32.8	**68.1**	60.0	5.0
	Generative Agents	36.7	38.8	60.4	48.3	5.6
	Voyager	37.7	**42.6**	55.4	58.3	**4.5**
	MemGPT	48.0	39.1	47.5	58.0	5.7
	CF Memory	**53.4**	40.7	66.1	**63.7**	4.8
Backbone: Gemini 3.1 Flash-Lite						
ReAct	None	38.5	30.9	57.6	45.0	5.0
	Best-of-N (compute control)	30.7	29.0	67.3	34.9	9.6
	ExpeL	**57.4**	37.3	68.6	37.2	4.8
	Generative Agents	42.4	31.3	57.2	43.9	5.7
	Voyager	39.9	34.4	48.2	55.1	4.3
	MemGPT	31.4	38.2	68.5	45.2	5.5
	CF Memory	54.5	**40.0**	**71.7**	**57.9**	**2.9**
Reflexion	None	55.5	47.7	59.1	43.7	4.3
	Best-of-N (compute control)	58.7	43.9	65.6	54.7	8.2
	ExpeL	50.3	43.2	70.2	36.8	4.0
	Generative Agents	44.6	36.6	66.6	**66.1**	4.9
	Voyager	54.9	37.1	49.0	32.1	3.9
	MemGPT	43.8	38.2	52.0	49.5	4.8
	CF Memory	**74.0**	**73.5**	**77.8**	62.3	**3.4**

Transfer and ablations. We compare CounterMem and ReasoningBank under four changes to the next task: a new request in the same environment, a request in a different environment, the same error expressed differently, and a look-alike request that makes the old correction wrong. An environment is the execution setting, such as a database or repository. For a look-alike example, changing > to >= is wrong when the new request says "strictly above." We measure success, whether guidance is retrieved, and harmful use. Ablations separately remove checks or storage, or change which records reach the selector (Section 6).

4.4 Evaluation Metrics

Scores use benchmark success measures on a 0–100 scale; differences are percentage points (pp). Domain columns average their benchmarks. Four-domain summaries weight domains equally; 12-benchmark summaries weight benchmarks equally.

Total tokens count reasoning, retrieval, and augmentation in the reported task runs, in millions (M). Evaluator calls include counterfactual checks; failed attempts are evaluated actions that fail the success criterion. Interaction plots divide total calls or failed attempts by the number of solved tasks; a lower ratio need not mean fewer total calls. Accumulation panel (b) counts only retrieval and context tokens per task. Offline selector training is not separately costed. Best-of-N has a different token total; a separate 6M-token comparison appears in Appendix E.2. Transfer-rate definitions and reporting limits are in Appendix D.4.

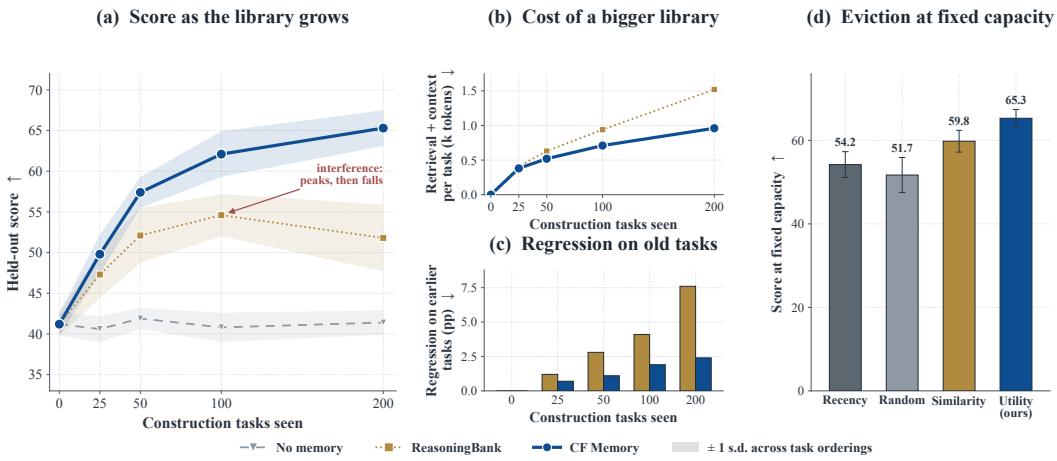


Figure 2: **Learning from more source tasks and choosing what to keep.** (a) Test score at successive memory-construction checkpoints. (b) Retrieval and context tokens per task; other solving costs are excluded. (c) Reported score loss on earlier tasks. (d) Test score under different record-retention rules at equal capacity. The horizontal axis in (a–c) counts source tasks, not stored records; shading shows one standard deviation across task orderings.

5 Results

5.1 Does Earlier Experience Help Solve New Tasks?

Within each agent–LLM group in Table 1, we first compare CF Memory with "None." For example, averaging the four domain columns for gpt-oss-120b with ReAct gives 39.0 without augmentation and 57.5 with CF Memory, a gain of 18.5pp. Gains across four groups range from 8.2 to 20.4pp. Reflexion gains 8.2pp with gpt-oss-120b and 20.4pp with Gemini 3.1 Flash-Lite, so the added corrections help even when reflections are available. Comparing all rows within each group, CounterMem has the highest four-domain mean in all four groups and leads in 11 of 16 domain comparisons.

Task-run tokens fall by 7.7–42.0% relative to "None". Best-of-N uses 8.2–10.0M tokens, compared with 2.9–4.8M for CounterMem, yet scores lower on average in every combination. The reported Best-of-N control therefore does not match CounterMem's mean score, despite using more task-run tokens.

5.2 What Changes as the Agent Encounters More Source Tasks?

Moving right in Figure 2(a) means testing memory collected after more source tasks. CounterMem's held-out score rises, while ReasoningBank's first rises and then falls. The latter decline shows that processing more source tasks may not necessarily make the stored guidance more useful. Retrieval and context costs rise for both (b). CounterMem also loses less performance on earlier tasks at each nonzero checkpoint (c), though neither fully preserves it. Appendix D.3 gives the checkpoint protocol and reporting details.

Panel (d) asks a different question: with the same record limit, which records should be kept? The utility rule favors corrections with larger checked improvements and more helpful reuse during construction. It scores 65.3, versus 59.8 for similarity, 54.2 for recency, and 51.7 for random retention. Its 5.5-point advantage over similarity supports utility-aware retention when storage is limited; the advantage does not come from a larger record capacity.

5.3 When Does a Stored Correction Apply to a New Task?

Figure 3 separates finding guidance (coverage), solving the task (success), and worsening an outcome through memory use (harm). Within each of the four conditions, CounterMem has higher success than ReasoningBank. Across conditions, its own score falls from 67.4 in the original environment to 52.6 in a new one, which indicates that earlier corrections remain useful, but their benefit does

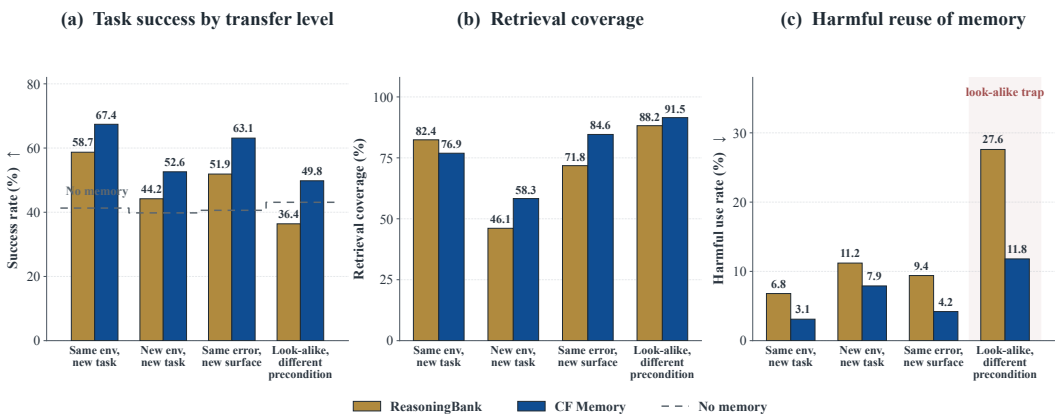


Figure 3: **Reusing memory under four changes to the task.** (a) Task success, with a no-memory reference. (b) Frequency of obtaining retrieved guidance. (c) Reported harmful-use percentages. "New surface" changes how the same error appears; "look-alike" preserves similarity while changing the condition needed for the correction. Definitions and reporting details are in Appendix D.4.

not remove the transfer difficulty. In the original environment, higher success accompanies lower coverage (76.9% versus 82.4%), meaning that finding guidance more often is insufficient to explain which method succeeds more often.

