VANTAGE: LEARNING MEMORY ADVANTAGE BY ASYMMETRIC RE-ATTEMPT

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic memory enables LLM agents to self-evolve without weight updates by accumulating experience in a memory bank and retrieving it for new queries. As the bank grows, the key is to learn each memory's *utility* score so that retrieval prioritizes the most useful memories. Existing methods learn utility from the outcome (success or failure) of each attempt, crediting a memory whenever it is retrieved in a successful attempt, even when the attempt would succeed without it. We therefore argue that utility should also account for memory advantage, *i.e.*, the outcome change that a memory brings compared to a memory-free attempt on the same query. To this end, we propose Vantage, an efficient utility-learning framework that incorporates outcome changes without the cost of memory-free attempts: the change is approximated by immediately re-attempting the query with the updated memory bank, so the first attempt itself serves as the reference. The re-attempt is asymmetric: only failed queries are re-attempted, whereas successful queries, whose outcome rarely changes, are not; this limits repeated crediting and redundant memories. Under frozen-bank evaluation on diverse tasks spanning coding, computer use, and embodied planning, Vantage improves success rate (SR) over the strong baseline MemRL by an absolute 6.4% on training queries and 2.2% on held-out queries, with 40% fewer attempts and thus a 40% lighter memory bank. Incorporating outcome changes into MemRL also improves it, further validating the advantage view. Detailed ablations confirm that both the asymmetric re-attempt and the incorporation of outcome changes drive the gains, and the learned utilities align more closely with memory advantage.

1 INTRODUCTION

Large language model (LLM) agents (Sumers et al., 2023; Zhang et al., 2025b) increasingly operate in interactive environments, from operating systems and databases (Zheng et al., 2025) to embodied households (Shridhar et al., 2020) and code generation (Zhuo et al., 2025), where similar situations recur across queries. A frozen LLM backbone, however, does not carry experience from one query to the next, and updating its weights is costly and may interfere with previously acquired capabilities (Kirkpatrick et al., 2017). Agentic memory fills this gap by accumulating experience in a memory bank (Packer et al., 2023) and reusing it for new queries (Gao et al., 2025; Huang et al., 2026). Existing work follows three lines: memory construction designs what is stored, such as reflections (Shinn et al., 2023) and procedural workflows (Fang et al., 2026); memory retrieval selects memories by semantic similarity (Wang et al., 2023) or handcrafted rules (Park et al., 2023); and learning-based methods learn from interaction outcomes (Ye et al., 2026), training a memory manager (Yu et al., 2026) or learning a utility score for each memory (Zhang et al., 2023; 2026).

Learning-based methods are promising because they improve the agent with experience from its own interaction, learned from the outcomes the environment returns and complementing static corpora that have largely been consumed. A commonly used method (Zhang et al., 2026; Liao et al., 2026; Yang et al., 2026b) attaches a learned *utility* to each stored memory. For each query, the agent acts with memories retrieved by similarity and utility in context. The outcome of this attempt is mapped to a reward that updates their utilities, and the attempt is written as a new memory. Utility is the only learned quantity in this loop and decides which memories the agent conditions on, so learning it well comes down to what the reward is computed from. That reward is the outcome alone, so whenever the attempt succeeds, every memory retrieved for it is credited, even when the attempt

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.

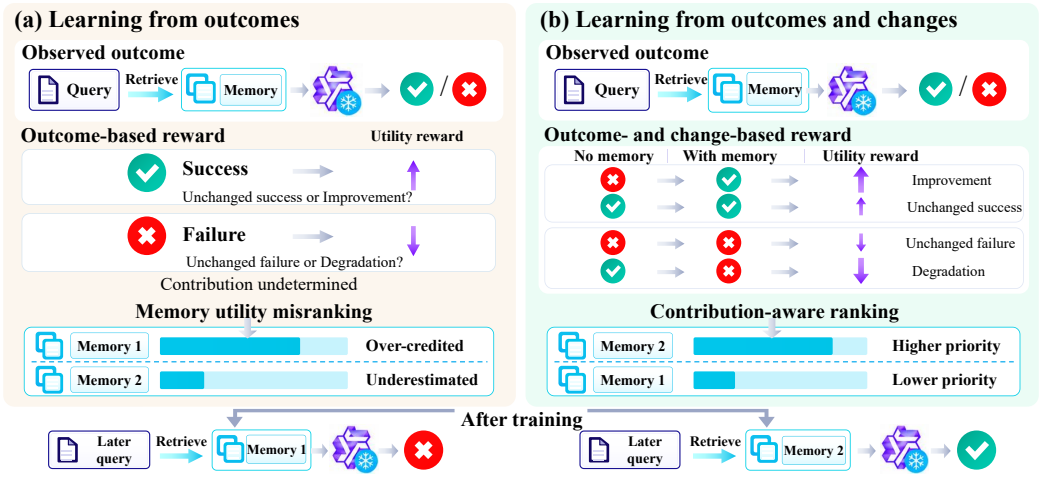

Figure 1: Comparison of (a) learning utility from the outcome alone, where a retrieved memory is credited whenever the attempt succeeds, and (b) learning utility from the outcome and memory advantage, where the reward also reflects the outcome change relative to a memory-free attempt.

would succeed without those memories. Queries that the backbone already solves succeed in every epoch, so their memories are credited repeatedly and redundant memories are written each time. These over-credited memories then outrank more useful candidates for other queries (Figure 1a).

This motivates rethinking what utility should measure. We argue that utility should also account for *memory advantage*, *i.e.*, the outcome change that a memory brings compared to a memory-free attempt on the same query (Figure 1b). Advantage complements the outcome rather than replacing it: keeping a query solved still deserves credit, turning a failure into a success deserves more, and turning a success into a failure deserves less than an unchanged failure. To test this view, we update MemRL with a reward that combines the outcome and this change; the variant, MemRL$^{\Delta}$, already improves over MemRL (Section 4.3). Measuring the change directly, however, requires a memory-free attempt on every query, and successful queries are still credited and written in every epoch.

We therefore propose **Vantage**, an efficient utility-learning framework that incorporates outcome changes without the cost of memory-free attempts (Figure 2). Vantage approximates the change by immediately re-attempting the query with the updated memory bank, so the first attempt itself serves as the reference. Because both attempts share the same memory-free outcome, comparing their combined rewards reduces to comparing their outcomes, and no memory-free attempt is needed (Section 3.3). The re-attempt is asymmetric: only failed queries are re-attempted, so a successful re-attempt marks an improvement over the first attempt. A successful query is credited and written once per epoch, since its re-attempt would almost always succeed again and would only credit the same memories and add a redundant memory. Each query is thus attempted at most twice per epoch, so Vantage reaches the per-query attempt budget of MemRL in half the epochs.

We evaluate Vantage on diverse benchmarks, including BigCodeBench (Zhuo et al., 2025), LifelongAgentBench (OS and DB) (Zheng et al., 2025), and ALFWorld (Shridhar et al., 2020), covering coding, computer use, and embodied planning. With Qwen3-8B under frozen-bank evaluation, five-epoch Vantage exceeds ten-epoch MemRL in average success rate (SR) by an absolute 6.4% on training queries and 2.2% on held-out queries, with 40% fewer attempts and thus a 40% lighter memory bank. Ablations confirm that both the asymmetric re-attempt and the incorporation of outcome changes drive the gains (Section 4.3), and the learned utilities align more closely with memory advantage (Section 4.4). Our contributions are threefold:

- We identify over-crediting in outcome-based utility learning and argue that utility should also account for memory advantage, as supported by the gains of MemRL$^{\Delta}$ over MemRL.
- We propose Vantage, which approximates memory advantage through an asymmetric re-attempt and thus incorporates outcome changes efficiently, without memory-free attempts.
- Under frozen-bank evaluation, Vantage improves SR on training queries and on held-out queries across three benchmarks, and its learned utilities align more closely with memory advantage.

2 Related Work

External memory is central to self-evolving agents, but as memory scales, the challenge increasingly shifts toward identifying and retrieving valuable experiences. Recent research has progressed from improving memory organization to learning memory utility from task outcomes.

