RETHINKING SELF-EVOLVING AGENT MEMORY: CONTENT, WRITER, AND SOURCE

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents can improve from interaction experience by storing and reusing external textual memory, yet it remains unclear which memory design choices actually drive self-evolution. Existing methods often entangle memory content, writer, source experience, and maintenance, making their effects difficult to isolate. We disentangle these factors in a unified framework and compare raw trajectories, reflections, rules, and procedural skills across six interactive benchmarks. We find that no memory representation is universally best: performance depends on matching the abstraction of stored experience to the reusable structure of the environment, while raw trajectories provide a stable, writer-free baseline. Stronger memory writers do not consistently produce better memory; effective writers instead construct compact, transferable abstractions. Memory source has a more consistent effect: success-derived memory outperforms failure-derived memory in every tested setting, by up to 38 percentage points, whereas greater task diversity does not consistently outperform repeated experience. We further find that most gains emerge within 50–100 evolving episodes, and that additional memory-management operations can substantially compress the store without reliably improving downstream performance. These findings suggest that effective self-evolving memory depends more on reliable experience and task-aligned representations than on greater scale or complexity.

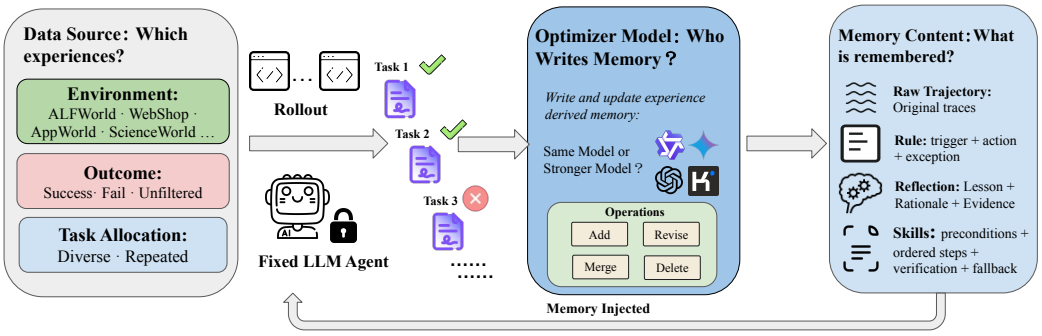

Figure 1: **Overview of our unified framework for studying experience-derived memory in self-evolving agents.** A fixed language agent generates rollout traces from controlled experience sources that vary in environment, outcome, and task allocation. These traces are either stored directly as raw trajectories or transformed by an optimizer model into reflections, rules, or procedural skills through memory operations such as addition, revision, merging, and deletion. The resulting memory is injected into subsequent rollouts, allowing us to isolate three design choices: which experiences to learn from, who writes the memory, and what information is remembered.

1 INTRODUCTION

LLM agents can improve from interaction experience without parameter updates by storing and reusing external memory, an approach explored across reasoning, web interaction, software engineering, tool use, and embodied environments (Gao et al., 2025; Shinn et al., 2023; Zhao et al.,

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.

2024; Ouyang et al., 2026). Yet how such memory should be designed remains poorly understood. Existing systems vary not only in what they store, from raw trajectories to reflections, rules, and procedural skills (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024; Ouyang et al., 2026; Zhang et al., 2026), but also in who writes the memory, which experiences are used to construct it, and how accumulated knowledge is maintained. Because these factors are typically changed together, observed gains conflate multiple design choices, making it difficult to determine what actually drives self-evolution.

Recent benchmarks have begun to compare agent memory systems more systematically (Wang et al., 2026; Lei et al., 2026), but they still largely evaluate complete methods end to end. As a result, several basic questions remain unanswered. At what level of abstraction should an agent remember its experience: as raw trajectories, reflections, rules, or procedural skills? Does a stronger memory optimizer produce more useful memory for the same agent? Which experiences—successful or failed, easy or difficult, diverse or repeatedly revisited—are most valuable to learn from? In the absence of controlled comparisons, these decisions are often determined by method-specific heuristics and intuitions, making it difficult to derive general principles for building self-evolving agents.

In this work, we aim to demystify the design of experience-derived memory for self-evolving language agents. Rather than introducing another specialized memory architecture, we decompose the memory pipeline into independent design dimensions and systematically study them under a unified agent framework. Specifically, we separate three central choices: what information is remembered, who writes and updates the memory, and which interaction experiences are used as its source. We instantiate four representative memory forms—raw trajectories, reflections, rules, and procedural skills—and evaluate them across six interactive benchmarks while controlling the acting agent and evaluation protocol. We further vary the memory writer, the amount and composition of source experience, and the memory-management policy, allowing us to isolate the effect of each design choice. Across these controlled comparisons, we find that effective self-evolving memory depends less on simply increasing experience, optimizer capability, or system complexity than on selecting reliable experience and representing it at an abstraction level appropriate to the task. Our experiments lead to the following key findings:

1. Most performance gains emerge within the first 50 to 100 episodes, after which additional experience often yields diminishing returns or even degrades performance due to redundant or interfering memories.
2. Success-derived memory outperforms failure-derived memory in all six tested benchmark–representation settings, with gaps of up to 38 percentage points, while failure-only memory can perform worse than using no memory at all.
3. There is no universally best memory representation. Performance depends on matching the abstraction of memory to the reusable structure of the environment.
4. Raw trajectories are a strong default: they require no LLM writer, avoid semantic drift, and never meaningfully underperform the no-memory baseline across the six benchmarks.
5. Stronger models are not necessarily better memory optimizers. The best optimizer varies across settings, while effective optimizers produce compact, transferable abstractions.

2 Related Work

Benchmarks for Evaluating LLM Agents. A diverse set of interactive benchmarks has emerged for evaluating language agents, spanning embodied and scientific environments such as ALFWorld and ScienceWorld (Shridhar et al., 2020; Wang et al., 2022), web interaction such as WebShop, Mind2Web, WebArena, and VisualWebArena (Yao et al., 2022; Deng et al., 2023; Zhou et al., 2024; Koh et al., 2024), general multi-environment evaluation such as AgentBench (Liu et al., 2024), tool and application interaction such as ToolBench, AppWorld, and τ-bench/τ^2-bench (Xu et al., 2023; Trivedi et al., 2024; Yao et al., 2024; Barres et al., 2025), and spreadsheet manipulation such as SpreadsheetBench (Ma et al., 2024), and software engineering environments such as SWE-bench (Jimenez et al., 2024). These benchmarks progressively increase environment fidelity, interaction horizon, statefulness, and tool complexity, providing complementary settings for studying experience-driven agent improvement. These benchmarks have also become common testbeds for studying agents that learn from interaction experience, since they provide repeated tasks, explicit

environment feedback, and objective success signals from which experience can be accumulated. However, benchmark performance alone does not reveal which properties of the accumulated experience make self-evolution effective.