The look-alike tasks test whether the agent avoids a familiar correction when the new request makes it wrong, as in the SQL example in Section 3. Both methods have high reported coverage (88.2% for ReasoningBank; 91.5% for CounterMem), but CounterMem achieves 49.8% success versus 36.4%. Reported harmful use falls from 27.6% to 11.8%. To test the role of the stored condition, a separate intervention removes the applicability field c from CounterMem records. Harmful use rises from 11.8% to 34.2% (Appendix E.3). This experiment supports the use of explicit conditions to avoid applying a correction to the wrong request.

5.4 Does Better Performance Require Fewer Evaluations?

For each of 12 benchmarks with gpt-oss-120b, we compare ReAct and Reflexion with their own CF Memory versions. Scores improve in all 24 comparisons (mean +12.6 pp; Figure 5 in Appendix D.7). Costs vary: SWE-bench Verified uses 3.6/1.2 fewer calls per solved task for ReAct/Reflexion, while Vericoding-Verus with ReAct gains 9.6pp but uses 0.5 more calls. Thus, the score gains are consistent across benchmarks, whereas interaction-cost changes are benchmark dependent.

6 Further Analysis

We first change how corrections are checked and saved, then change which decisions receive them. All comparisons use gpt-oss-120b; paired numbers report ReAct followed by Reflexion. Score gains use each agent's own unaugmented version as the reference.

6.1 What Do Verification and Persistent Storage Contribute?

"CF without WM" removes world-model-based checks before storage, saving proposed corrections as untested advice. "WM without CF" checks alternatives within the current task but saves no correction records for later tasks. These comparisons ask whether suggestions without checks, or checks without storage, recover the full gain. Both variants still evaluate ordinary task actions.

Unverified advice gives +5.2/+2.1pp for ReAct/Reflexion; testing alternatives without saving records gives +7.1/+3.4pp. The full method reaches +18.5/+8.2pp (Figure 4(a)). Unverified advice also uses more calls per solved task than the base agents: 10.4/9.9 versus 9.6/9.1. The full method uses 7.0/8.4 (panel (b)). Neither untested advice nor unsaved alternatives recover the full gain. This experiment supports our design of both checking corrections and retaining them for reuse.

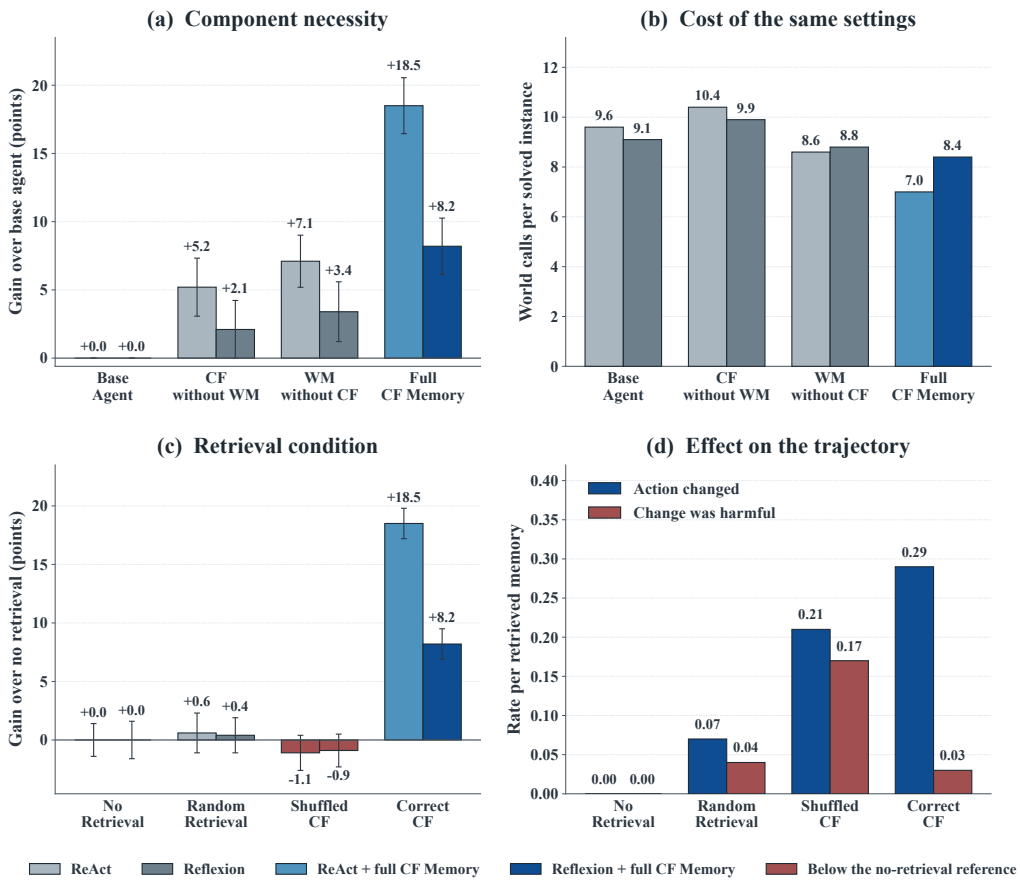


Figure 4: **Removing components and changing record assignment.** (a) Score gains over the base agent. "CF without WM" stores unverified advice; "WM without CF" checks current-task alternatives without keeping cross-task corrections. (b) Evaluator calls per solved task. (c) Score gains when candidate records are absent, random, shuffled across states, or matched to the current decision. (d) Action changes and harmful changes per retrieved record.

6.2 What Happens When a Valid Record Reaches the Wrong Decision?

We vary where stored corrections are offered. Random retrieval ignores state matching, shuffled retrieval reassigns records to other decision states, and matched retrieval uses the stored situation and conditions. Thus, a record can remain valid on its source task yet be unsuitable for the current decision. Score changes are relative to no retrieval.

Random retrieval gives only +0.6/+0.4pp and shuffling -1.1/-0.9pp, versus +18.5/+8.2pp with matched retrieval (panel (c)). In panel (d), shuffling yields 0.21 action changes and 0.17 harmful changes per retrieved record, compared with 0.29 and 0.03 for matched retrieval. Matched records therefore change actions more often but cause less harm, supporting the importance of matching guidance to the current decision rather than merely limiting how often memory changes actions. See Appendix D.4.

7 Conclusion

CounterMem turns tested corrections from earlier tasks into guidance for later ones. It checks edits to failed actions, saves what worked better and when it applies, and trains a selector to offer one record to the LLM or let it act without memory. Across 12 benchmarks in six domains, CounterMem improves both base agents, averaging +12.6 points with gpt-oss-120b; across two backbones, it reduces task-run tokens by 7.7–42.0%. Further analyses show that useful counterfactual memory requires both verified corrections and appropriate reuse: mismatched records can erase gains or cause harm.

8 AI USE STATEMENT(DOES NOT COUNT TOWARDS THE PAGE LIMIT)

The authors used ChatGPT to revise the manuscript's language, clarify explanations, and reduce repetition. The authors reviewed the revised content and take responsibility for the submitted manuscript. For manuscript editing, the models were used primarily for language polishing and concise clarification, for example by replacing awkward phrasing with shorter equivalent expressions, adding brief descriptive phrases such as "as an LLM agent" when needed for readability, or rewriting a sentence without changing its technical meaning. The models were not used as a substitute for experimental validation or scientific judgment. All AI-assisted outputs were reviewed by the authors, and the reported experimental configurations and results were checked against the authors' run artifacts and checkpoints. The authors take responsibility for the final content of the paper, including all claims, results, code, and AI-assisted text or artifacts.