Memory for self-evolving agents. External memory enables agents to accumulate and reuse experience (He et al., 2026; Hu et al., 2025b) without repeatedly updating the backbone (Gao et al., 2025; Huang et al., 2026). Early systems store trajectories, reflections (Shinn et al., 2023), or structured knowledge (Zhao et al., 2024) and retrieve them using semantic similarity (Zhong et al., 2024; Wang et al., 2023) or handcrafted rules (Packer et al., 2023; Park et al., 2023). Recent work improves memory construction through graph (Xu et al., 2026; Chhikara et al., 2025; Yang et al., 2026a) and hierarchical organization (Hu et al., 2025a; Kang et al., 2025), distilled strategies (Ouyang et al., 2026; Fu et al., 2024), procedural workflows (Wang et al., 2024; Fang et al., 2026; Banerjee et al., 2026), and consolidation (Liu et al., 2026). As the bank grows, experiences compete for limited context, making memory selection critical to continual adaptation (Hu et al., 2026). Learning-based approaches address this by optimizing memory operations or retrieval decisions from interaction outcomes (Ye et al., 2026). Some train parametric memory managers (Yan et al., 2026; Yu et al., 2026; Ma et al., 2026), retrieval policies (Zhou et al., 2025), or evolving memory architectures (Zhang et al., 2025a), whereas others attach a learned utility to each stored experience (Zhang et al., 2023). We focus on the latter setting and follow MemRL's two-phase retrieval (Zhang et al., 2026), where similarity constructs a candidate pool and utility reranks it in final selection.

Memory utility. Within utility-based retrieval, a central question is how the outcome of an attempt should be translated into a memory's utility. Rememberer updates experience utility from downstream rewards (Zhang et al., 2023), and MemRL assigns the task outcome jointly to the retrieved memories. MemQ propagates credit through memory provenance chains (Liao et al., 2026), and RoMeRL refines attribution among co-retrieved memories through observational and marginal utility (Yang et al., 2026b). These methods refine how the outcome is distributed among memories, but the reward itself remains the outcome, so a memory is credited even when the attempt would succeed without it. Work on memory reward inflation studies a different source of over-crediting, where inaccurate self-evaluation assigns high scores to incorrect experiences and amplifies their reuse (Asadolahi et al., 2026). Closest to our view, MemSifter recognizes that final task performance alone may not reveal the contribution of retrieved memory, and trains a proxy retriever using performance lift relative to a memory-free reference, together with progressive Top-K evaluations (Tan et al., 2026). Unlike MemSifter, we learn a utility for each stored memory rather than training a separate retriever, and Vantage approximates the outcome change by an asymmetric re-attempt with the first attempt as the reference, so no memory-free attempt is needed.

3 Methodology

Section 3.1 formalizes utility learning under two-phase retrieval, following MemRL. Section 3.2 incorporates memory advantage, the outcome change compared to a memory-free attempt, into the reward that updates utility. Section 3.3 presents Vantage, which approximates the outcome change by an asymmetric re-attempt that uses the first attempt as the reference.

3.1 Problem Formulation

We consider an agent with a frozen LLM backbone f_{llm} that adapts to a stream of queries $\{q_1, q_2, \dots\}$ through a memory bank $\mathcal{M} = \{m_1, m_2, \dots\}$. Following MemRL (Zhang et al., 2026), each memory is a triplet $m_i = (z_i, e_i, Q_i)$: z_i is a query embedding computed by a frozen embedder f_{emb}; e_i is the stored experience; and Q_i is a scalar utility. For each query q, the retriever μ returns a context $\mathcal{C} = \mu(q, \mathcal{M}) \subseteq \mathcal{M}$ of memories selected by similarity and utility, and the backbone produces an action a for q given $\mathcal{C}$, $a \sim f_{\text{llm}}(\cdot \mid q, \mathcal{C})$. This retrieval and action form an *attempt*, whose outcome y is mapped to a reward r that updates the utilities of the memories in $\mathcal{C}$; the attempt is then written to $\mathcal{M}$ as a new memory. Learning thus updates only the utilities $\{Q_i\}$, which shape the agent's behavior only through retrieval, while f_{llm} stays fixed. The agent passes over the queries repeatedly, and each pass is an *epoch*. We next specify two-phase retrieval, then the utility update and memory writing.

Two-phase retrieval. Phase A computes the cosine similarity ρ_i between $f_{\text{emb}}(q)$ and z_i for every $m_i \in \mathcal{M}$ and keeps as candidates the memories above a similarity threshold δ, up to the k_1 most similar,

$$\mathcal{P} = \{\, m_i \in \mathcal{M} \mid \rho_i > \delta,\ \text{rank}(\rho_i) \le k_1 \,\}.$$

Phase B ranks the candidates by a ranking score that combines similarity and utility,

$$\psi_i = (1-\lambda)\,\widehat{\rho}_i + \lambda\,\widehat{Q}_i, \qquad m_i \in \mathcal{P}, \tag{1}$$

where hats denote z-score normalization within $\mathcal{P}$ and $\lambda \in [0,1]$ weights utility. The context $\mathcal{C} \subseteq \mathcal{P}$ consists of the k_2 candidates with the highest ψ_i.

Utility update and memory writing. The environment scores the action a and returns the outcome $y = f_{\text{env}}(q,a)$, with $y > 0$ on success and $y \le 0$ on failure. This outcome is mapped to a reward r, toward which every memory in $\mathcal{C}$ is updated,

$$Q_i \leftarrow Q_i + \alpha\,(r - Q_i), \qquad m_i \in \mathcal{C}, \tag{2}$$

with step size α; each Q_i thus estimates the expected reward of attempts that include m_i in their context. MemRL learns from the outcome alone and sets $r = y$; Section 3.2 adds memory advantage to r. Finally, a frozen summarizer f_{sum} distills the attempt, successful or failed, into an experience $e = f_{\text{sum}}(q,a,y)$, which is written as a new memory $(z,\, e,\, Q_{\text{init}})$ with $z = f_{\text{emb}}(q)$ and initial utility Q_{init}; MemRL and Vantage use the same summarizer.

3.2 Memory Advantage

The reward $r = y$ credits a memory whenever the attempt succeeds, whether or not it would have succeeded without that memory. We therefore incorporate memory advantage, the outcome change compared to a memory-free attempt on the same query, into the reward. Let $a_\varnothing \sim f_{\text{llm}}(\cdot \mid q)$ and $y_\varnothing = f_{\text{env}}(q, a_\varnothing)$ denote the action and outcome of that memory-free attempt, respectively. Since the with-memory outcome y and the memory-free outcome $y_\varnothing$ are each a success or a failure, write $y, y_\varnothing \in \{0,1\}$ (Appendix B.1 covers the other outcome values); the pair $(y_\varnothing, y)$ then falls into one of the 4 cases of Table 1: improvement and degradation, where the memory changes the outcome, and unchanged success and unchanged failure, where it does not. Neither signal alone separates all 4 cases: the with-memory outcome y gives improvement and unchanged success the same reward, and the change $y - y_\varnothing$ gives unchanged success and unchanged failure the same reward. Combining the two separates all 4 cases, so we define the reward as

$$r = (1-\eta)\,y + \eta\,(y - y_\varnothing) = y - \eta\,y_\varnothing, \tag{3}$$

where $\eta \in [0,1]$ weights memory advantage.

Table 1: The 4 cases of memory-free outcome $y_\varnothing$ and with-memory outcome y, and the reward r of Eq. (3).

	$y_\varnothing$	y	r
Improvement	0	1	1
Unchanged success	1	1	$1-\eta$
Unchanged failure	0	0	0
Degradation	1	0	$-\eta$

Table 1 lists the reward in each case. At $\eta = 0$ this recovers the MemRL reward $r = y$, which needs no memory-free attempt; at $\eta = 1$ it keeps only the outcome change; and for $0 < \eta < 1$ the rewards are ordered *"improvement > unchanged success > unchanged failure > degradation"*. The middle inequality is by design: an unchanged success implies that the memory at least did not hurt and may have helped, whereas an unchanged failure implies that it did not help enough and may have hurt. Any $0 < \eta < 1$ preserves this ordering; we take the midpoint $\eta = 0.5$, which weights the outcome and its change equally, and call MemRL updated with this reward MemRL$^{\Delta}$. Its gains over MemRL support incorporating outcome changes into utility learning (Section 4.3). However, MemRL$^{\Delta}$ needs a memory-free attempt on every query, and it attempts a query that keeps succeeding as often as one that fails, crediting the same memories and writing a similar memory each time.