Experience-derived memory for self-evolving agents. A growing line of work enables LLM agents to improve from interaction experience without modifying the parameters of the acting model. Existing approaches span a spectrum of abstraction in how past experience is represented. At the most concrete level, trajectory-based methods retain previous interaction traces and reuse them as demonstrations. Synapse stores abstracted state–action trajectories and retrieves similar trajectories as in-context exemplars, while ExpeL combines retrieved experiences with natural-language insights extracted across prior trials (Zheng et al., 2024; Zhao et al., 2024). Reflection-based approaches instead compress individual executions into textual lessons. Reflexion derives verbal feedback from previous attempts and stores it in episodic memory, while ReasoningBank distills reusable reasoning strategies from both successful and failed trajectories; related approaches extract failure causes, recovery strategies, and optimization guidance from execution traces (Shinn et al., 2023; Ouyang et al., 2026; Fang et al., 2026a). More structured systems represent experience as reusable rules or principles. FORGE extracts textual heuristics from failed trajectories, TACO continually discovers and refines task-adaptive compression rules (Bogdanov et al., 2026; Ren et al., 2026). Finally, procedural-memory approaches distill experience into reusable multi-step routines or skills. Agent Workflow Memory induces recurring workflows from trajectories; Memp constructs both fine-grained instructions and higher-level procedural abstractions; ReMe continually distills and refines procedural memories; and CODESKILL learns to construct and maintain a bank of procedural coding skills (Wang et al., 2024; Fang et al., 2026b; Cao et al., 2026; Li et al., 2026). These methods demonstrate the value of experience-derived memory, but they differ simultaneously in memory representation, construction procedure, updating, and agent configuration. Consequently, existing results do not directly establish whether trajectories, reflections, rules, or procedural skills are preferable when the remaining components of the agent are held fixed.

3 Methods Overview

3.1 Problem Formulation

We study experience-derived textual memory through a unified self-evolving agent framework. The framework separates the *content* stored in memory from the *mechanisms* used to construct and maintain it, allowing us to attribute performance differences to individual design choices. For an evolving task t_i, the agent first retrieves a memory block R_i from the current memory store M_i, executes the task conditioned on R_i, and produces an episode E_i containing its interaction trajectory and task outcome. A memory writer then transforms E_i into a set of memory operations:

$$R_i = \mathrm{Retrieve}(M_i, t_i), \qquad E_i = \mathrm{Agent}(t_i, R_i), \qquad M_{i+1} = \mathrm{Update}(M_i, E_i, R_i).$$

After evolving on a designated set of tasks, the resulting memory is frozen and evaluated on held-out tasks. No memory writes are permitted during evaluation.

3.2 Memory content types

We instantiate four memory representations while holding the remaining agent and memory pipeline fixed. Each of the four representations is a canonical instantiation of its family of methods, and dropped mechanisms specific to a single system. **Raw trajectory** memory stores the task instruction and the complete action–observation trace of a rollout without using an LLM writer, following the general paradigm of trajectory-based memory and exemplar reuse explored by prior methods such as Synapse and ExpeL (Zheng et al., 2024; Zhao et al., 2024). **Reflection** memory represents an episode as a situation, an imperative and transferable lesson, and a causal rationale explaining why the lesson applies, similar in spirit to verbal and reasoning memories in Reflexion and ReasoningBank Shinn et al. (2023); Ouyang et al. (2026). Depending on the available evidence, it identifies a non-trivial decision in a successful rollout, the earliest consequential error in a failed rollout, or the first divergence between successful and failed attempts. **Rule** memory distills experience into atomic condition–action rules, each linking a locally observable condition to an action the

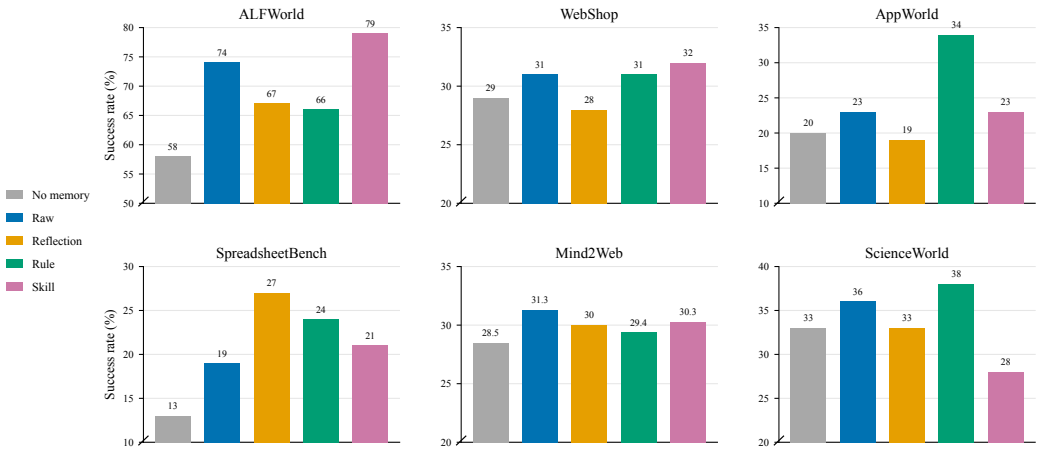

Figure 2: Comparison of four memory content types after 100 evolving episodes. We report held-out success rates on six benchmarks for the agent without memory and with Raw trajectory, Reflection, Rule, or Skill memory.

agent should take or avoid, with optional exceptions, capturing the structured heuristics and principles used in methods such as FORGE and TACO (Bogdanov et al., 2026; Ren et al., 2026). **Skill** memory captures a reusable multi-step procedure, including its invocation condition, preconditions, parameterized steps, verification checks, and recovery actions, corresponding to procedural-memory approaches such as Agent Workflow Memory, Mem^p, ReMe, and CODESKILL Wang et al. (2024); Fang et al. (2026b); Cao et al. (2026); Li et al. (2026). The detailed schema for each of the memory representations is in Appendix B.