9 ETHICS STATEMENT(DOES NOT COUNT TOWARDS THE PAGE LIMIT)

Persistent memory can carry an inaccurate or outdated correction into a later task. Checking that correction on its source task does not establish that it is safe to reuse under different requirements or permissions. Records should retain their source and scope, and systems should protect sensitive information contained in stored trajectories. The benchmark results measure task performance; they are not a deployment safety guarantee.

10 REPRODUCIBILITY STATEMENT(DOES NOT COUNT TOWARDS THE PAGE LIMIT)

This appendix records the configuration used by the reported runs at the level needed to interpret the method and diagnostic figures. In particular, it specifies the memory thresholds and retention weights, the selector architecture and optimizer settings, the five selector seeds and checkpoint rule, the fixed-selector policy used across accumulation snapshots, the transfer sample counts and rate denominators, the c-removal intervention, and the prompt/parser interfaces. These settings were checked against the run checkpoints used to reproduce the reported outputs.

The manuscript package also supplies figures, numerical summaries, and plotting scripts for checking reported arithmetic. A complete independent rerun still requires the task-level run manifests, experiment runner, benchmark environments, evaluator traces, and trained checkpoint files; offline selector-training cost is not included in the reported task-run token totals.

REFERENCES

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a1e6783e4d739196cad3336f12d402bf-Abstract-Conference.html.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. *arXiv preprint arXiv:2302.12433*, 2023. URL https://arxiv.org/abs/2302.12433.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). SMT-LIB initiative, 2016. URL https://smt-lib.org/.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A Benchmark for Vericoding: Formally Verified Program Synthesis. *arXiv preprint arXiv:2509.22908*, 2025. URL https://arxiv.org/abs/2509.22908.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web Agents

with World Models: Learning and Leveraging Environment Dynamics in Web Navigation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a00548031e4647b13042c97c922fadf1-Abstract-Conference.html.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI research report, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Google. Gemini 3.1 Flash-Lite. Google AI for Developers, 2026. URL https://ai.google.dev/gemini-api/docs/models/gemini-3.1-flash-lite. Accessed September 23, 2026.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning Latent Dynamics for Planning from Pixels. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2555–2565. PMLR, 2019. URL https://proceedings.mlr.press/v97/hafner19a.html.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to Control: Learning Behaviors by Latent Imagination. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1lOTC4tDS.

Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In Ian P. Gent, Hans van Maaren, and Toby Walsh (eds.), *SAT 2000: Highlights of Satisfiability Research in the Year 2000*, pp. 283–292. IOS Press, 2000. URL https://www.cs.ubc.ca/~hoos/Publ/sat2000-satlib.pdf.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037.

Wolfgang Lehrach, Daniel Hennes, Miguel Lázaro-Gredilla, Xinghua Lou, Carter Wendelken, Zun Li, Antoine Dedieu, Jordi Grau-Moya, Marc Lanctot, Atil Iscen, John Schultz, Marcus Chiam, Ian Gemp, Piotr Zielinski, Satinder Singh, and Kevin P. Murphy. Code World Models for General Game Playing. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/d8a12fde9e72444e1b356e8c37e53753-Abstract-Conference.html.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36, pp. 42330–42357. Curran Associates, Inc., 2023. doi: 10.52202/075280-1835. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/

83fc8fab1710363050bbd1d4b8cc0021-Paper-Datasets_and_Benchmarks.pdf.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A Benchmark for Formal Software Verification. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=yBgTVWccIx.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236. URL https://www.nature.com/articles/nature14236.

OpenAI. gpt-oss-120b & gpt-oss-20b Model Card, 2025. URL https://arxiv.org/abs/2508.10925.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

SAT Competition Organizers. SAT Competition: Benchmarks and Competition Archive. Official competition site, 2026. URL https://satcompetition.github.io/2026/. Accessed September 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

SMT-COMP Organizers. SMT-COMP: The International Satisfiability Modulo Theories Competition. Official competition site, 2026. URL https://smt-comp.github.io/2026/. Accessed September 2026.

Aniket Vashishtha, Qirun Dai, Hongyuan Mei, Amit Sharma, Chenhao Tan, and Hao Peng. Executable Counterfactuals: Improving LLMs' Causal Reasoning Through Code. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/a50aa557c4be35aa2bf13a471601e23f-Abstract-Conference.html.

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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1556–1560. Association for Computing Machinery, 2020. doi: 10.1145/3368089.3417943. URL https://doi.org/10.1145/3368089.3417943.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic Memory for LLM Agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-0593. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.

Bowen Yang, Yi Yuan, Chenyi Li, Ziyu Wang, Liangqi Li, Bo Zhang, Zhe Li, and Zaiwen Wen. Construction-Verification: A Benchmark for Applied Mathematics in Lean 4. *arXiv preprint arXiv:2602.01291*, 2026. URL https://arxiv.org/abs/2602.01291.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921. Association for Computational Linguistics, 2018. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425/.

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. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: a Cross-System Benchmark for Formal Olympiad-Level Mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

APPENDIX CONTENTS

A Limitations and Future Work . . . 15
B Memory Construction and Retrieval Rules . . . 15
B.1 Verification and Admission . . . 15
B.2 What a Record Contains . . . 15
B.3 Capacity and Reuse . . . 16
B.4 From Stored Records to a Selection . . . 16
C Reinforcement Learning for Memory Use . . . 17
C.1 Decision Process and Observations . . . 17
C.2 Reward and Q-Learning . . . 18
C.3 Construction, Training, and Evaluation Procedure . . . 19
D Experimental Details . . . 19
D.1 Benchmarks and Evaluators . . . 19
D.2 Held-Out SQL Configuration . . . 19
D.3 Accumulation Checkpoints and Record Retention . . . 20
D.4 Transfer Conditions and Reuse Measurements . . . 20
D.5 Baselines and Interventions . . . 21
D.6 Metric Conventions . . . 22
D.7 Values Underlying the Benchmark Breakdown . . . 22
D.8 Prompt and Parser Interfaces . . . 22
E Additional Analyses . . . 24
E.1 Backbone Summary . . . 24
E.2 Comparison at a Common 6M-Token Budget . . . 24
E.3 Rescued Tasks, Broken Tasks, and Applicability Conditions . . . 24

A Limitations and Future Work

What verification establishes. The method requires executable checks and a way to reset or copy the state before testing a local edit. A test suite may omit cases, and a reference check may cover only one input. A checked improvement supports using the correction in that setting; it does not prove correctness on every future task. Small edits can also miss failures that require a different plan.

What memory can represent and reuse. A record can become stale when tools, schemas, or requirements change. The transfer experiments show that a correction valid at its source can harm a different task. The selector sees compressed features and chooses at most one record, so it cannot directly combine several records into a single correction.

What the experiments establish. The comparisons evaluate the complete method. They do not isolate DQN's benefit against a fixed selector. The appendix specifies the principal reported-run configuration, while exact task manifests, evaluator traces, and execution environments remain separate run artifacts (Appendix 10). Total task-run tokens, calls per solved task, and retrieval overhead measure different costs; none alone establishes lower total training cost, runtime, financial cost, or safe deployment.

Future work. The limitations above motivate three direct extensions: combining multiple verified records when one correction is insufficient, detecting or retiring stale records when tools or task requirements change, and reporting end-to-end training and runtime costs alongside task-run interaction metrics.

B Memory Construction and Retrieval Rules

B.1 Verification and Admission

This appendix expands the admission, retention, and retrieval steps. Admission decides whether to save a newly checked comparison; retention decides what to remove when space is needed; retrieval finds candidates for a current decision. The selector then decides whether to use one.