3.3 Vantage

Vantage approximates the outcome change by re-attempting a query. The re-attempt retrieves from the bank that the first attempt has just updated, and its outcome is compared with that of the first attempt, which thus serves as the reference in place of a memory-free attempt. The re-attempt is

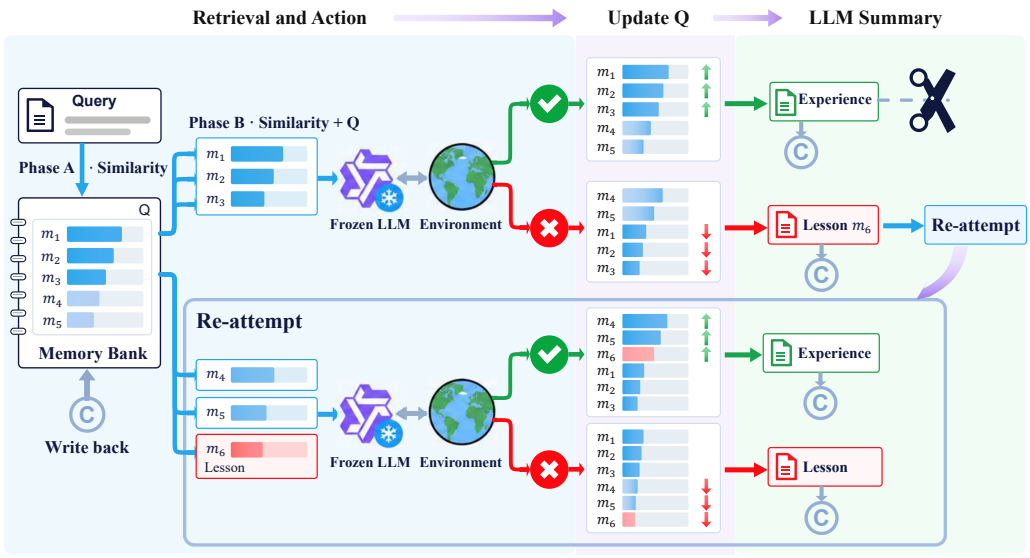

Figure 2: Vantage allows up to two attempts on a query within an epoch. Phase A builds a similarity pool; Phase B ranks by similarity and utility. A failed first attempt is re-attempted once, retrieving from the updated bank; a successful one is not re-attempted in that epoch.

asymmetric, taken once and only immediately after a failed attempt. This is because a re-attempt after a success would almost always succeed again, crediting the same memories a second time and writing a similar memory. Figure 2 and Algorithm 1 (Appendix C.1) give the procedure; every attempt follows the retrieval, utility update, and memory writing of Section 3.1.

Re-attempt approximates the outcome change. Let $y^{(1)}$ and $y^{(2)}$ be the outcomes of the first attempt and the re-attempt on query q. The re-attempt is made immediately, so the bank it retrieves from differs from that of the first attempt only by the first utility update and the experience just written, and the outcome change $y^{(2)} - y^{(1)}$ provides a within-query comparison before and after these changes. Both attempts share the memory-free outcome $y_\varnothing$ of the same query, so $y_\varnothing$ cancels in Eq. (3) and the rewards of the two attempts differ by $y^{(2)} - y^{(1)}$ for every η. Motivated by this comparison, Vantage sets $\eta = 0$, rewarding every attempt by its outcome, which keeps this difference without any memory-free attempt. The approximation uses the first memory-assisted attempt as a reference for comparing outcomes before and after the memory update, while utilities are updated separately using each attempt's outcome.

We compare Vantage with Vantage$^{\Delta}$, which keeps the asymmetric re-attempt but sets $\eta = 0.5$, so each update uses an explicit memory-free reference. The two reach similar average performance (Table 5), and their learned utilities relate to memory advantage almost identically: across bins of learned utility on OS, the share of outcome changes that are improvements over the memory-free success rate (Section 4.4) differs between them by 2.9 percentage points on average, against 11.4 between Vantage and MemRL (Figure 9 in Appendix E.1).

No re-attempt after success. A re-attempt after a successful first attempt almost always succeeds again, in at least 98.3% of cases on every domain (Appendix D.5), because the memories credited by the first attempt are ranked higher and largely retrieved again. Such a re-attempt is unnecessary: as an unchanged success it measures no outcome change, and it would only credit those memories a second time and write a similar memory. Therefore, Vantage re-attempts only after a failure, where the outcome can still change. This rule also saves attempts: an epoch takes one attempt per query plus one re-attempt per failed first attempt, so its number of attempts decreases as more first attempts succeed, whereas re-attempting after a success as well would keep it at two per query. Since every attempt writes one memory, the memory bank also grows more slowly. Section 4 compares the two rewards, whether successful queries are re-attempted, and how many re-attempts follow a failure.

4 EXPERIMENTS

4.1 SETUP

Baselines and benchmarks. We compare Vantage with Mem0 (Chhikara et al., 2025), MemP (Fang et al., 2026), MemRL (Zhang et al., 2026), and MemRL$^{\Delta}$ (Appendix C.5) on 4 domains: Code (BigCodeBench (Zhuo et al., 2025)), OS and DB (LifelongAgentBench (Zheng et al., 2025)), and Planning (ALFWorld (Shridhar et al., 2020)). Mem0 maintains a memory store of past experience, and MemP distills successful trajectories into workflows. MemRL provides the two-phase retrieval and utility update used by Vantage; MemRL$^{\Delta}$ keeps one attempt per epoch and sets $\eta = 0.5$ in Eq. (3). *No memory* is the frozen backbone without retrieval. All methods use Qwen3-8B as the backbone and follow MemRL's dataset splits; results with an additional backbone are in Appendix D.3.

Metrics. Runtime learning passes over the training queries in epochs, with utility updates and memory writing enabled. SR is the fraction of queries solved in an epoch, where a query counts as solved if any of its attempts in that epoch succeeds, including Vantage's re-attempt; LSR is the SR of the last epoch, and CSR is the fraction of queries solved at least once during runtime learning.

For "No memory", LSR is the SR of one attempt and CSR is Pass@10, sampled at temperature 0.6; all other runs decode greedily. We compare Vantage after 5 epochs with the memory-based baselines after 10 epochs, so that each query receives at most 10 attempts in both. CSR is therefore comparable; LSR is not, since Vantage may attempt a query twice in the final epoch. (So it is reported for reference). Frozen-bank evaluation on the training and held-out queries gives the fair comparison: it fixes the bank from the last epoch and attempts every query once, with memory writing and utility updates disabled; Frozen SR and Held-out SR are the SR on the training and held-out queries, respectively. Vantage after 10 epochs is also reported for reference.

Table 2: **Runtime learning.** Each cell reports **LSR / CSR** (%). Memory-based baselines use 10 epochs. MemRL$^{\Delta}$ uses $\eta = 0.5$. Averages are unweighted means across the 4 domains. For No memory, CSR is Pass@10. Best CSR excluding 10-epoch Vantage is in **bold**. LSR is reported for reference and is not bold.

Method	BigCodeBench	LifelongAgentBench		ALFWorld	Average
	Code	OS	DB	Planning	
No memory	43.1 / 54.0	44.9 / 56.0	83.1 / 90.0	50.3 / 74.8	55.4 / 68.7
Mem0	40.4 / 50.9	48.9 / 63.1	83.9 / 88.6	78.7 / 89.6	63.0 / 73.1
MemP	52.0 / 54.4	56.6 / 66.3	93.4 / 93.9	76.2 / 90.1	69.6 / 76.2
MemRL	54.5 / 57.5	57.4 / 65.7	94.7 / 95.0	82.5 / 94.2	72.3 / 78.1
MemRL$^{\Delta}$	57.5 / 63.8	58.3 / 68.3	94.7 / 95.0	87.0 / 94.6	74.4 / 80.4
Vantage (5 epochs)	61.9 / **64.2**	64.3 / **69.1**	95.6 / **95.8**	93.4 / **94.9**	78.8 / **81.0**
Vantage (10 epochs)	63.3 / 66.0	64.6 / 71.4	96.4 / 96.4	94.6 / 96.2	79.7 / 82.5

4.2 MAIN RESULTS

Runtime learning. The results in Table 2 validate the efficiency of Vantage. With at most as many attempts per query, and 40% fewer in total (Section 4.4), 5-epoch Vantage reaches a higher CSR than every baseline on all 4 domains (Table 2); its average CSR is 81.0, compared to 78.1 for MemRL and 80.4 for MemRL$^{\Delta}$. MemRL$^{\Delta}$ improves over MemRL on average, so incorporating outcome changes already helps with one attempt per epoch; Vantage leads both without any memory-free attempt, and Section 4.3 separates the contributions of incorporating outcome changes and of the asymmetric re-attempt. Vantage also keeps improving with a larger budget: with 10 epochs, its CSR rises further on every domain, to an average of 82.5.