3.3 Selection of Evaluation Benchmark

We evaluate the framework on six agent benchmarks spanning complementary environments and interaction modalities: ALFWorld Shridhar et al. (2020), WebShop (Yao et al., 2022), AppWorld (Trivedi et al., 2024), Mind2Web (Deng et al., 2023), ScienceWorld (Wang et al., 2022), and SpreadsheetBench (Ma et al., 2024). ALFWorld evaluates long-horizon embodied household tasks through text-based interaction; WebShop evaluates multi-step product search and purchasing in a simulated e-commerce environment; AppWorld evaluates stateful API-based workflows spanning multiple applications; Mind2Web evaluates language-guided web navigation and action grounding across diverse real-world websites; ScienceWorld evaluates multi-step scientific reasoning and experimentation in an interactive text-based environment; and SpreadsheetBench evaluates spreadsheet manipulation through program generation and tool execution. The detail of each benchmark is in Appendix A.

In our main experiments, we use Qwen3.5-9B for both agent action generation and memory writing unless otherwise specified. The maximum interaction horizons are 50 steps for ALFWorld, 15 steps for WebShop, 100 steps for ScienceWorld, and 30 interactions for AppWorld and SpreadsheetBench. Mind2Web is evaluated at the annotated action-step level.

4 Memory Content and Representations

We examine how memory representation, the amount of experience, and memory-management operations shape held-out agent performance.

4.1 Comparison of Different Memory Contents

Figure 2 compares four memory content types after 100 evolving episodes on each benchmark. The results show that no single memory representation consistently dominates across all benchmarks. Instead, performance depends strongly on whether the granularity of the stored content matches the

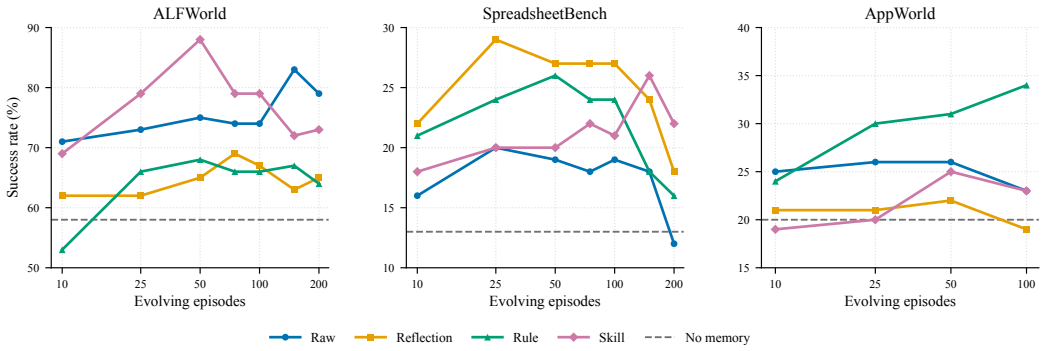

Figure 3: Scaling behavior of self-evolving agent memory. Held-out performance of Raw, Reflection, Rule, and Skill memory as the number of evolving episodes increases.

reusable structure of the target environment. Each configuration is independently run three times, and we report the mean success rate.

Raw trajectory memory provides a faithful and relatively stable default. Raw memory stores interaction trajectories without using an LLM writer to abstract or rewrite them. It therefore avoids semantic drift and preserves the exact action sequences that were valid in the environment. Raw memory improves ALFWorld from 58% to 74%, while producing smaller positive gains on WebShop, AppWorld, SpreadsheetBench, Mind2Web and ScienceWorld. This relatively consistent behavior suggests that retaining grounded experience is often safer than introducing potentially incorrect abstractions.

Reflection memory is useful when tasks reveal transferable causal lessons. Reflection memory converts past outcomes into natural-language lessons describing what went wrong, what should be done differently, and why. It performs particularly well on SpreadsheetBench, improving success from 13% to 27%, and also improves ALFWorld from 58% to 67%. These environments contain failures that can often be explained through reusable lessons, such as preserving spreadsheet structure or satisfying an action precondition before executing the next subgoal. In contrast, reflection underperforms the raw trajectory baseline on WebShop, AppWorld, Mind2Web, and ScienceWorld.

Rule memory is effective when decisions depend on local and identifiable conditions. Rule memory represents experience as compact condition–action pairs. Its strongest result is on AppWorld, where it improves task success from 20% to 34%. AppWorld requires an agent to interact with stateful APIs, and many failures arise from local preconditions, which align naturally with short, atomic rules that can be evaluated at the current step.

Skill memory is effective when tasks can be compressed into reusable multi-step procedures. Skill memory achieves the strongest result on ALFWorld, improving success from 58% to 79%. ALFWorld contains a small number of recurring task families, such as cleaning, cooling, heating, examining, and placing objects. A small number of skills can therefore encode workflows such as locating an object, acquiring it, performing the required transformation, and placing it in the target receptacle. Skill memory is less effective on benchmarks whose tasks do not share such stable workflows, remaining at the baseline on AppWorld and falling below it on ScienceWorld.

Overall, memory content should match the structure of the task. The comparison indicates that memory quality is determined not only by how much experience is stored, but also by how that experience is represented. Raw trajectories provide a grounded and comparatively safe default; reflections help when failures support reliable causal diagnosis; rules are suitable for local decision constraints; and skills are most effective for recurring multi-step workflows.

Table 1: Effect of additional memory management operations after 100 evolving episodes. Performance change is measured in percentage points as $\Delta\mathrm{SR} = \mathrm{SR}_{\text{full}} - \mathrm{SR}_{\text{minimal}}$. Store reduction is computed as $1 - |\mathcal{M}_{\text{full}}|/|\mathcal{M}_{\text{minimal}}|$. Positive values therefore indicate either improved performance or a smaller memory store under the full policy.