For the original action's outcome $o = H(s, a)$ and an edited action's outcome $o' = H(s, a')$, let $\Delta = U(o') - U(o)$. Both actions are evaluated from the same state. A record can be admitted only if the candidate evaluation completes and $\Delta > \epsilon_{\text{adm}} \geq 0$. The reported runs use $\epsilon_{\text{adm}} = 0.05$, applied to the raw utility gain Δ before any normalization. With binary correctness this means replacing a failed outcome with a successful one; partial-credit improvements need not solve the full task. The record is

$$m_i = (q_i, a_i^-, a_i^+, e_i, c_i, u_i). \tag{3}$$

Its source-task identifier is retained as metadata. The first five fields describe the situation, original action, improved action, check results, and conditions for using the correction. The statistic u_i starts at zero and summarizes reuse outcomes observed while construction remains active. It is frozen with the record before selector training and held-out testing.

Among records that pass verification, this score determines which to save:

$$S_{\text{adm}}(m_i) = \lambda_\Delta \widetilde{\Delta}_i + \lambda_u \widetilde{u}_i - \lambda_r R(m_i, \mathcal{M}) - \lambda_I I(m_i, \mathcal{M}), \tag{4}$$

where tildes denote normalized quantities, R measures repeated advice, and I measures conflicting advice. The score favors larger checked gains and helpful reuse, and penalizes repetition and conflict. A record enters only if its score exceeds τ_{adm}. An unverified record cannot enter, however useful it might appear.

B.2 What a Record Contains

Table 2 expands the running SQL example. It is an illustration of the record format, not an additional measured test case. A condition states when to try the correction. Matching that wording does not guarantee that the correction will solve a future task.

Table 2: Illustrative memory record for an inclusive age boundary.

Field	Content and purpose
q: situation	A numeric threshold predicate may omit its boundary value.
a^-: inferior action	The original query uses `age > 18`.
a^+: improved action	Replace that predicate with `age >= 18`.
e: evidence	The original and edited queries' result-check outcomes, from the same database state.
c: conditions	The question requests "at least" the threshold; the column and comparison types are compatible.
u: reuse statistic	Initially zero; later summarizes observed helpful, neutral, or harmful reuse during construction.
Source identifier	Identifies the task that supplied the comparison, allowing same-task retrieval to be excluded.

B.3 Capacity and Reuse

Capacity B is a limit on the number of correction records, not on source tasks, tokens, or model parameters. The store holds at most B records; processing another task need not add a record if no candidate passes verification and admission. When capacity is exceeded, retention ranks records by

$$S_{\text{keep}}(m_i) = \alpha\widetilde{\Delta}_i + \beta\widetilde{u}_i + \omega F_i - \eta R(m_i, \mathcal{M}), \tag{5}$$

Here $\widetilde{\Delta}_i$ measures improvement on the source action, $\widetilde{u}_i$ summarizes later reuse outcomes, F_i is normalized reuse frequency, and R penalizes redundant records. The coefficients α, β, ω, and η weight improvement, reuse utility, frequency, and redundancy. The reported utility-retention runs use $(\alpha, \beta, \omega, \eta) = (1.0, 1.0, 0.5, 0.7)$. The lowest-ranked records are removed. During construction, when records can still be updated, the score summarizing a record's reuse is updated as

$$u_i^{(j+1)} = (1-\rho)u_i^{(j)} + \rho y_{i,j}, \tag{6}$$

Here j indexes reuse observations and $y_{i,j} \in \{-1, 0, +1\}$ labels harmful, neutral, or beneficial reuse. In the reference implementation, this label is assigned relative to the memory-free draft at the same decision. If the executed action is unchanged from that draft, the reuse is neutral. If memory changes the action and the task is solved, the reuse is beneficial. Otherwise, when a previous evaluated attempt is available, the label follows the sign of the utility change relative to that attempt; if no such reference is available, the label is neutral. This definition avoids penalizing a retrieved record merely because it was shown on a difficult task without changing the submitted action. Records and these statistics remain unchanged in held-out evaluation. The three scores answer different questions. Δ_i asks whether the edit improved its original action. u_i summarizes what happened when the saved correction was reused during construction. The DQN value Q_θ predicts the reward from choosing that record at the current decision; training this prediction leaves Δ_i and u_i fixed.

B.4 From Stored Records to a Selection

For task x and state s, define eligible records as

$$\mathcal{C}(x, s) = \{m_i \in \mathcal{M} : x_i^{\text{source}} \neq x,\ c_i \text{ is compatible with } s\}. \tag{7}$$

Retrieval selects up to k eligible records using

$$r_i(s) = \text{sim}(\phi(s), \phi(q_i)) + \lambda_{\text{reuse}} u_i, \qquad \mathcal{R}_k = \text{TopK}_{m_i \in \mathcal{C}(x,s)}\, r_i(s), \tag{8}$$

where ϕ represents text for similarity comparison. The first term matches the current situation to the record; the second favors records whose past use helped, with weight λ_{reuse}. The ordered list $\mathcal{R}_k$ is passed to the selector as $\mathcal{R}_t$. The selector μ_θ gives the LLM one record or none (Section 3.4); the evaluator checks the resulting action.

Reference implementation details. The generator uses the same backbone as the task-solving agent rather than a separate model. Counterfactual generation is triggered only after a failed decision while verification budget remains, with at most two failed decisions expanded per task. The implementation first asks the model for local edits and uses rule-based edit operators only to fill unused candidate

slots up to $K = 4$. Thus the generator changes one decision unit at a time; it does not replace the complete solution with an independently generated submission.

For retrieval, the implementation embeds the record situation q_i and the current task prompt plus state, and ranks them with cosine similarity before adding the reuse term in Eq. (8). The default embedder is a 512-dimensional character n-gram hashing representation. This is an implementation choice for the reported system, not a claim that this representation is uniquely required by the method.

The specified defaults refer to different stages:

Symbol	Value	What it limits
K	4	Alternatives proposed for one failed action
B	200	Correction records retained in the store
k	3	Records returned at one retrieval step
L	4	Candidate slots in the selector architecture

Only one retrieved record can be used at a decision. With $k = 3$, the fourth architectural slot is masked; it does not create a fourth retrieved record. Admission weights in Eq. (4) are $(1.0, 0.5, 0.7, 1.0)$, with threshold $\tau_{\text{adm}} = 0.1$; the raw improvement floor is $\epsilon_{\text{adm}} = 0.05$; the reuse update uses $\rho = 0.3$; and retrieval uses $\lambda_{\text{reuse}} = 0.2$. The utility-retention coefficients are $(\alpha, \beta, \omega, \eta) = (1.0, 1.0, 0.5, 0.7)$. Construction checks have a separate verification allowance from ordinary solving; both enter interaction accounting. These values are fixed in the reported runs.

The exact normalization functions used inside Eq. (4) and Eq. (5), together with the internal redundancy and conflict computations, are implementation details of the run code rather than new measured quantities. They do not change the definitions of Δ, u, or the eligibility rule in Eq. (7).

C Reinforcement Learning for Memory Use

C.1 Decision Process and Observations

The selector chooses which record, if any, the LLM sees before it acts. The LLM still writes the task action. An episode is one task. At each decision, the frozen LLM drafts an action $\hat{a}_t$, retrieval supplies $\mathcal{R}_t$, and the memory-use policy chooses $j_t \in \mathcal{J}_t = \{0, \ldots, |\mathcal{R}_t|\}$. Action zero executes the draft. Action $j_t > 0$ presents only record $\mathcal{R}_t[j_t]$ and asks the LLM to revise the draft. The resulting action is checked by H, producing the next agent state and cost counters. The policy does not choose an arbitrary subset or modify the verifier.