Frozen-bank evaluation. With the bank frozen, 5-epoch Vantage leads every baseline on Frozen SR and Held-out SR on all 4 domains (Table 3). Its average Frozen SR is 78.9, an absolute 6.4% and 4.4% above MemRL (72.5) and MemRL$^{\Delta}$ (74.5) after 10 epochs, and its average Held-out SR is 67.2, 2.2% and 1.4% above them (65.0 and 65.8). Since memory writing and utility updates are disabled, these margins come from the memories and utilities learned during runtime learning. A single attempt with this bank also exceeds the Pass@10 of No memory on every domain, with an average of 78.9 against 68.7 (Table 2), so the learned memory does better than ten samples without

Table 3: **Frozen-bank evaluation.** Each cell reports **Frozen SR / Held-out SR** (%), with memory writing and utility updates disabled. Memory-based baselines are evaluated after 10 epochs. Vantage after 10 epochs is shown separately for reference.

Method	BigCodeBench	LifelongAgentBench		ALFWorld	Average
	Code	OS	DB	Planning	
No memory	43.1 / 40.9	44.9 / 44.7	83.1 / 82.7	50.3 / 45.7	55.4 / 53.5
Mem0	40.4 / 40.9	48.9 / 46.0	83.9 / 83.5	79.0 / 75.7	63.1 / 61.5
MemP	52.0 / 41.5	57.4 / 52.7	93.4 / 84.2	75.9 / 75.0	69.7 / 63.4
MemRL	53.4 / 41.8	57.1 / 53.3	95.0 / 85.6	84.6 / 79.3	72.5 / 65.0
MemRL$^{\Delta}$	57.5 / 42.1	58.9 / 54.0	94.7 / 85.6	86.8 / 81.4	74.5 / 65.8
Vantage (5 epochs)	**62.3 / 43.0**	**64.3 / 56.0**	**95.3 / 87.8**	**93.7 / 82.1**	**78.9 / 67.2**
Vantage (10 epochs)	63.3 / 43.6	64.6 / 58.0	96.4 / 88.5	94.6 / 81.4	79.7 / 67.9

it. The lead on the held-out queries shows that the learned utilities transfer beyond the training queries. With 10 epochs, Vantage raises both further, to 79.7 and 67.9.

4.3 Asymmetric Re-attempt and Outcome Changes

The ablations test the asymmetric re-attempt and the incorporation of outcome changes. One comparison varies η in Eq. (3). The others ask whether a successful query is re-attempted, and how many re-attempts follow a failure. These comparisons share the same two-phase retrieval and the same summarizer. Frozen SR and Held-out SR follow Section 4.1.

Incorporating outcome changes. Combining the outcome with memory advantage can outperform either alone. Table 4 varies η in Eq. (3) on OS, with one attempt per epoch as in MemRL. At $\eta = 0.75$, LSR, CSR, Frozen SR, and Held-out SR are all higher than at $\eta = 0$ and at $\eta = 1$. Held-out SR is 56.0, compared to 53.3 for the outcome alone and 54.7 for the outcome change alone. The preferred weight varies by metric: $\eta = 0.25$ leads LSR and Frozen SR, while $\eta = 0.25$ and $\eta = 0.5$ tie on CSR. $\eta = 0.5$, used by MemRL$^{\Delta}$, improves all four metrics over $\eta = 0$. These results support retaining credit for a successful attempt while accounting for memory advantage, as in Section 3.2.

Table 4: Weight η on memory advantage on OS. LSR / CSR, Frozen SR, and Held-out SR (%).

η	LSR / CSR	Frozen SR	Held-out SR
0	57.4 / 65.7	57.1	53.3
0.25	**59.4 / 68.3**	**59.1**	53.3
0.5	58.3 / **68.3**	58.9	54.0
0.75	59.1 / 66.9	58.9	**56.0**
1	58.3 / 66.0	58.3	54.7

One re-attempt after a failure works best. Let d be the maximum number of re-attempts after a failure; a success ends the attempts on a query, and $d = 1$ is Vantage. Figure 3 compares $d = 1, 2, 3$ at a matched attempt budget, the total of the 5-epoch run at $d = 3$, so the runs at $d = 1$ and $d = 2$ continue past 5 epochs. All three depths reach about the same CSR (70.7 on OS; 96.0, 95.6, and 95.6 on DB), whereas Held-out SR falls with depth: 56.7, 55.3, and 54.7 on OS, and 87.8, 86.3, and 85.6 on DB. The first re-attempt (Vantage) tests, on the same query, whether the update and the experience just written change the outcome (Section 3.3). Further re-attempts spend the budget on a query that has already failed twice without raising CSR, and since their utility updates and writes all come from that query, the learned ranking transfers less to other queries and Held-out SR falls.

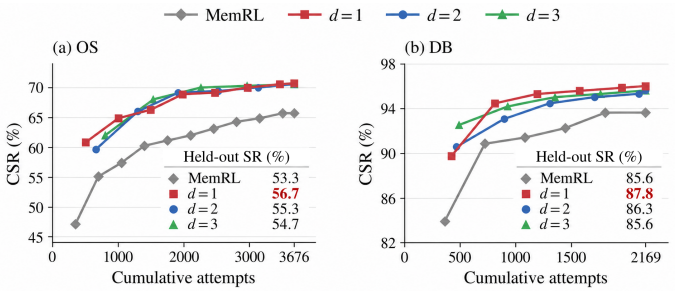

Figure 3: CSR versus cumulative attempts on OS and DB for MemRL and $d = 1, 2, 3$. Insets report Held-out SR with memory banks frozen at the common attempt budget, determined by the five-epoch run at $d = 3$.

Re-attempt approximates the outcome change. Table 5 crosses whether a successful query is also re-attempted with $\eta \in \{0, 0.5\}$; all 4 settings re-attempt a failed query. Under the asymmetric re-

Table 5: The four settings cross re-attempting a successful query and $\eta \in \{0, 0.5\}$; all four re-attempt a failed query. Entries are Frozen SR and Held-out SR (%). Average is over the 4 domains; Code, Planning, and LSR / CSR are in Appendix D.1. Best in **bold**, second underlined. Qwen3-8B.

	OS		DB		Average	
	Frozen	Held-out	Frozen	Held-out	Frozen	Held-out
re-attempt after success, $\eta = 0$	59.7	54.0	94.7	84.9	75.7	64.3
re-attempt after success, $\eta = 0.5$	63.1	59.3	95.0	84.9	76.9	65.8
Vantage	64.3	56.0	**95.3**	**87.8**	78.9	**67.2**
Vantage$^{\Delta}$	**66.3**	**60.0**	**95.3**	85.6	**79.1**	67.0

attempt, adding the memory-free reference barely changes the average: Vantage$^{\Delta}$ ($\eta = 0.5$) reaches 79.1 Frozen SR and 67.0 Held-out SR, against 78.9 and 67.2 for Vantage ($\eta = 0$), while running a memory-free attempt on every query. The first attempt thus suffices as the reference, and the memory-free attempt is unnecessary. When successful queries are also re-attempted, $\eta = 0.5$ does help, raising average Frozen SR from 75.7 to 76.9 and Held-out SR from 64.3 to 65.8, as it does for MemRL (Table 3): at $\eta = 0$, these settings credit every unchanged success in full.

No re-attempt after success. Not re-attempting a successful query raises Frozen SR and Held-out SR at both values of η, with fewer attempts: on average from 75.7 / 64.3 to 78.9 / 67.2 at $\eta = 0$ (Vantage) and from 76.9 / 65.8 to 79.1 / 67.0 at $\eta = 0.5$ (Vantage$^{\Delta}$); the same holds on every domain (Appendix D.1).

4.4 Analysis

We examine how closely learned utility tracks memory advantage, the efficiency of the asymmetric re-attempt, and how many redundant memories are written.

Utility and memory advantage. Vantage learns utilities that track memory advantage more closely than MemRL: when a retrieved memory changes the outcome, the change is more often an improvement. Let P be a query's success rate over ten memory-free attempts and $y \in \{0, 1\}$ its outcome on one with-memory attempt: an improvement is $y - P > 0$, and a degradation is $y - P < 0$. Figure 4 keeps only these two outcome changes on OS and groups them by the utility Q of the top-ranked memory, following MemRL's analysis (Zhang et al., 2026); each bar is the share of improvements in its bin, and the dashed line is that share over all plotted bins. Vantage's share is higher than MemRL's in every bin, and its pooled share is 87.4% against 77.0%; the pooled share is also higher on the other 3 domains, with a four-domain average of 89.8% against 86.3% (Appendix E.1). Vantage$^{\Delta}$, whose updates use a memory-free attempt on every query, reaches the same 89.8%, so utilities learned with the first attempt as the reference align with memory advantage as well as those learned with a memory-free attempt.