(a) Performance change (percentage points)

Benchmark	Raw	Refl.	Rule	Skill
ALFWorld	+3.0	−5.0	+3.0	−3.0
WebShop	−5.0	+3.0	+2.0	+1.0
AppWorld	+5.0	+3.0	−5.0	0.0
SpreadsheetBench	+2.0	+1.0	+3.0	+5.0

(b) Memory-store reduction (%)

Benchmark	Raw	Refl.	Rule	Skill
ALFWorld	18.7	−16.7	40.0	−33.3
WebShop	35.7	36.8	94.2	80.0
AppWorld	−3.8	40.0	61.5	33.3
SpreadsheetBench	38.1	59.0	63.9	57.1

4.2 Self-Evolving Agent Memory Scaling Behavior

We investigate how the performance of a self-evolving agent changes as the number of episodes used for memory evolution increases. As shown in Figure 3, we vary the evolution budget from 10 to 200 episodes and compare four memory representations: Raw, Reflection, Rule, and Skill. Each data point in the figure is obtained by independently running the complete evolution-and-evaluation procedure three times and reporting the average success rate. The dashed line denotes the performance of the agent without memory.

Overall, increasing the number of evolving episodes provides clear benefits at small and moderate scales, but the improvement is neither monotonic nor universal across memory types and environments. On ALFWorld, Skill memory improves rapidly and reaches its highest performance at 50 episodes, while Raw memory continues to improve more gradually and peaks at 150 episodes. On SpreadsheetBench, most memory types initially benefit from additional experience, but their performance saturates or declines when the evolution budget becomes large; Skill memory is comparatively robust and reaches its best result at 150 episodes. AppWorld exhibits a different pattern: Rule memory improves consistently from 24% to 34% as the budget grows from 10 to 100 episodes, whereas the other representations show only limited gains or eventually regress.

These results indicate that the performance gains from scaling self-evolving memory emerge relatively early, with most memory representations reaching their peak or beginning to saturate after approximately 50–100 evolving episodes. Beyond this range, additional experience generally provides diminishing returns and may even reduce performance due to redundant or interfering memories. This suggests that effectively selecting and consolidating experience may be more important than continuously increasing the memory evolution budget.

4.3 Effect of Extra Memory Management Method

To examine whether the memory-management modules proposed in prior work on self-evolving agents provide consistent benefits, we implement several representative components within our unified memory system. These include outcome-based verification and refinement inspired by Reflexion and ExpeL (Shinn et al., 2023; Zhao et al., 2024), utility-based pruning similar to ReMe (Cao et al., 2026), and periodic cross-task induction related to Agent Workflow Memory (Wang et al., 2024). We compare these components across four benchmarks and four memory representations while keeping the retrieval procedure, context budget, and evolution data fixed. This controlled comparison allows us to determine whether these additional components are generally useful or effective only for particular benchmarks or memory representations. Each configuration is independently run three times, and we report the average performance.

Specifically, the minimal policy performs only online extraction, evidence grounding, and duplicate merging, whereas the full policy additionally verifies retrieved memories, refines or removes entries with low confidence or utility, and performs cross-task induction every 25 evolving episodes. Since Raw memory has no LLM writer, its full policy introduces only utility-based pruning.

Table 1 shows that the full policy reduces the memory-store size in 13 of the 16 evaluated configurations, with particularly large reductions for Rule and Skill memories on WebShop and SpreadsheetBench. However, this compression does not consistently translate into higher task success:

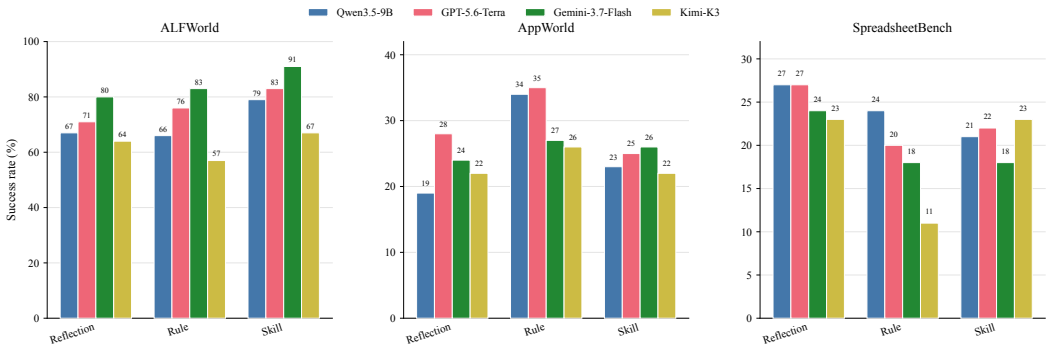

Figure 4: Effect of optimizer model choice on the utility of learned memory after 100 evolving episodes. Bars report held-out success rates on ALFWorld, AppWorld, and SpreadsheetBench.

the full policy improves performance in 11 configurations, degrades it in four, and leaves one unchanged. All four memory representations improve on SpreadsheetBench, whereas ALFWorld and AppWorld exhibit mixed effects across representations. These results suggest that the additional operations primarily act as a memory regularizer rather than a universally effective performance enhancer. With sufficient and reliable feedback, verification and pruning can remove redundant or misleading entries; under sparse or noisy feedback, they may instead penalize useful memories or produce overly general conclusions. Therefore, the benefit of full memory management depends on both the task environment and the memory representation, while the computationally cheaper minimal policy remains a competitive alternative.

5 Factor Analysis

Our next question is whether memory gains depend on who writes the memory and which episodes supply its evidence. We first examine whether the observed benefits persist across models, and then analyze how task diversity, repetition, and episode outcomes influence memory quality.

5.1 Effect of Optimizer Model Choice

To isolate the effect of the model used to optimize memory, we fix everything else and vary only the optimizer that transforms execution traces into Reflection, Rule, or Skill memory. Raw trajectory memory is excluded because it does not involve an LLM-based optimizer. Figure 4 shows that optimizer choice substantially affects downstream performance, but no optimizer consistently dominates across benchmarks or memory representations. On ALFWorld, Gemini-3.7-Flash performs best for all three representations, improving Reflection, Rule, and Skill from 67%, 66%, and 79% with Qwen3.5-9B to 80%, 83%, and 91%, respectively. AppWorld instead favors GPT-5.6-Terra for Reflection and Rule, while the four optimizers produce similar results for Skill memory. The trend reverses on SpreadsheetBench: Qwen3.5-9B matches or outperforms the stronger optimizers for Reflection and Rule, with Rule performance decreasing from 24% under Qwen3.5-9B to 20%, 18%, and 11% under GPT-5.6-Terra, Gemini-3.7-Flash, and Kimi-K3, respectively. These results show that memory quality does not scale monotonically with the general capability of the optimizer model. We also notice that optimizer capability is a second-order factor relative to representation–task fit. Across all three benchmarks, the spread induced by changing the optimizer is smaller than the spread induced by changing the memory representation, and no single optimizer dominates: the best writer differs by benchmark and, within a benchmark, by representation.