The selector receives a numerical description z_t of the current situation, draft action, available records, and remaining budgets:

$$z_t = \big[PE(s_t) \,\|\, PE(\hat{a}_t) \,\|\, f_{t,1} \,\|\, \cdots \,\|\, f_{t,L} \,\|\, b_t\big]. \tag{9}$$

Here, $\|$ joins the feature vectors. E converts text to vectors (it is denoted ϕ in the retrieval rule), and the fixed mapping P reduces each text vector to 32 dimensions. In the reference implementation, P is a fixed-seed Gaussian random projection from the 512-dimensional retrieval embedding to 32 dimensions (seed 20260824). It is not fitted during memory construction or selector training. The vector b_t describes the remaining budgets, using normalized values from the execution log. Each seven-dimensional record vector contains relevance to the state, admission score, estimated improvement from its evidence, reuse utility, use frequency, helpful-use fraction, and validity. Features are normalized, and an unused helpful-use fraction is set to zero. The specified controller has $L = 4$ candidate slots and six budget features, giving $32 + 32 + 4 \times 7 + 6 = 98$ dimensions. Retrieval may return fewer records: the unused slots are zero-padded and masked, including the fourth slot when $k = 3$. Slot capacity L, retrieval count k, and generation budget K are separate quantities.

The Q-network is a two-layer multilayer perceptron with hidden widths $(64, 64)$ and ReLU activations. This architecture maps the 98-dimensional observation to one value for the skip action and one value for each architectural record slot.

This compressed observation does not expose the complete agent history. The controller therefore learns under partial observation; we do not assume that z_t is a fully observed Markov state. Frozen LLM parameters also do not imply deterministic action generation.

C.2 Reward and Q-Learning

At decision t, the selector receives

$$r_t = \mathbf{1}[t = T]\mathbf{1}[\text{solved}] - \lambda_w \delta_{w_t} - \lambda_f \delta_{f_t} - \lambda_\tau \delta_{\tau_t}/\tau_0. \tag{10}$$

T is the final decision in the task. The first term pays one unit only when the task ends successfully. The nonnegative increments δ_{w_t}, δ_{f_t}, and δ_{τ_t} count evaluator calls, failed evaluated actions, and tokens charged to that decision. The λ coefficients weight the penalties, and τ_0 sets the token scale. This reward favors completing the task while spending less on unsuccessful attempts and unnecessary use of memory.

The cost increments are positive changes in the charged counters; reducing cost avoids a penalty rather than generating a separate reward. Token normalization prevents thousands of tokens from overwhelming the call penalty solely because of their unit. The local gain Δ determines eligibility for storage; adding it again as an RL reward would reward source-task improvement without establishing useful reuse.

The DQN assigns each choice j a score: its expected total future reward, with rewards farther in the future weighted by powers of γ:

$$Q_\theta(z, j) \approx \mathbb{E}\left[\sum_{\ell=0}^{T-t} \gamma^\ell r_{t+\ell} \mid z_t = z, j_t = j\right]. \tag{11}$$

Training saves each choice and its outcome as $(z_t, j_t, r_t, z_{t+1}, d_t)$, where $d_t = 1$ marks an episode's end. A periodically updated copy of the Q-network, with parameters θ^-, provides the learning target:

$$y_t = r_t + \gamma(1 - d_t) \max_{j \in \mathcal{J}_{t+1}} Q_{\theta^-}(z_{t+1}, j), \tag{12}$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\mathcal{D}}\big[(y_t - Q_\theta(z_t, j_t))^2\big]. \tag{13}$$

$\mathcal{D}$ is the replay buffer of saved training steps. The loss reduces the difference between the predicted score and target y_t. Terminal transitions have target r_t; unavailable record slots cannot be selected. During training, ε-greedy selection occasionally tries a random valid choice instead of the highest-scoring one. Held-out evaluation uses $j_t = \arg\max_{j \in \mathcal{J}_t} Q_\theta(z_t, j)$. The replay buffer stores training transitions; it is distinct from the persistent store of correction records. Only θ is optimized: the base LLM, generator, evaluator, and fixed admission rules are not trained by this objective.

Table 3: DQN configuration used for the reported memory-use selector.

Setting	Value
Training decision steps	2,000
Hidden layers	$64, 64$
Activation	ReLU
Optimizer	Adam, $\epsilon_{\text{Adam}} = 10^{-8}$
Learning rate	10^{-3}
Learning starts	100 decisions
Training frequency	every 4 decisions
Discount γ	0.95
Replay capacity	10^4 transitions
Batch size	32
Target synchronization interval	250 steps
Exploration ε	0.30 to 0.05
Call penalty λ_w	0.15
Failure penalty λ_f	0.25
Token penalty λ_τ	0.02
Token scale τ_0	10^4 tokens
Selector seeds	$0, 1, 2, 3, 4$

Table 3 counts decisions, not tasks. For each selector seed, candidate checkpoints are evaluated every 250 training decisions on a held-out selector-validation subset drawn from $\mathcal{T}_{\text{train}}$. The checkpoint with the highest mean return on that subset is used for evaluation. The validation subset remains disjoint from $\mathcal{T}_{\text{eval}}$.

C.3 Construction, Training, and Evaluation Procedure

Let $\mathcal{T}_{\text{build}}$, $\mathcal{T}_{\text{train}}$, and $\mathcal{T}_{\text{eval}}$ denote construction, RL training, and evaluation tasks. The held-out protocol requires

$$(\mathcal{T}_{\text{build}} \cup \mathcal{T}_{\text{train}}) \cap \mathcal{T}_{\text{eval}} = \varnothing. \tag{14}$$

Construction and training may share task identifiers under this condition; both must exclude all evaluation identifiers. Retrieval separately excludes a record whenever its source identifier equals the current task identifier, including during training. Memory construction collects and admits verified records; training then learns which records to use while keeping that saved store fixed. If additional source tasks are used to update the store, the result is a new snapshot even if its record count stays at the capacity limit. During held-out evaluation, neither records nor features derived from reuse history change. Evaluation logs may count uses, but must not feed those counts back into retrieval or the selector. Policy parameters are frozen as well.

Algorithm 1 Verified memory construction, RL training, and held-out reuse

Require: Fixed LLM π, generator G, evaluator H; budgets K, B, k; task sets $\mathcal{T}_{\text{build}}, \mathcal{T}_{\text{train}}, \mathcal{T}_{\text{eval}}$
Initialize store $\mathcal{M}$, Q-network Q_θ, target Q_{θ^-}, replay $\mathcal{D}$
Construct memory on $\mathcal{T}_{\text{build}}$:
for each observed failed decision (x, s, a, o) with verification budget **do**
    Generate up to K local alternatives $G(s, a, o; K)$
    for each alternative a' while verification budget remains **do**
        Evaluate $o' \leftarrow H(s, a')$ from a copy or reset of s
        if evaluation completes and $U(o') - U(o) > \epsilon_{\text{adm}}$ **then**
            Distill (q, a^-, a^+, e, c, u) with source identifier x
            Apply admission and retention rules; keep $|\mathcal{M}| \leq B$
        end if
    end for
end for
Freeze memory snapshot and reuse features
Train selection on $\mathcal{T}_{\text{train}}$:
for each task until the training-step budget is exhausted **do**
    for each decision until the task terminates or its budget is exhausted **do**
        Draft $\hat{a} \sim \pi(\cdot \mid s)$; retrieve $\mathcal{R}$ excluding this task's records
        Encode z; choose valid j by ε-greedy Q_θ
        Use draft if $j = 0$; otherwise revise it with record $\mathcal{R}[j]$
        Execute action with H; obtain reward r, next observation z', and terminal flag d
        Add (z, j, r, z', d) to $\mathcal{D}$; minimize Eq. (13) on a batch
        Synchronize $\theta^- \leftarrow \theta$ at the target-update interval
    end for
end for
Evaluate on $\mathcal{T}_{\text{eval}}$:
Freeze $\mathcal{M}$ and θ; repeat the decision loop with greedy selection and no updates

D Experimental Details

D.1 Benchmarks and Evaluators

A benchmark supplies tasks; an evaluator checks attempted solutions. The decision unit is the part of a solution that the agent changes, not necessarily a complete task. The twelve settings below provide executable checks for candidate actions. SQL execution is combined with the benchmark's result checker; successful execution alone does not establish that a query answers the question.