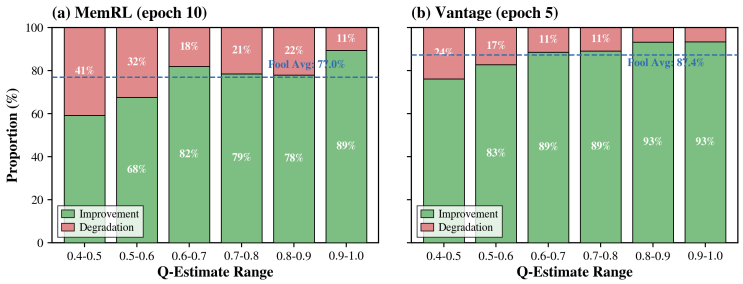

Figure 4: Share of improvements ($y > P$) and degradations ($y < P$) on OS, grouped by the utility Q of the top-ranked memory, where P is the success rate over 10 memory-free attempts. Dashed lines are pooled shares.

Learning over epochs. Figure 5 tracks CSR during runtime learning: Vantage leads from the first epoch on all 4 domains, and no baseline reaches its 5-epoch CSR, marked by the dashed line, within 10 epochs. The lead over MemRL is largest on Code and OS, where many first attempts fail and a re-attempt can still turn them into successes, and smallest on DB and Planning, where every method is close to saturation.

Learning per attempt. Part of the early lead comes from the re-attempts within each epoch, but the attempt axis compares all methods at equal cost, and Vantage still has the highest CSR at every matched number of attempts on all 4 domains. Its cost also adapts to the domain, since only failed

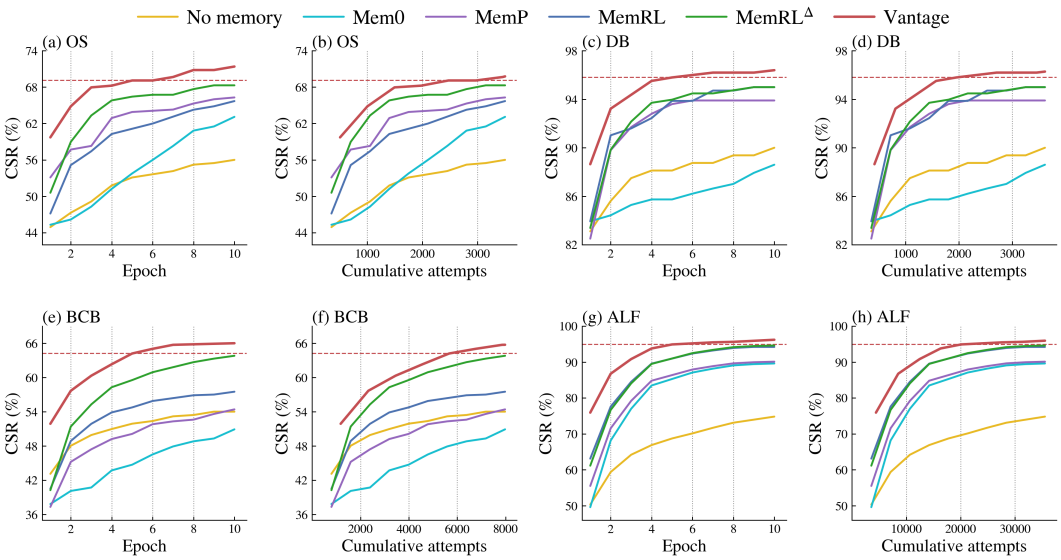

Figure 5: Runtime-learning CSR on OS, DB, BigCodeBench, and ALFWorld, plotted against the epoch and against the cumulative number of attempts. The attempt axis counts the first attempt and the re-attempt after failure, and leaves out the memory-free attempt MemRL$^{\Delta}$ uses to obtain $y_{\varnothing}$. Each curve is cut at the common attempt budget. The dashed red line marks the CSR of 5-epoch Vantage.

first attempts are re-attempted: over 5 epochs, an epoch costs on average 1.08 and 1.13 attempts per query on DB and Planning, where the lead is smallest, and 1.41 and 1.42 on OS and Code, where it is largest. Compared to 10-epoch MemRL, Vantage thus saves 46%, 43%, 29%, and 29% of the attempts on these domains, 40% in total, and as many memories, since every attempt writes one (Table 14). MemRL$^{\Delta}$ would cost twice as much if its memory-free attempts were counted.

Redundant memories. Vantage writes fewer redundant memories. We count two writes as duplicates when they come from the same source query, have the same outcome, and execute the same command or SQL sequence, and report the ratio of distinct sequences to writes on OS and DB (Figure 6; more results are presented in Appendix E.2). Vantage reaches 73.1% on OS and 32.8% on DB, above 10-epoch MemRL (62.9%, 17.2%) and MemRL$^{\Delta}$ (56.4%, 16.4%), and above the 5-epoch setting that also re-attempts after success (59.9%, 17.8%). That setting shares Vantage's 5 epochs and its re-attempt after failure, so the gap isolates the asymmetry: a re-attempt after success writes another memory for a query that already succeeded, often with the same sequence.

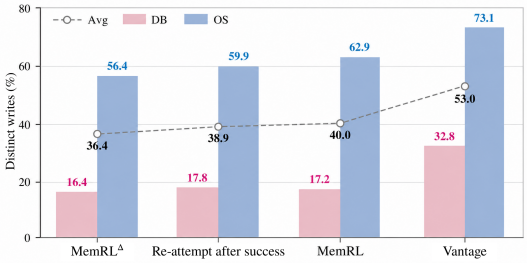

Figure 6: Distinct command or SQL sequences per write on OS and DB (MemRL and MemRL$^{\Delta}$: 10 epochs; others: 5).

5 Conclusion

Existing methods learn memory utility from the outcome alone, crediting a retrieved memory whenever the attempt succeeds, even when it would succeed without that memory. We argue that utility should also account for memory advantage, the outcome change compared to a memory-free attempt, and adding this change to the reward already improves MemRL. Vantage incorporates outcome changes without memory-free attempts: it re-attempts only failed queries, immediately and with the updated memory bank, so the first attempt serves as the reference. Across coding, computer use, and embodied planning, 5-epoch Vantage improves frozen-bank SR over 10-epoch MemRL by an absolute 6.4% on training queries and 2.2% on held-out queries, with 40% fewer attempts and a 40% lighter memory bank, and its learned utilities align more closely with memory advantage. Accounting for outcome changes thus makes utility learning both better and cheaper. Limitations are discussed in Appendix G.

AI USE STATEMENT

In this work, we used generative AI tools for editing manuscript text. We have not used generative AI tools to generate experimental results, figures, or bibliographic entries. We have reviewed all AI-assisted writing. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies memory updates for frozen-backbone LLM agents on public benchmarks and does not involve human subjects or the release of new datasets. We do not foresee additional ethical concerns beyond those already associated with LLM agents.

REPRODUCIBILITY STATEMENT

Notation is summarized in Table 6. The problem setup and utility update are given in Section 3.1. The reward in Eq. (3), the asymmetric re-attempt, and Vantage are specified in Section 3 and Algorithm 1. Success and failure outcomes are in Appendix B.1. The z-score in Eq. (1) is stated in Section 3.1 and Appendix C.3. Implementation details, hyperparameters, and data splits are in Appendix C; prompt templates, action interfaces, and schematic traces are in Appendix F. Bank size, Top-1 utility versus improvement and degradation, and distinct command or SQL sequences are in Section 4.4, with additional details in Table 14 and Appendices E.1 and E.2. Comparisons of η, not re-attempting after success, and the number of re-attempts after failure are in Section 4.3 and Figure 3. Re-attempts after a successful first attempt are in Appendix D.5. Benchmarks, the runtime-learning and held-out splits, and the primary success-rate metric follow the retrieval-utility baseline. Runtime LSR and CSR numbers are in Table 2. Frozen-bank success rates on training queries and held-out queries are in Table 3. Bank sizes and attempt counts during runtime learning are in Table 14. Top-1 utility versus improvement and degradation on OS is in Figure 4. Results with additional backbones are in Appendix D.3.

REFERENCES

Mohammad Asadolahi, Amir Amini, Samira Talebi, Amirfarhad Farhadi, and Azadeh Zamanifar. Memory reward inflation in self-improving llm agents. *arXiv preprint arXiv:2608.00017*, 2026.

Pratyay Banerjee, Masud Moshtaghi, and Ankit Chadha. Apex-em: Non-parametric online learning for autonomous agents via structured procedural-episodic experience replay. *arXiv preprint arXiv:2603.29093*, 2026.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

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, 2026.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents. *Advances in Neural Information Processing Systems*, 37: 119919–119948, 2024.

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.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025a.

Qisheng Hu, Quanyu Long, and Wenya Wang. When continual learning moves to memory: A study of experience reuse in llm agents. *arXiv preprint arXiv:2604.27003*, 2026.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, et al. Memory in the age of ai agents. *arXiv preprint arXiv:2512.13564*, 2025b.