Table 2: Memory-store utilization on ALFWorld.

Writer	Live entries	Retrievals per entry	Never retrieved
Gemini-3.7-Flash	7	39.9	0%
GPT-5.6-Terra	21	20.4	0%
Kimi-K3	37	11.9	11%
Qwen3.5-9B	39	11.1	8%

Table 3: Effect of the outcome distribution of memory-source episodes under the minimal write policy. All outcomes passes the first 100 evolving episodes to the memory system; Success only and Failure only run until 100 episodes of the selected outcome have been passed to it. Entries are held-out success rates (SR, %). Parenthetical values show Failure only minus Success only, in percentage points.

Benchmark	Memory	All outcomes SR	Success-only SR	Failure-only SR
ALFWorld	Reflection	67.0	73.0	47.0 (−26.0)
	Rule	66.0	84.0	46.0 (−38.0)
	Skill	79.0	78.0	58.0 (−20.0)
SpreadsheetBench	Reflection	27.0	30.0	22.0 (−8.0)
	Rule	24.0	25.0	21.0 (−4.0)
	Skill	21.0	27.0	14.0 (−13.0)

Table 4: Effect of source diversity at a fixed budget of 150 evolving episodes under the minimal write policy. We compare 150 distinct tasks observed once, 75 distinct tasks observed twice, and 50 distinct tasks observed three times. Entries are success rates (%) on the same 100 held-out tasks within each benchmark.

Benchmark	Memory content	$\mathbf{150\times1}$	$\mathbf{75\times2}$	$\mathbf{50\times3}$
ALFWorld	Raw trajectory	83.0	74.0	77.0
	Reflection	63.0	56.0	56.0
	Rule	67.0	63.0	56.0
	Skill	72.0	68.0	73.0
SpreadsheetBench	Raw trajectory	18.0	19.0	17.0
	Reflection	24.0	21.0	25.0
	Rule	18.0	24.0	27.0
	Skill	26.0	28.0	22.0

We further analyze memory writers on ALFWorld. On ALFWorld, the strongest writer also constructs the most compact store: Gemini-3.7-Flash reaches 80% success with 7 live Reflection entries, approximately 650 tokens, whereas Kimi-K3 and Qwen3.5-9B reach 64% and 67% with 37 and 39 entries, approximately 2100–2500 tokens. As shown in Table 2, compactness is accompanied by broader reuse per entry: Gemini's entries are retrieved 39.9 times each on average and none is unused, whereas 8–11% of the Qwen and Kimi entries are never retrieved. Inspection of the entries suggests that the more important distinction is abstraction rather than length. Specifically, Gemini appears to produce more transferable abstractions, capturing recurring structure across task instances while avoiding episode-specific details and redundant entries. By contrast, Qwen more often records episode-specific observations and repeatedly propose already-stored knowledge.

5.2 Effect of Experience Source

We study two properties of the experience used to construct memory: task diversity and outcome composition. For task diversity, we hold the evolving budget fixed at 150 episodes and allocate it as 150 distinct tasks observed once, 75 tasks observed twice, or 50 tasks observed three times. For outcome composition, we compare the first 100 unfiltered episodes with streams that admit 100 successful or 100 failed episodes to the memory system. We use the minimal write policy throughout, thereby isolating the source study from the additional verification, refinement, deletion, and induction mechanisms in the full policy. Every condition is evaluated on the same 100 held-out tasks within each benchmark.

5.2.1 Successful versus Failed Source Episodes

Table 3 compares memories constructed from an unfiltered stream, successful episodes only, and failed episodes only. Across both benchmarks and all three memory types, success-derived memory consistently outperforms failure-derived memory. The difference is particularly large on ALFWorld:

Reflection improves from 47.0% to 73.0%, Rule from 46.0% to 84.0%, and Skill from 58.0% to 78.0%. Rule memory is therefore the most sensitive to source outcome on this benchmark. Importantly, failure-only memory is worse than using no memory on ALFWorld. The no-memory agent achieves 58.0%, whereas failure-derived Reflection and Rule memories obtain only 47.0% and 46.0%, respectively.

The same tendency holds on SpreadsheetBench, although the differences are smaller. Success-only sources improve Reflection from 22.0% to 30.0% (+8.0 points), Rule from 21.0% to 25.0% (+4.0 points), and Skill from 14.0% to 27.0% (+13.0 points). Unlike ALFWorld, where Reflection and Rule exhibit the largest gaps, SpreadsheetBench shows the strongest source sensitivity for procedural Skill memory. Skill memory represents a special case under the failure-only condition. Its performance remains close to the no-memory baseline on both ALFWorld and SpreadsheetBench. This is explained by the construction of procedural memory: a failed trajectory may reveal what went wrong, but it does not provide a complete, executable procedure that the Skill writer can abstract and reuse.

Comparing success-only memory with the unfiltered condition provides a more nuanced picture. Success-only filtering substantially improves Rule memory on ALFWorld and Skill memory on SpreadsheetBench, but provides only small gains—or almost no change—for the remaining configurations. In contrast, failure-only memory consistently underperforms the unfiltered condition. These results suggest that the main risk is constructing memory exclusively from unsuccessful trajectories, while the benefit of filtering for successful experience depends on both the benchmark and the memory representation.

5.2.2 Diversity versus Repetition

Table 4 shows that the preferred allocation of a fixed evolving budget is benchmark- and representation-dependent. On ALFWorld, greater diversity generally performs better, with 150×1 achieving the strongest results for Raw trajectory and Rule memory. In contrast, SpreadsheetBench often benefits from repeated exposure: 75×2 performs best for Raw trajectory and Skill memory, while 50×3 performs best for Reflection and Rule.