D.2 Held-Out SQL Configuration

The held-out SQL configuration uses 60 construction tasks and 40 evaluation tasks, sampled with seeds $12345 + 8675309$ and 12345, respectively. Task identifiers shared by the two sets are removed, and the sets are checked for overlap before memory is frozen. The configuration that also separates

Table 4: Evaluation suite and executable checker used as the world model.

Domain	Benchmark	Evaluator	Score	Decision unit
Math	miniF2F (Zheng et al., 2022)	Lean kernel	pass rate	tactic / proof step
Math	ProofNet (Azerbayev et al., 2023)	Lean kernel	pass rate	proof continuation
Math	AMBER (Yang et al., 2026)	Lean-based checker	pass rate	tactic / lemma choice
Coding	SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024)	repository tests	resolved rate	edit / patch
Coding	BugsInPy (Widyasari et al., 2020)	regression tests	repair rate	localized edit
Formal	DafnyBench (Loughridge et al., 2025)	Dafny verifier	verification	invariant / annotation
Formal	Vericoding-Dafny (Bursuc et al., 2025)	Dafny verifier	verified synthesis	code step
Formal	Vericoding-Verus (Bursuc et al., 2025)	Verus verifier	verified synthesis	proof / code step
Text-to-SQL	Spider (Yu et al., 2018)	SQL engine	exec. accuracy	join / predicate
Text-to-SQL	BIRD (Li et al., 2023)	SQL engine	exec. accuracy	schema / join
SMT	SMT-LIB / SMT-COMP (Barrett et al., 2016; SMT-COMP Organizers, 2026)	SMT solver	solved rate	encoding / solver action
SAT	SAT Competition / SATLIB (SAT Competition Organizers, 2026; Hoos & Stützle, 2000)	SAT solver	solved rate	branching / encoding

databases contains 35 construction databases and 34 evaluation databases with no overlap. These are configuration details, not additional outcome measurements in the main table.

The agent observes the question, database identifier, and optional evidence. The verifier returns reference-derived correctness feedback without revealing the reference SQL. Evaluation disables record construction, admission, eviction, and reuse-score updates; the low-utility suppression gate is also inactive. The memory-use policy is trained on a separate training stream and is fixed during evaluation (Appendix C). The construction/evaluation counts above do not specify the size of that RL training stream; 2,000 decision steps should not be read as 2,000 tasks.

D.3 Accumulation Checkpoints and Record Retention

What accumulates. As the agent attempts source tasks, it tests edits to failed actions and saves corrections that pass verification and admission. The store accumulates *accepted correction records*, not raw tasks or new LLM parameters. A source task can add no record or several. A checkpoint is a saved store after a specified number of source tasks have been processed. The plotted checkpoints are 0, 25, 50, 100, and 200 tasks. At zero source tasks, the store contains no corrections from that construction stream.

How to read the panels. Panel (a) compares held-out performance across construction checkpoints within each method and between CounterMem and ReasoningBank. Evaluation reads that snapshot without adding records or changing reuse statistics. Panel (b) reports retrieval and context tokens per task, rather than all construction and solving tokens. Panel (c), labeled regression on earlier tasks, describes loss of earlier-task performance as construction advances; it is a different measurement from the held-out score in (a). The figure marks one standard deviation across five source-task orderings. The same trained selector is reused unchanged at the 0/25/50/100/200 construction checkpoints; only the memory snapshot changes. This keeps the accumulation curve focused on additional stored experience rather than retraining a different selector at each checkpoint.

What fixed capacity controls. Panel (d) gives each retention rule the same maximum record count. Recency prioritizes recent records; random retention chooses without a utility or similarity ranking; similarity uses a similarity ranking; and utility uses the original checked improvement and outcomes of later use during construction, as in Eq. (5). Keeping the record limit equal prevents a method from gaining simply by being allowed to retain more records. It does not equate record lengths, retrieval tokens, or all computational costs.

Run interpretation. The mechanism uses $B = 200$ as the record-capacity limit. Checkpoints still count processed source tasks rather than stored records, so a 200-task checkpoint need not contain 200 memories. The five task orderings determine the mean and standard deviation shown in the figure, while the fixed selector policy described above is shared across checkpoints. Panel (c) remains a regression measurement on the designated earlier-task set rather than another held-out-score curve. The figure is therefore not evidence of unbounded growth in the number of memories.

D.4 Transfer Conditions and Reuse Measurements

Transfer asks whether a correction learned on a source task remains useful when some part of the next task changes. An environment is the setting in which actions are executed, such as a database and schema, repository, or verifier configuration. The examples in Table 5 illustrate the intended distinctions. The reported transfer study uses 60 source–target task pairs for each of the four conditions.

Table 5: Meaning of the four transfer conditions, with illustrative examples.

Condition	What changes	Example of the reuse question
Same environment, new task	A different request within the source environment.	Does the boundary correction help a new query over the same schema?
New environment, new task	The execution setting as well as the request.	Can a predicate correction apply in a different database with compatible types?
Same error, new surface	The form of the task or action, while the underlying mistake is similar.	Does the record match a renamed column or a differently worded inclusive request?
Look-alike, changed condition	Surface similarity remains, but a requirement for the correction fails.	Can the agent avoid changing $>$ to $\geq$ when the request requires a strict boundary?

Three separate questions. Task success asks whether the complete task passes its evaluator. Retrieval coverage asks whether retrieval supplies guidance. Harmful use asks whether acting on memory worsens the outcome; retrieval alone is not harmful use, and a failed task alone does not establish that memory caused its failure. A candidate can be retrieved and then skipped by the selector. The transfer figure's reported coverage should therefore not be interpreted as either a selection rate or a correctness rate.

Denominators and harm labels. Retrieval coverage is measured per *retrieval opportunity*: the numerator is the number of retrieval opportunities for which at least one eligible record is returned, and the denominator is the number of opportunities at which retrieval is attempted. Records rejected by the compatibility filter do not count as coverage.

Harmful use is measured per *selected record use*. The denominator counts decisions at which the selector chooses a memory record rather than skipping. The numerator counts those selected uses labeled harmful by the same decision-level reuse criterion used during construction (Appendix B). Thus a record that is retrieved but skipped is not a harmful use, and a task failure without a selected harmful action is not counted as one. The condition-removal percentages use the same selected-use denominator.

The retrieval intervention in Figure 4(d) uses a different normalization: its action-change and harmful-change rates are reported per retrieved memory, as stated in that figure. The rescued/broken experiment in Appendix E.3 is different again, using 120 paired tasks per domain. These three denominators should not be interchanged.

D.5 Baselines and Interventions

In each main-table group, the agent procedure and base LLM are fixed; the rows change the added mechanism. ExpeL extracts lessons from past attempts; Generative Agents combines saved experiences and reflections; Voyager retains reusable procedures; MemGPT manages information between context and external storage. These components are adapted to the common task interface. Best-of-N generates additional current-task candidates without persistent memory. ReasoningBank extracts strategies from past attempts. In the separate 6M-token comparison, MaTTS gives it more trial experience from which to learn; the verified-feedback variant gives its memory-building process checked outcomes. These controls ask whether either addition brings ReasoningBank to the full method's score. They compare complete methods and do not isolate each difference in how records are formed or used.

In the component ablation, *CF without WM* proposes memory advice without testing whether the suggested change works; ordinary task actions are still evaluated. *WM without CF* tests alternatives for the current task but saves no corrections for later tasks. In the retrieval ablation, *random* retrieval samples records without state matching; *shuffled* retrieval reassigns records to other decisions; *correct* (matched) retrieval uses the recorded situations and conditions. No retrieval means the LLM receives no memory record. These interventions alter the records available for selection; they are not a comparison between a trained selector and a fixed selection rule. The manuscript package alone does not contain complete runnable baseline configurations. The reference implementation resolves two details for the Best-of-N compute control: it uses $N = 1 + K = 5$ candidates, and an LLM

judge selects one candidate for execution. The verifier is not used to choose among those candidates, so this control increases current-task inference rather than persistent memory or verifier-guided selection. Complete baseline prompts and per-method construction budgets are still not supplied with the manuscript package.