Wei-Chieh Huang, Wei-Zhi Zhang, Yueqing Liang, Yuanchen Bei, Yankai Chen, Tao Feng, Xinyu Pan, Zhen Tan, Yu Wang, Tianxin Wei, Shanglin Wu, Ruiyao Xu, Liang Yang, Rui Yang, Wooseong Yang, Chin-Yuan Yeh, Han-Rong Zhang, Haozhen Zhang, Si-Qi Zhu, Henry Peng Zou, Wan-Jia Zhao, Song Wang, Wujian Xu, Zi-Xuan Ke, Zheng Hui, Da-Wei Li, Yaozu Wu, Langzhou He, Chen Wang, Xiong Xu, Baixiang Huang, Juntao Tan, Shelby Heinecke, Huan Wang, Caiming Xiong, Ahmed A. Metwally, Jun Yan, Chen-Yu Lee, Han Zeng, Ying Xia, Xiao-Min Wei, Ali Payani, Hai-Tong Ma, Wen-Yu Wang, Chengguang Wang, Yu Zhang, Xin Wang, Yongfeng Zhang, Ji-Chun You, Hang-Hang Tong, Xiao Luo, Yizhou Sun, Wei Wang, Julian McAuley, James Zou, Jia-Wei Han, Philip S. Yu, and Kai Shu. A survey of agent memory in the second half: Towards self-evolving and long-horizon agents. *Trans. Mach. Learn. Res.*, 2026, 2026. URL https://api.semanticscholar.org/CorpusID:285401413.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory os of ai agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25972–25981, 2025.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Junwei Liao, Haoting Shi, Ruiwen Zhou, Jiaqian Wang, Shengtao Zhang, Wei Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Bo Tang, et al. Memq: Integrating q-learning into self-evolving memory agents over provenance dags. *arXiv preprint arXiv:2605.08374*, 2026.

Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Simplemem: Efficient lifelong memory for llm agents. *arXiv preprint arXiv:2601.02553*, 2026.

Weitao Ma, Xiaocheng Feng, Lei Huang, Xiachong Feng, Zhanyu Ma, Jun Xu, Jiuchong Gao, Jinghua Hao, Renqing He, and Bing Qin. Fine-mem: Fine-grained feedback alignment for long-horizon memory management. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19666–19684, 2026.

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.

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.

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*, pp. 1–22, 2023.

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.

Theodore R Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*, 2023.

Jiejun Tan, Zhicheng Dou, Liancheng Zhang, Yuyang Hu, Yiruo Cheng, and Ji-Rong Wen. Memsifter: Offloading llm memory retrieval via outcome-driven proxy reasoning. *arXiv preprint arXiv:2603.03379*, 2026.

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

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

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z Pan, et al. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12805–12825, 2026.

Ke Yang, Zixi Chen, Xuan He, Jize Jiang, Michel Galley, Chenglong Wang, Jianfeng Gao, Jiawei Han, and ChengXiang Zhai. Plugmem: A task-agnostic plugin memory module for llm agents. *arXiv preprint arXiv:2603.03296*, 2026a.

Yi Yang, Zhennan Chen, Yihong Zhuang, Tiehan Fan, Yinan Chen, Jian Li, Jian Yang, and Ying Tai. Romerl: Balancing feedback coverage and the memory-reward trap in self-evolving agent memory via reduced-order utility states. *arXiv preprint arXiv:2608.02508*, 2026b.

Bowen Ye, Yongchao Xu, Zhijian Li, Xiang Yin, Junkai Ma, and Wenzhao Li. Coevo-mem: Co-evolving retrieval policy and memory bank for llm agents. *arXiv preprint arXiv:2608.01739*, 2026.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21457–21483, 2026.

Danyang Zhang, Lu Chen, Situo Zhang, Hongshen Xu, Zihan Zhao, and Kai Yu. Large language models are semi-parametric reinforcement learning agents. *Advances in Neural Information Processing Systems*, 36:78227–78239, 2023.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025a.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, et al. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025b.

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.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, et al. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*, 2025.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

Appendix

Table of Contents

A Notation **15**

B Reward and attempts **15**

B.1 Other success and failure outcomes 15

B.2 Two attempts on the same query 16

B.3 Attempts during runtime learning 16

C Implementation details **16**

C.1 Runtime learning 16

C.2 Models and decoding 16

C.3 Hyperparameters 17

C.4 Data splits 18

C.5 Baselines and benchmarks 18

D Additional experimental results **18**

D.1 Re-attempt after success and outcome changes 18

D.2 Weight on memory advantage on OS 19

D.3 Additional backbones 21

D.4 Memory bank size and attempt cost 21

D.5 Re-attempts after success 22

E Memory-bank analysis **22**

E.1 Improvement and degradation by retrieved utility 22

E.2 Distinct sequences 23

F Prompts **24**

F.1 Shared experience summarization 25

F.2 Retrieved-context formatting 25

F.3 Generation and action interfaces 26

G Limitations **28**

A NOTATION

Table 6 lists the symbols used in the main text and this appendix. Hats denote z-score normalization within the Phase-A pool $\mathcal{P}$. In Eq. (1), λ is the utility weight. In Eq. (3), η is the weight on memory advantage. The two weights are not interchangeable.

Table 6: **Notation.** Symbols used in the main text and appendix.

Symbol	Meaning
Query, outcome, and reward	
q	Query; $f_{\text{emb}}(q)$ is its embedding
f_{llm}, f_{emb}, f_{sum}, f_{env}	Frozen LLM backbone, embedder, summarizer, and environment
a	Action, $a \sim f_{\text{llm}}(\cdot \mid q, \mathcal{C})$
$a_\varnothing$	Memory-free action, $a_\varnothing \sim f_{\text{llm}}(\cdot \mid q)$
y	Outcome $y = f_{\text{env}}(q, a)$, positive on success
$y_\varnothing$	Memory-free outcome $y_\varnothing = f_{\text{env}}(q, a_\varnothing)$
r	Reward that updates utility; $r = y - \eta\, y_\varnothing$ (Eq. (3))
η	Weight on memory advantage; $\eta = 0$ gives $r = y$
μ	Retriever in Section 3.1; $\mathcal{C} = \mu(q, \mathcal{M})$
Memory and two-phase retrieval	
$\mathcal{M}$, m_i	Memory bank and item $m_i = (z_i, e_i, Q_i)$
z_i, e_i, Q_i	Stored query embedding, experience, and learned utility; a new memory is written with $z = f_{\text{emb}}(q)$
Q_{init}	Initial utility of a new memory
ρ_i	Cosine similarity between $f_{\text{emb}}(q)$ and z_i
$\mathcal{P}$	Phase-A pool: memories with $\rho_i > \delta$, the k_1 most similar
$\mathcal{C}$	Context $\mathcal{C} \subseteq \mathcal{P}$: the k_2 candidates with the highest ψ_i
δ, k_1, k_2	Similarity threshold, Phase-A pool size, Phase-B context size
ψ_i, λ	Phase-B score (Eq. (1)) and its utility weight
α	Step size in Eq. (2)
Vantage	
$y^{(1)}$, $y^{(2)}$	Outcomes of the first attempt and the re-attempt on query q (Section 3.3)
d	Maximum number of re-attempts after a failure; $d = 1$ is Vantage
Evaluation	
SR	Fraction of queries solved in an epoch; a query counts as solved if any attempt in that epoch succeeds, including the re-attempt
LSR	SR of the last epoch, reported for reference
CSR	Fraction of queries solved at least once during runtime learning
Frozen SR	SR on the training queries with the bank fixed and one attempt per query
Held-out SR	SR on the held-out queries in that same frozen-bank evaluation
P	A query's success rate over ten memory-free attempts

B REWARD AND ATTEMPTS

This section records how the four rewards after Eq. (3) change when the failure outcome is -1 rather than 0, then the comparison between two attempts on the same query, and the attempt rule in Section 3.3.

B.1 OTHER SUCCESS AND FAILURE OUTCOMES

On OS and DB the outcome is 1 on success and 0 on failure, so the four rewards after Eq. (3) are 1, $1 - \eta$, 0, and $-\eta$. On BigCodeBench and ALFWorld the outcome is 1 on success and -1 on failure (Table 8). For the failure outcome -1, the same four cases give $1 + \eta$, $1 - \eta$, $-1 + \eta$, and $-1 - \eta$. Setting $\eta = 0$ yields $r = y$, with no memory-free attempt. Setting $\eta = 1$ uses only the outcome change. Q_{init} is 0 when the failure outcome is -1, and 0.5 when it is 0.