This difference can be understood through the successful experience generated by repeated attempts. When 50 tasks are revisited for three rounds, the cumulative number of successful tasks increases from 12.63 after the first round to 18.50 after the third on SpreadsheetBench, a 46.5% increase, compared with 33.25 to 44.50 (33.8%) on ALFWorld. Similarly, when 75 tasks are revisited twice, SpreadsheetBench increases from 16.13 to 22.63 successful tasks (40.3%), whereas ALFWorld increases from 50.38 to 62.63 (24.3%). Repetition therefore contributes relatively more new successful experience on SpreadsheetBench. In light of the preceding result that success-derived episodes produce substantially more useful memory than failed episodes, this provides a explanation for why repeated exposure is more beneficial on SpreadsheetBench. The value of repetition therefore depend on whether additional attempts convert previously unsuccessful tasks into successful experience.

6 Conclusion

We presented a controlled study of experience-derived textual memory for self-evolving LLM agents, separating memory content, writer, source experience, and management. Across six interactive benchmarks, no representation consistently dominates; performance depends on matching memory abstraction to task structure, with raw trajectories as a stable default and reflections, rules, and skills benefiting different environments. Most gains emerge within the first 50–100 evolving episodes, after which additional experience yields diminishing returns or interference. More elaborate memory-management operations can substantially compress the store, but do not reliably improve performance. Stronger optimizer models do not monotonically produce better memory; effective writers instead construct compact, transferable abstractions. Diverse experience also does not consistently outperform repeated exposure, with the preferred allocation depending on the benchmark and representation. Most consistently, success-derived memory outperforms failure-derived memory by up to 38 percentage points, while failure-only memory can underperform no memory. Overall, self-evolving memory benefits more from reliable experience and task-aligned representations than from simply accumulating more experience.

AI USE STATEMENT

Generative AI tools were used solely for limited assistance with language polishing, formatting, routine coding, and debugging. The research, analysis, interpretation of results, and preparation of the final content were carried out by the human authors. All results and citations were independently verified by the authors, who take full responsibility for the work.

REFERENCES

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Igor Bogdanov, Chung-Horng Lung, Thomas Kunz, Jie Gao, Adrian Taylor, and Marzia Zaman. Forge: Self-evolving agent memory with no weight updates via population broadcast. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 292–310, 2026.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16803–16822, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Gaodan Fang, Vatche Isahagian, KR Jayaram, Ritesh Kumar, Vinod Muthusamy, Punleuk Oum, and Gegi Thomas. Trajectory-informed memory generation for self-improving agent systems. *arXiv preprint arXiv:2603.10600*, 2026a.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, 2026b.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

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*, volume 2024, pp. 54107–54157, 2024.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Yingtie Lei, Zhongwei Wan, Jiankun Zhang, Samiul Alam, Zixuan Zhong, Peizhou Huang, Xin Wang, Jingxuan Zhang, Donghao Zhou, Yunta Hsieh, et al. Skillevolbench: Benchmarking the evolution from episodic experience to procedural skills. *arXiv preprint arXiv:2605.24117*, 2026.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. Codeskill: Learning self-evolving skills for coding agents. *arXiv preprint arXiv:2605.25430*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *Advances in Neural Information Processing Systems*, 37:94871–94908, 2024.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, et al. A self-evolving framework for efficient terminal agents via observational context compression. *arXiv preprint arXiv:2604.19572*, 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.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

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.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.

Yuyao Wang, Zhongjian Zhang, Mo Chi, Kaichi Yu, Yuhan Li, Miao Peng, Bing Tong, Chen Zhang, Yan Zhou, and Jia Li. Evomembench: Benchmarking agent memory from a self-evolving perspective. *arXiv preprint arXiv:2605.18421*, 2026.

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

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*, 2023.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL `https://arxiv.org/abs/2406.12045`.

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. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026.

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.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, volume 2024, pp. 19036–19066, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A BENCHMARK DETAILS

We evaluate self-evolving agent memory on six interactive benchmarks that cover embodied interaction, web navigation, API-based application use, scientific reasoning, and spreadsheet manipulation. These environments differ substantially in interaction horizon, state representation, action space, and the type of reusable structure available across tasks, providing complementary settings for studying how different forms of experience-derived memory transfer to unseen tasks. Below, we briefly describe each benchmark and the evaluation protocol used in our experiments.

A.1 ALFWORLD

ALFWorld is a text-based embodied environment in which an agent performs household tasks by interacting with objects, receptacles, and appliances. Typical tasks require the agent to locate an object, manipulate it according to a target state, and place it at a specified destination. Examples include cleaning, heating, cooling, examining, and relocating objects. Although the surface realization of a task varies with object identities and room layouts, many tasks share recurring procedural structures, making ALFWorld particularly useful for studying the transfer of trajectory- and procedure-level memory.

The agent interacts with the environment through natural-language observations and textual actions. We allow at most 50 interaction steps per episode. At each evaluation checkpoint, memory is frozen and evaluated on the same 100 held-out tasks. We report *strict task success*, where a task receives a score of one only when its complete goal is satisfied.

A.2 WEBSHOP

WebShop evaluates agents on multi-step product search and purchasing in a simulated e-commerce website. Given a natural-language shopping request, the agent must navigate product categories and search results, inspect product attributes, select an appropriate item and configuration, and complete the corresponding purchase action. Successful behavior therefore requires both interpreting user constraints and grounding them in information distributed across multiple web pages.

We use a maximum horizon of 15 interaction steps. Evaluation is conducted on a fixed set of 100 held-out tasks with memory frozen throughout evaluation. We report both *strict success rate* and the benchmark's *graded reward*. Strict success requires satisfying the complete shopping objective, whereas the graded reward provides partial credit according to how well the selected product matches the requested attributes.

A.3 APPWORLD

AppWorld evaluates agents on stateful workflows that require interacting with multiple applications through APIs. A task may require the agent to retrieve information from one application, use the returned entities or identifiers in subsequent API calls, and modify application state to satisfy the user request. Because API calls frequently depend on locally observable preconditions, such as authentication state, entity identifiers, or the output of an earlier call, AppWorld provides a useful setting for evaluating whether memory can capture reusable decision constraints across tasks.