D.6 Metric Conventions

A world-model call is one executable evaluation: a Lean check cycle, repository test invocation, Dafny/Verus attempt, SQL execution with checking, or solver invocation. A failed attempt reaches that evaluator and fails the task's success criterion. LLM-only proposals are excluded from these two counts but included in token accounting. Counterfactual checks are included.

For a benchmark with n_{solved} solved instances, normalized interaction counts are $N_{\text{WM}}/n_{\text{solved}}$ and $N_{\text{fail}}/n_{\text{solved}}$. Reported reductions subtract the augmented agent's normalized count from its base agent's count. These ratios can change through both their numerator and denominator; a reduction does not establish a lower total call count on its own. The main table reports total task-run tokens, without dividing by solved tasks. Offline selector-training tokens and training compute are not separately itemized. A fixed task-run token budget therefore does not establish equal total training cost or equal wall-clock time.

D.7 Values Underlying the Benchmark Breakdown

	(a) Score gain (pp)		**(b) Call reduction (per solved)**		**(c) Failure reduction (per solved)**	
	ReAct	Reflexion	ReAct	Reflexion	ReAct	Reflexion
miniF2F	+31.5	+15.8	+1.1	+0.4	+0.8	+0.3
ProofNet	+24.2	+12.1	+0.9	+0.3	+0.7	+0.4
AMBER	+27.4	+9.9	+2.4	+0.9	+1.6	+0.6
SWE-bench Verified	+21.4	+8.4	+3.6	+1.2	+1.1	+0.5
BugsInPy	+14.4	+3.8	+2.7	+1.4	+1.4	+0.7
DafnyBench	+11.2	+5.2	+0.6	-0.2	+0.3	+0.1
Vericoding-Dafny	+7.8	+3.4	+0.4	-0.3	+0.2	-0.1
Vericoding-Verus	+9.6	+4.8	-0.5	-0.4	-0.3	+0.2
Spider	+23.6	+13.8	+0.9	+0.3	+0.5	+0.2
BIRD	+16.8	+9.2	+1.2	+0.5	+0.6	+0.3
SMT-LIB / SMT-COMP	+11.3	+5.2	+0.8	+0.2	+0.5	+0.2
SAT Comp. / SATLIB	+8.2	+2.5	+0.7	-0.1	+0.4	-0.1

Figure 5: **Adding CF Memory to each base agent on 12 benchmarks.** The LLM is gpt-oss-120b. Each cell compares the augmented agent with its own unaugmented version. Positive entries mean higher scores in (a) or fewer evaluator calls and failed attempts per solved task in (b,c). Negative reductions mean additional interaction.

The mean over all 24 score changes is 12.5625 pp. Averaging only the four main-table domains, with equal domain weight, gives 18.5 pp for ReAct and 8.175 pp for Reflexion. This domain-weighted summary differs from the 12-benchmark mean because domains contain different numbers of benchmarks.

D.8 Prompt and Parser Interfaces

The reported runs use fixed prompts with parser-checked output contracts. The contracts below state the behavior that affects the experiments; they do not give the LLM permission to bypass the evaluator or alter more of the action than the corresponding module allows.

For `DISTIL_RECORD`, the condition instruction is:

Table 6: Values in Figure 5. Each entry is ReAct / Reflexion. Score changes are pp; call and failure reductions are per solved instance.

Benchmark	Score gain	Call reduction	Failure reduction
miniF2F	31.5 / 15.8	1.1 / 0.4	0.8 / 0.3
ProofNet	24.2 / 12.1	0.9 / 0.3	0.7 / 0.4
AMBER	27.4 / 9.9	2.4 / 0.9	1.6 / 0.6
SWE-bench Verified	21.4 / 8.4	3.6 / 1.2	1.1 / 0.5
BugsInPy	14.4 / 3.8	2.7 / 1.4	1.4 / 0.7
DafnyBench	11.2 / 5.2	0.6 / −0.2	0.3 / 0.1
Vericoding-Dafny	7.8 / 3.4	0.4 / −0.3	0.2 / −0.1
Vericoding-Verus	9.6 / 4.8	−0.5 / −0.4	−0.3 / 0.2
Spider	23.6 / 13.8	0.9 / 0.3	0.5 / 0.2
BIRD	16.8 / 9.2	1.2 / 0.5	0.6 / 0.3
SMT-LIB / SMT-COMP	11.3 / 5.2	0.8 / 0.2	0.5 / 0.2
SAT Competition / SATLIB	8.2 / 2.5	0.7 / −0.1	0.4 / −0.1

Table 7: Prompt interfaces used by the reported runs.

Template	Used in	Required behavior
`GENERATE_LOCAL ALTERNATIVES`	Module I	Return local edits in the fixed edit grammar, one changed decision unit per candidate; do not generate a complete replacement submission.
`DISTIL_RECORD`	Module II	Distill the checked contrast into a reusable record. The condition must be checkable on a future task and must be able to be false on a look-alike task.
`DRAFT_ACTION`	Module III	Produce the action draft before any memory is shown. This is the memory-free draft used by the selector and reuse-credit logic.
`REVISE_WITH_RECORD`	Module III	Show exactly one selected record and revise the draft only when useful; the model may return the draft unchanged.
`RECORD_RENDER` / `RECORD_RENDER_NO CONDITION`	Condition ablation	Render the same record fields in both cases; the only intended difference is whether the applicability clause is shown.
`BEST_OF_N_JUDGE`	Best-of-N control	Choose exactly one candidate and return `CHOICE: <n>`. Parse failures are logged and fall back to candidate 1; the verifier does not choose the candidate.
`EXPEL_INSIGHT`	ExpeL baseline	Extract one transferable lesson from the completed attempt.
`GENERATIVE_AGENTS REFLECT`	Generative Agents baseline	Write one generalized reflection for later retrieval.
`VOYAGER_SKILL`	Voyager baseline	Store a reusable procedure with a name, applicability description, and steps.
`REASONINGBANK EXTRACT`	ReasoningBank baseline	Produce a strategy with title, description, and content, without adding COUNTERMEM's condition field.
`MATTS_SELF CONTRAST`	MaTTS control	Contrast the available attempts to extract strategy-level guidance, without COUNTERMEM's pairwise executable verification gate.
`REASONINGBANK VERIFIED`	Verified-feedback control	Provide checked step outcomes to the strategy extractor while retaining the ReasoningBank record format.
`LOOK_ALIKE_TASK`	Transfer study	Generate a superficially similar target for which the stored applicability condition is false; the generated target is retained only after the applicability check confirms that the recorded condition does not hold.

```
CONDITION: <a requirement of the NEW task that must hold
before applying this.  It must be checkable and it must
be able to be FALSE for a task that looks similar.  Write
"none" only if no such requirement exists.>
```

The parser maps `none` to an empty condition, which the compatibility rule treats as universally applicable. This makes the absence of a condition explicit rather than storing the literal word as a constraint.

Prompt fields are truncated before rendering to keep token accounting bounded: state 2500 characters; action or draft 2000; evaluator feedback 800; decision units 4000; environment context 3000; rendered record 1200; and trajectory 3000.