B.2 Two attempts on the same query

Both attempts share the memory-free outcome $y_{\varnothing}$ of the same query, so $y_{\varnothing}$ cancels in Eq. (3) and the rewards of the two attempts differ by $y^{(2)} - y^{(1)}$ for every η (Section 3.3). Vantage runs the second attempt only after the first fails. With $\eta = 0$ the reward is the outcome, and the first attempt is the reference. Vantage$^{\Delta}$ and MemRL$^{\Delta}$ use Eq. (3) at $\eta = 0.5$. The memory-free attempt used to obtain $y_{\varnothing}$ is not counted among the attempts.

B.3 Attempts during runtime learning

Under Vantage ($d = 1$), an epoch takes one attempt per query plus one re-attempt per failed first attempt, so a query is attempted at most twice in an epoch (Section 3.3). For $\eta = 0.5$, the memory-free attempt used to obtain $y_{\varnothing}$ is not counted. MemRL and MemRL$^{\Delta}$ use 10 epochs and attempt each query once per epoch, so each query is attempted 10 times. The settings that re-attempt after success use 5 epochs and also re-attempt after a successful first attempt, so each query is attempted 10 times. Vantage and Vantage$^{\Delta}$ use 5 epochs and do not re-attempt after a first-attempt success ($y^{(1)} > 0$). An epoch takes one attempt per query plus one re-attempt per failed first attempt, so its number of attempts decreases as more first attempts succeed. Figure 3 uses the finished 5-epoch budgets at $d = 3$: 3,676 attempts on OS and 2,169 on DB. Here d is the maximum number of re-attempts after failure, as in Section 4.3. The runs at $d = 1$ and $d = 2$ are cut at the same total.

C Implementation details

Unless noted, Vantage, Vantage$^{\Delta}$, MemRL, and MemRL$^{\Delta}$ share the frozen backbone, the embedding model, the two-phase retrieval, the summarizer, and the decoding configuration. Vantage differs from MemRL in the asymmetric re-attempt: only failed queries are re-attempted, and the first attempt is the reference (Algorithm 1). It sets $\eta = 0$, so the reward is the outcome and no memory-free attempt is used. MemRL$^{\Delta}$ keeps one attempt per epoch and sets $\eta = 0.5$ in Eq. (3). Vantage$^{\Delta}$ uses the same attempts as Vantage and the same $\eta = 0.5$. Both $\eta = 0.5$ settings run a memory-free attempt on every query to obtain $y_{\varnothing}$; that attempt is not counted among the attempts. All methods follow the splits of MemRL (Zhang et al., 2026). Prompt templates are in Appendix F.

C.1 Runtime learning

Algorithm 1 records Vantage on one query in one epoch ($d = 1$). The same rule applies to every query in every epoch. Each executed attempt uses two-phase retrieval, updates the utilities of the memories shown to f_{llm} with $r = y$, and writes the attempt with f_{sum}. The re-attempt, when taken, retrieves from the bank after the first update and write. Before each re-attempt, we restore the query's initial environment state and start a fresh interaction, carrying over only the updated memory bank from the previous attempt. Vantage$^{\Delta}$ uses the same control flow and replaces y in each update with $r = y - \eta\, y_{\varnothing}$ at $\eta = 0.5$.

MemRL attempts each query once per epoch and writes once per attempt. Vantage allows a second attempt on the same query in the same epoch only after failure (Figure 2). 5 epochs of Vantage and 10 epochs of MemRL can each make at most 10 attempts on a query; Vantage makes fewer because it does not re-attempt after success. The main comparisons report 5-epoch Vantage against 10-epoch MemRL and MemRL$^{\Delta}$; 10-epoch Vantage is reported separately. MemRL$^{\Delta}$ keeps one attempt per epoch. Both $\eta = 0.5$ settings additionally run a memory-free attempt on every query to obtain $y_{\varnothing}$; that attempt is not counted among the attempts.

C.2 Models and decoding

Table 7 lists the models and decoding choices used in the main experiments.

Algorithm 1 Vantage on one query in one epoch ($d = 1$).

Require: Query q, memory bank $\mathcal{M}$, step size α, initial utility Q_{init}.
$\mathcal{C} \leftarrow \mu(q, \mathcal{M})$ Phase A then Phase B
Produce $a \sim f_{\text{llm}}(\cdot \mid q, \mathcal{C})$ and receive $y \leftarrow f_{\text{env}}(q, a)$ greedy
for all $m_i \in \mathcal{C}$ **do**
  $Q_i \leftarrow Q_i + \alpha(y - Q_i)$ $r = y$ since $\eta = 0$
end for
$e \leftarrow f_{\text{sum}}(q, a, y)$; $z \leftarrow f_{\text{emb}}(q)$; $\mathcal{M} \leftarrow \mathcal{M} \cup \{(z,\, e,\, Q_{\text{init}})\}$
if $y > 0$ **then**
  return $\mathcal{M}$ do not re-attempt
end if
$\mathcal{C} \leftarrow \mu(q, \mathcal{M})$ retrieve from the updated bank
Produce $a \sim f_{\text{llm}}(\cdot \mid q, \mathcal{C})$ and receive $y \leftarrow f_{\text{env}}(q, a)$ greedy
for all $m_i \in \mathcal{C}$ **do**
  $Q_i \leftarrow Q_i + \alpha(y - Q_i)$
end for
$e \leftarrow f_{\text{sum}}(q, a, y)$; $z \leftarrow f_{\text{emb}}(q)$; $\mathcal{M} \leftarrow \mathcal{M} \cup \{(z,\, e,\, Q_{\text{init}})\}$
return $\mathcal{M}$

Table 7: **Models and decoding.** Values for the main experiments.

Component	Value	Note
Backbone	Qwen3-8B	Frozen
Embedder	Qwen3-Embedding 0.6B	A new memory stores $z = f_{\text{emb}}(q)$
Temperature	0.0	Runtime learning and Frozen-bank evaluation; greedy
Pass@10 temperature	0.6	No-memory CSR: 10 attempts
k_1 / k_2	10 / 5	Planning uses 5 / 3
Max steps	15 / 30 / 1	OS and DB / ALFWorld / BigCodeBench

C.3 Hyperparameters

Table 8 lists retrieval and utility hyperparameters. Hats in the Phase-B score denote z-score normalization within the Phase-A pool. If the standard deviation of a quantity in $\mathcal{P}$ is zero, that z-score is set to 0. The weight η applies to MemRL$^{\Delta}$ and Vantage$^{\Delta}$; Vantage uses $\eta = 0$, so the reward is the outcome. The listed $\eta = 0.5$ is the default used in the four-domain tables, not a claim that this value is best for every method or metric.

Table 8: **Hyperparameters.** Retrieval and utility hyperparameters across benchmarks. α is the utility step size; λ is the Phase-B utility weight in Eq. (1); η is the weight on memory advantage in Eq. (3) and applies to MemRL$^{\Delta}$ and Vantage$^{\Delta}$. δ is the similarity threshold on cosine similarity. MemRL and MemRL$^{\Delta}$ use 10 epochs; the main Vantage comparison uses 5 epochs. 10-epoch Vantage is reported separately in Tables 2, 3, and 14.

Parameter	Code	OS	DB	Planning
α	0.3	0.3	0.3	0.3
λ	0.5	0.5	0.5	0.5
δ	0.5	0.5	0.5	0.62
k_1	10	10	10	5
k_2	5	5	5	3
Q_{init}	0.0	0.5	0.5	0.0
failure outcome	-1	0	0	-1
η (MemRL$^{\Delta}$, Vantage$^{\Delta}$)	0.5	0.5	0.5	0.5
Epochs (MemRL / Vantage)	10 / 5	10 / 5	10 / 5	10 / 5

k_1 and k_2 are MemRL's Phase-A pool size and Phase-B context size. Vantage uses the same retriever.

C.4 Data splits

Table 9 records the number of training queries and held-out queries. Runtime learning passes over the training queries in epochs, with utility updates and memory writing enabled. Frozen-bank evaluation reuses the training queries with the bank held fixed and one attempt per query. Frozen-bank evaluation on held-out queries uses queries that do not appear in runtime learning, with one attempt per query. The re-attempt after failure is used only during runtime learning.

Table 9: **Data splits.** Training-query and held-out sizes. Sizes and the split procedure follow MemRL (Zhang et al., 2026).