We allow at most 30 interactions per task and evaluate on 100 held-out tasks. The memory store is frozen during evaluation. We report *Task Goal Completion (TGC)* together with the mean fraction of unit tests passed. TGC measures whether the overall task objective is completed, while the unit-test metric provides a finer-grained measure of how much of the required application state has been correctly established.

A.4 MIND2WEB

Mind2Web evaluates language-guided web navigation and action grounding on diverse real-world websites. Given a user instruction and the current webpage state, the agent must determine the appropriate interaction, including the target element and the operation to execute. Compared with environments whose task structure is dominated by recurring workflows, Mind2Web places greater emphasis on grounding decisions in the detailed content of the current observation.

Following our experimental setup, Mind2Web is evaluated at the annotated action-step level rather than by executing complete online website trajectories. Our held-out evaluation set contains 100 annotations comprising 838 action steps. We report *step success rate*, *element accuracy*, *operation F1*, and *annotation-level task success*. An annotation is counted as fully successful only when every constituent action step is predicted correctly.

A.5 SCIENCEWORLD

ScienceWorld is an interactive text-based environment designed to evaluate multi-step scientific reasoning and experimentation. Tasks require the agent to reason about scientific concepts while manipulating objects in a simulated environment, for example by collecting materials, operating instruments, changing physical conditions, and observing the resulting state changes. Completing a task therefore often requires coordinating a sequence of actions whose effects accumulate over a relatively long interaction horizon.

We allow up to 100 interaction steps per episode and evaluate on a fixed set of 100 held-out tasks. Memory is frozen before evaluation. We report both *strict task completion* and the benchmark's *mean task score*. Strict completion indicates whether the full objective is achieved, while the task score provides partial credit for satisfying intermediate components of the goal.

A.6 SPREADSHEETBENCH

SpreadsheetBench evaluates agents on real-world spreadsheet manipulation tasks. Given a natural-language request and a spreadsheet state, the agent must produce and execute a solution that modifies the workbook to satisfy the requested transformation. Tasks may require reasoning about spreadsheet structure, formulas, ranges, formatting, or other dependencies in the workbook, making the benchmark useful for studying whether agents can reuse experience across different spreadsheet instances.

We allow at most 30 interactions per task and evaluate on the same 100 held-out tasks for all compared memory methods. We report *strict task success* and the mean fraction of test cases passed. Each task is evaluated using three test cases and is considered fully successful only when the generated solution passes all three. The fraction of test cases passed provides a complementary fine-grained measure for partially correct solutions.

A.7 SHARED EVALUATION PROTOCOL

Across all benchmarks, evolving and evaluation tasks are separated. At each checkpoint, we freeze the memory store before evaluating the agent, and no memory writes are permitted while evaluation tasks are being executed. Within each benchmark, every compared method is evaluated on the same held-out tasks or annotations and uses the same seeded ordering. This protocol ensures that performance differences reflect changes in the learned memory rather than differences in evaluation data or online adaptation during testing.

B MEMORY CONTENT SCHEMAS

This appendix gives the complete specification of the four memory content types compared in Section 3: the shared item envelope, the per-type content schema exactly as presented to the memory writer, the validity conditions enforced at write time, the text that is embedded for retrieval, the rendering that is injected into the agent context, and a representative entry.

B.1 SHARED ITEM ENVELOPE

All four types are stored in the same record, so retrieval, deduplication, budget packing, eviction, and logging are implemented once and are identical across conditions. Only the `content` payload varies by type.

```
MemoryItem:
  type            str        # reflection | rule | skill | raw
```

```
content          dict       # type-specific payload; see A.2 - A.5
id               str        # 12-hex-digit identifier
retrieval_key    str        # text that is embedded; derived from
  content
embedding        [float]    # encoder output for retrieval_key
scope            dict       # {"env": ..., "task_type": ...}; hard
  filter before scoring

# provenance (required by the memory-source study, Section 5)
source_task_ids [str]       # evolving tasks this entry was derived
  from
source_outcome   str        # success | failure | mixed
writer_model     str        # model that produced the entry
created_at_step int         # index of the evolving episode that
  created it
updated_at_step int
version          int        # incremented on every REVISE or MERGE
superseded_by    str|null   # soft deletion: points at the entry that
  replaced it

# verification bookkeeping (uniform across types)
support          int        # episodes in which the entry was followed
  and the task succeeded
refute           int        # episodes in which it was followed and the
   task failed

# retrieval statistics (used by utility-based eviction)
n_retrieved          int
n_retrieved_success int
```

Two placements in this record are deliberate. First, `support` and `refute` live in the envelope rather than inside `content`, because verification is a write mechanism and not a property of the rule format; once it applies to every type, its bookkeeping belongs to every type. Second, deletion is soft: a superseded entry is retained with a `superseded_by` pointer so that store dynamics remain reconstructible after a run.

Derived quantities used by the write policy are

$$\text{conf} = \frac{\mathit{support}+1}{\mathit{support}+\mathit{refute}+2}, \qquad \text{util} = \frac{n_{\text{ret-succ}}+1}{n_{\text{ret}}+2}, \qquad \text{prio} = \text{util}\cdot\big(1+\log(1+n_{\text{ret}})\big), \tag{1}$$

with Laplace-smoothed confidence driving refinement and deletion, and prio driving eviction when the store exceeds its capacity.

B.2 Reflection

Schema (as shown to the writer).

```
situation:   str   # the situation this applies to, 1 sentence
lesson:      str   # transferable lesson, imperative, 1-2 sentences
rationale:   str   # why it holds (causal), 1-2 sentences
evidence:    str   # a VERBATIM snippet from the trajectory, <=2 lines
outcome_tag: str   # from_success | from_failure | from_contrast
```

Validity. `situation`, `lesson`, and `rationale` must be non-empty; `outcome_tag` must be one of the three listed values; and `evidence` must be non-empty, and locatable in the source trajectory. The writer is additionally instructed that `lesson` must be imperative and actionable rather than a description of what happened.

Retrieval key. `situation + " " + lesson`.