E Additional Analyses

E.1 Backbone Summary

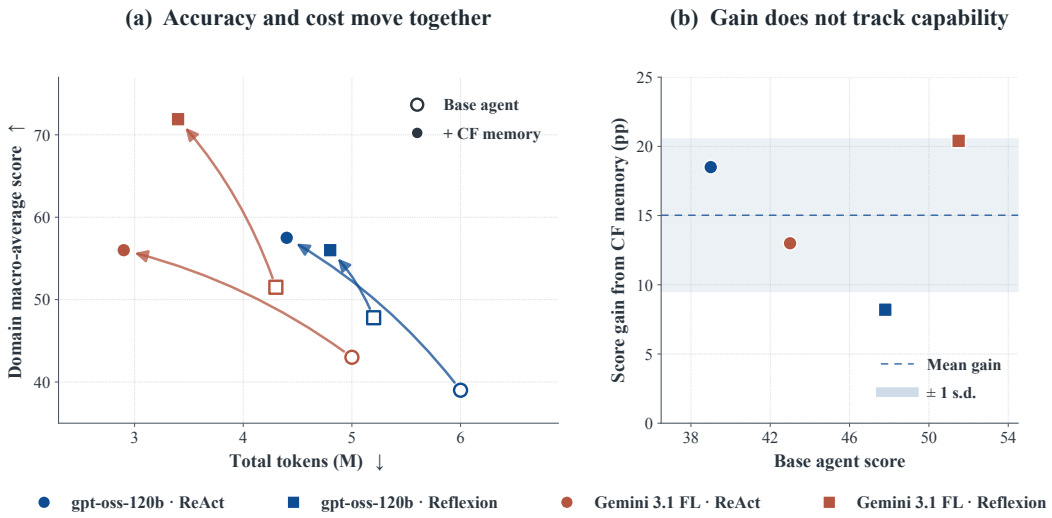


Figure 6: **Four-domain summaries of the main table.** Each arrow links one unaugmented agent to its CF Memory version. The right panel plots gain against base score; the shaded band is variation across the four agent–backbone combinations, not a confidence interval over runs.

This figure re-expresses the None and CF Memory rows of Table 1. All four arrows move toward higher score and fewer total tokens. Gains are not ordered by the unaugmented score: gpt-oss-120b ReAct gains 18.5 pp from a score of 39.0, while Gemini Reflexion gains 20.4 pp from 51.5. Four configurations are insufficient to establish how memory gains generally depend on the base model's ability.

E.2 Comparison at a Common 6M-Token Budget

To ask whether the score advantage persists when the reported token allowance is equal, this separate five-domain study compares ReasoningBank, ReasoningBank with MaTTS (Ouyang et al., 2026), our verified-feedback variant of ReasoningBank, and CounterMem. The MaTTS control asks whether more trial experience closes the gap; the verified-feedback control asks whether giving memory construction checked outcomes closes it. All methods use a common 6M-token task-run budget. The five domains are Math, Coding, Text-to-SQL, Formal Verification, and SAT. The budget matches task-run tokens, not offline training cost or wall-clock time. The package does not specify the run's agent–LLM pair or how the 6M tokens are allocated among construction and solving.

Neither control closes the gap. CounterMem averages 63.5, exceeding ReasoningBank with MaTTS by 7.8 pp and ReasoningBank with verified feedback by 10.4 pp (Figure 7).

On Formal Verification, giving ReasoningBank checked outcomes changes its score from 66.1 to 63.8; CounterMem reaches 73.5. Checked outcomes alone therefore do not explain the gap. The methods also store and use memory differently, so this comparison neither identifies the decisive difference nor establishes that verified feedback generally hurts performance.

E.3 Rescued Tasks, Broken Tasks, and Applicability Conditions

For each method and domain, the experiment pairs runs on 120 tasks with and without that method's memory. A *rescued* task fails without memory and succeeds with it. A *broken* task succeeds without memory and fails with it. Tasks whose success status is unchanged contribute to neither count. The difference, rescued minus broken, measures the net change in successful tasks within these paired runs.

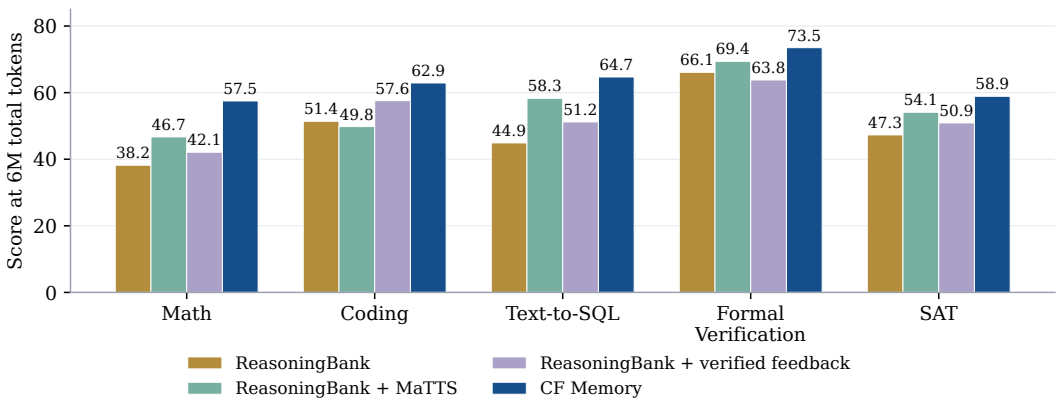


Figure 7: **Comparison at a common 6M-token budget.** CF Memory outperforms ReasoningBank, ReasoningBank with MaTTS, and ReasoningBank with verified feedback in all five domains. Reported averages give each domain equal weight.

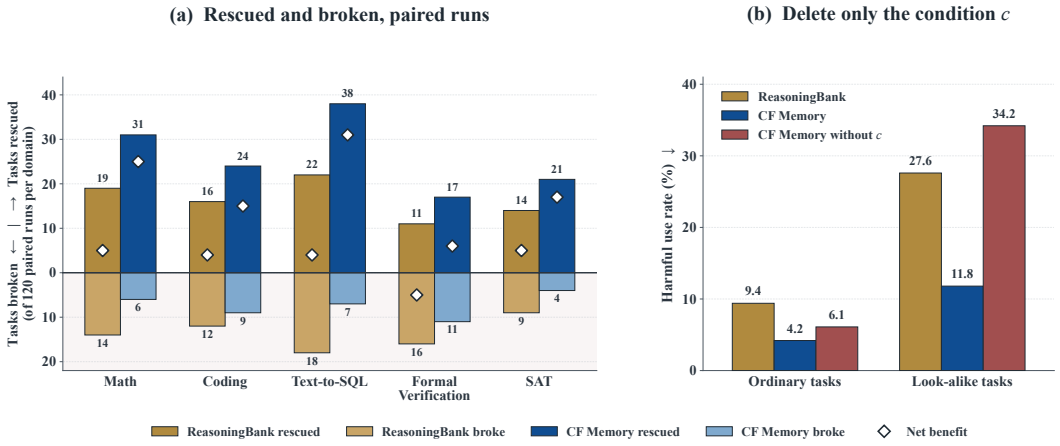


Figure 8: **Benefits and harm from reuse.** (a) In 120 paired runs per domain, rescued tasks fail without memory and succeed with it; broken tasks have the opposite outcome. Diamonds mark the net count, rescued minus broken. (b) Removing only the applicability field c increases harmful use, especially on look-alike tasks.

Comparing each method with its own no-memory runs, the net counts for ReasoningBank are $+5, +4, +4, -5, +5$ across Math, Coding, SQL, Formal Verification, and SAT. CF Memory yields $+25, +15, +31, +6, +17$. Across 600 paired runs per method, these sum to 13 and 94 net additional successes, respectively. CF Memory still breaks tasks: its Formal Verification results include 17 rescued and 11 broken cases. The aggregate gain therefore does not mean that memory helps every task.

A second intervention removes only the applicability field c from the CF mechanism. In the reported ablation, removal is applied at both sites where c is used: the retrieval filter no longer enforces the condition, and the rendered record no longer shows the applicability clause. All other record fields and selector settings are unchanged. Removing c raises harmful use from 4.2% to 6.1% on ordinary tasks and from 11.8% to 34.2% on look-alike tasks. The larger increase on look-alike tasks supports keeping c: stating and enforcing when a correction applies helps avoid repeating it under the wrong conditions. These percentages are normalized per selected record use (Appendix D.4); they are not the per-domain broken-task rates from the paired experiment.