Injected rendering. Under the header `## Past experience (lessons):`

```
[when <situation>] <lesson>
   why: <rationale>
```

Example (ALFWorld).

```
situation:   you need to heat an object but are already holding a
    different one
lesson:      confirm the held object is the target before going to the
    microwave;
             otherwise put the current object down first and take the
    target
rationale:   the environment allows only one held object and heating
    applies to
             the held object, so heating the wrong one fails without an
    error
evidence:    "> heat tomato 1 with microwave 1  |  Nothing happens."
outcome_tag: from_failure
```

B.3 RULE

Schema (as shown to the writer).

```
trigger:   str        # "the search page shows no item matching all
    constraints"
                      # MUST be decidable from the current observation
directive: str        # exactly ONE action or action-constraint
polarity:  str        # do | avoid
exception: str|null   # when the rule does NOT apply
evidence:  str        # a VERBATIM snippet from the trajectory that
    motivates it
```

Validity. `trigger` and `directive` must be non-empty; a `directive` containing a coordinating conjunction ("and then", "; then", "then", "and also") is rejected as a compound action and must be split into separate rules; and the same evidence conditions as above apply. The writer prompt additionally requires that `trigger` be a predicate evaluable on the spot from the current observation or the last action's feedback—conditions such as "when the task is hard" are invalid—and that `directive` name an action in the environment's action space or a constraint on choosing among actions. Before emitting a rule, the writer is asked to self-check the trigger by estimating the fraction of steps on which it would fire, and to rewrite or drop the rule if that fraction approaches one.

Retrieval key. `trigger + " " + directive`.

Injected rendering. Under the header `## Rules`, in descending confidence order:

```
When <trigger>, DO <directive>. (unless <exception>)
When <trigger>, AVOID <directive>.
```

Confidence itself is not injected, so that the agent cannot treat a low value as licence to ignore an entry; it is used only for ordering and eviction.

Example (WebShop).

```
trigger:   the search results page shows no item matching all attribute
    constraints
directive: relax the least-specified attribute and re-search
polarity:  do
exception: fewer than 3 results are returned, in which case open the top
     result
           first to read its options
evidence:  "Search results: no products found matching 'wireless 3.5mm
    ...'"
```

B.4 SKILL

Schema (as shown to the writer).

```
name:          str        # verb phrase, snake_case
trigger:       str        # task-level condition for using this
    procedure
preconditions: [str]      # state that must hold before executing
steps:         [str]      # 3..12 ordered steps, use <obj>/<recep>
    placeholders,
                          # never concrete instance names
verification:  [str]      # how to confirm each key step succeeded
fallback:      [str]      # recovery when verification fails
evidence:      str        # a VERBATIM snippet from a trajectory this
    came from
```

Validity. `name` and `trigger` must be non-empty; `steps` must contain between 3 and 12 entries; `verification` must be non-empty, since a procedure with no success check is not reusable; and the same evidence conditions apply. Steps must use placeholders rather than the concrete instance names observed in the source rollouts. A procedure requiring more than twelve steps is treated as two skills, and the writer is instructed to emit the more reusable one.

Retrieval key. `name + " " + trigger + " " + join(steps[1:3])`.

Injected rendering. Under the header `## Applicable procedures`; when `verification` and `fallback` are index-aligned with `steps` they are attached inline, otherwise they are listed after the steps:

```
### <name>
Use when: <trigger>
Preconditions: <p1>; <p2>
Steps:
  1. <step>   [check: <verification>]   [if fails: <fallback>]
  2. ...
```

Example (AppWorld).

```
name:          resolve_entity_id_before_mutation
trigger:       the task requires modifying or deleting a named entity
               (contact, playlist, order)
preconditions: ["logged into the relevant app",
                "the natural-language name of the entity is known"]
steps:         ["call the app's search/list API filtered by the name to
    obtain
                 candidate entities",
                "if more than one candidate remains, disambiguate using
    the
                 additional constraints in the task (time, owner, status
    )",
                "call the mutation API with the single disambiguated id
    ",
                "re-read the entity and confirm the field changed"]
verification:  ["search returns a non-empty result whose fields match
    the task",
                "the mutation API returns 2xx and the response id
    matches step 2",
                "the re-read field equals the target value"]
fallback:      ["if search returns nothing, page through the full list
    API and
                 match locally",
                "if more than one candidate remains and cannot be
    disambiguated,
                 ask the supervisor rather than picking arbitrarily"]
```

B.5 RAW TRAJECTORY (BASELINE)

Schema.

```
instruction: str     # the task instruction
trajectory:  str     # the abstracted observation-action trace
reward:      float   # achieved reward
```

Validity. `instruction` and `trajectory` must be non-empty. There is no grounding field, because the entry *is* the trajectory, and no LLM writer is involved: the store is append-only over episodes with exact deduplication.

Retrieval key. `instruction`.

Injected rendering. Under the header `## Similar solved tasks`:

```
Task: <instruction>
<trajectory>
```

B.6 WRITER INTERFACE

All writers share one prompt skeleton; the type-specific schema block above is substituted verbatim into the `[SCHEMA]` field, and the type-specific extraction instruction into `[TASK]`.

```
[ROLE]        memory writer for an agent operating in <env>
[INPUT]       task instruction, scope, selected rollouts, retrieved
   entries
[TASK]        <type-specific extraction instruction>
[SCHEMA]      <type-specific schema block>
[CONSTRAINTS] <shared constraints + type-specific constraints>
[OUTPUT]      Reply with JSON only, no prose:
              {"ops": [{"op": "APPEND"|"REVISE"|"DELETE",
                        "target_id": "<id, REVISE/DELETE only>",
                        "content": {<schema fields>},
                        "reason": "<short>"}]}
              An empty list is a valid and often correct answer.
```

Four constraints are applied to every type: entries must transfer to other task instances and must never encode one-off instance names; an entry already present in the retrieved memory must not be restated, and a writer that considers an existing entry wrong must emit a `REVISE` with its `target_id` instead of a near-duplicate `APPEND`; at most $n_{\max}$ entries may be emitted per episode, with an empty list explicitly sanctioned; and `evidence` must be copied verbatim from the trajectory, with entries whose evidence cannot be located being discarded.

The rollouts shown to the writer are selected identically for all types: for a mixed episode, one successful and one failed rollout; for an all-successful episode, the shortest success; for an all-failed episode, the longest failure. Without this rule, a comparison of contents would be partly a comparison of how much of the episode each writer was allowed to see.