Benchmark	Training queries	Held-out	Split / note
BigCodeBench (Code)	798	342	Instruct split
LifelongAgentBench (OS)	350	150	Official OS split
LifelongAgentBench (DB)	361	139	Official DB split
ALFWorld (Planning)	3553	140	Full split

C.5 Baselines and benchmarks

Backbone without memory. The runtime table also records the frozen backbone with no retrieval, writing, or utility updates. The number before the slash is the SR of one attempt, and Pass@10 is recorded after the slash. Pass@10 uses temperature 0.6. Runtime learning and frozen-bank evaluation use temperature 0.0.

Baselines. **Mem0** (Chhikara et al., 2025) maintains a memory store of past experience. **MemP** (Fang et al., 2026) distills successful trajectories into procedural workflows. **MemRL** (Zhang et al., 2026) provides the two-phase retrieval and utility update that Vantage uses, with one attempt per epoch and $r = y$. **MemRL$^{\Delta}$** uses Eq. (3) at $\eta = 0.5$, and runs a memory-free attempt on every query to obtain $y_{\varnothing}$; that attempt is not counted among the attempts.

Vantage. **Vantage** is Algorithm 1 ($d = 1$). **Vantage$^{\Delta}$** uses the same attempts and sets $\eta = 0.5$, which likewise requires a memory-free attempt not counted among the attempts. MemRL and Vantage use the same summarizer.

Benchmarks. **BigCodeBench** (Zhuo et al., 2025) evaluates library-oriented code generation; the Instruct split follows MemRL. **LifelongAgentBench** (Zheng et al., 2025) provides interactive OS and DB tasks, reported as separate domains. **ALFWorld** (Shridhar et al., 2020) is an embodied household planning benchmark. Together the four domains cover coding, computer use, and embodied planning. The success outcome is 1. The failure outcome is 0 on OS and DB and -1 on BigCodeBench and ALFWorld. LLM-as-a-judge labels are not used.

D Additional experimental results

This section reports four-domain runtime-learning, Frozen SR, and Held-out SR results for the 5-epoch settings that re-attempt after success and the 5-epoch settings that do not, at $\eta = 0$ and $\eta = 0.5$, together with MemRL and MemRL$^{\Delta}$ after 10 epochs; the OS sweep of η; a second backbone; bank size versus attempt count; and re-attempts after a successful first attempt.

D.1 Re-attempt after success and outcome changes

Tables 10 and 11 report the comparison of Table 5 on all four domains, with MemRL and MemRL$^{\Delta}$ after 10 epochs as baselines. The settings cross the same two choices: whether a successful query is also re-attempted in the same epoch, and whether $\eta = 0$ or $\eta = 0.5$. Settings with $\eta = 0$ use $r = y$ and make no memory-free attempt. Settings with $\eta = 0.5$ use Eq. (3) and run a memory-free attempt on every query to obtain $y_{\varnothing}$; that attempt is not counted among the attempts. Vantage does not re-attempt after a first-attempt success ($y^{(1)} > 0$); this is the 5-epoch Vantage in Tables 2 and 3. Vantage$^{\Delta}$ uses the same attempts at $\eta = 0.5$.

Table 10: **Runtime learning, four domains.** LSR / CSR (%). MemRL and MemRL$^{\Delta}$ are baselines after 10 epochs; the other settings use 5 epochs: re-attempt after success at $\eta = 0$ and at $\eta = 0.5$, and Vantage and Vantage$^{\Delta}$, which do not re-attempt after success. Averages are unweighted means across the four task domains. MemRL$^{\Delta}$ and Vantage$^{\Delta}$ use $\eta = 0.5$. Best and second-best CSR in each column are in **bold** and underlined. LSR is shown for reference and is not ranked.

	BigCodeBench	**LifelongAgentBench**		**ALFWorld**	**Average**
Method	Code	OS	DB	Planning	LSR / CSR
MemRL	54.5 / 57.5	57.4 / 65.7	94.7 / 95.0	82.5 / 94.2	72.3 / 78.1
MemRL$^{\Delta}$	57.5 / 63.8	58.3 / 68.3	94.7 / 95.0	87.0 / 94.6	74.4 / 80.4
re-attempt after success, $\eta = 0$	61.2 / 61.9	60.3 / 67.1	94.7 / 94.7	88.5 / 94.4	76.2 / 79.5
re-attempt after success, $\eta = 0.5$	61.2 / 63.7	63.1 / 69.1	94.5 / 95.0	91.5 / 94.3	77.6 / 80.5
Vantage	61.9 / **64.2**	64.3 / 69.1	95.6 / **95.8**	93.4 / 94.9	78.8 / 81.0
Vantage$^{\Delta}$	61.5 / 62.5	66.6 / **71.4**	95.0 / 95.3	94.6 / **95.3**	79.4 / **81.1**

Table 11: **Frozen-bank evaluation, four domains.** Each cell reports Frozen SR / Held-out SR (%), using one attempt per query with bank contents and utilities fixed. MemRL and MemRL$^{\Delta}$ are banks after 10 epochs; the other settings are banks after 5 epochs. Averages are unweighted means across the four task domains. MemRL$^{\Delta}$ and Vantage$^{\Delta}$ use $\eta = 0.5$. Best in each column in **bold**, second underlined.

	BigCodeBench	**LifelongAgentBench**		**ALFWorld**	**Average**
Method	Code	OS	DB	Planning	Frozen SR / Held-out SR
MemRL	53.4 / 41.8	57.1 / 53.3	95.0 / 85.6	84.6 / 79.3	72.5 / 65.0
MemRL$^{\Delta}$	57.5 / 42.1	58.9 / 54.0	94.7 / 85.6	86.8 / 81.4	74.5 / 65.8
re-attempt after success, $\eta = 0$	59.5 / 40.4	59.7 / 54.0	94.7 / 84.9	89.0 / 77.9	75.7 / 64.3
re-attempt after success, $\eta = 0.5$	61.0 / 40.9	63.1 / 59.3	95.0 / 84.9	88.5 / 77.9	76.9 / 65.8
Vantage	**62.3 / 43.0**	64.3 / 56.0	**95.3 / 87.8**	**93.7 / 82.1**	78.9 / **67.2**
Vantage$^{\Delta}$	61.3 / 41.8	**66.3 / 60.0**	**95.3** / 85.6	93.4 / 80.7	**79.1** / 67.0

Under the asymmetric re-attempt, adding the memory-free reference barely changes the average: Vantage$^{\Delta}$ reaches 79.1 Frozen SR and 67.0 Held-out SR, against 78.9 and 67.2 for Vantage. When successful queries are also re-attempted, $\eta = 0.5$ raises average Frozen SR from 75.7 to 76.9 and Held-out SR from 64.3 to 65.8. Not re-attempting those queries raises the same pair from 75.7/64.3 to 78.9/67.2 and from 76.9/65.8 to 79.1/67.0; the same holds on every domain. Vantage reaches that result without a memory-free attempt, and with 40% fewer attempts and a 40% lighter memory bank than 10-epoch MemRL (Table 14). The LSR values are reference only: the last epoch may contain two attempts, so they are not the comparison with 10-epoch MemRL or MemRL$^{\Delta}$.

Figures 7 and 8 plot the same four 5-epoch settings. The top row is the SR of each epoch and the bottom row is CSR.

D.2 Weight on memory advantage on OS

Table 12 reports the η sweep of Table 4 on OS together with the same grid under the asymmetric re-attempt. The left block uses one attempt per epoch for 10 epochs. The right block uses 5 epochs and re-attempts only after failure. For $\eta > 0$, including Vantage$^{\Delta}$, each update uses Eq. (3) and a memory-free attempt to obtain $y_{\varnothing}$. At $\eta = 0$ the reward is the outcome, $r = y$: the left block matches MemRL on OS in Tables 2 and 3 (57.4 / 65.7, Frozen SR 57.1, Held-out SR 53.3), and the right block matches Vantage on OS in Tables 10 and 11 (64.3 / 69.1, Frozen SR 64.3, Held-out SR 56.0). Only the right block at $\eta = 0$ is Vantage: the first attempt is the reference, and no memory-free attempt is used. At $\eta = 1$ the reward is the outcome change alone. MemRL$^{\Delta}$ and Vantage$^{\Delta}$ are the entries at $\eta = 0.5$, not the whole sweep.

With one attempt per epoch, the leading η depends on the metric, as in Table 4: $\eta = 0.25$ leads LSR and Frozen SR, $\eta = 0.25$ and $\eta = 0.5$ tie on CSR, and $\eta = 0.75$ leads Held-out SR. At $\eta = 0.75$, LSR, CSR, Frozen SR, and Held-out SR are all higher than at $\eta = 0$ and at $\eta = 1$, including Held-out SR (56.0, versus 53.3 at $\eta = 0$ and 54.7 at $\eta = 1$). The four-domain MemRL$^{\Delta}$ results use

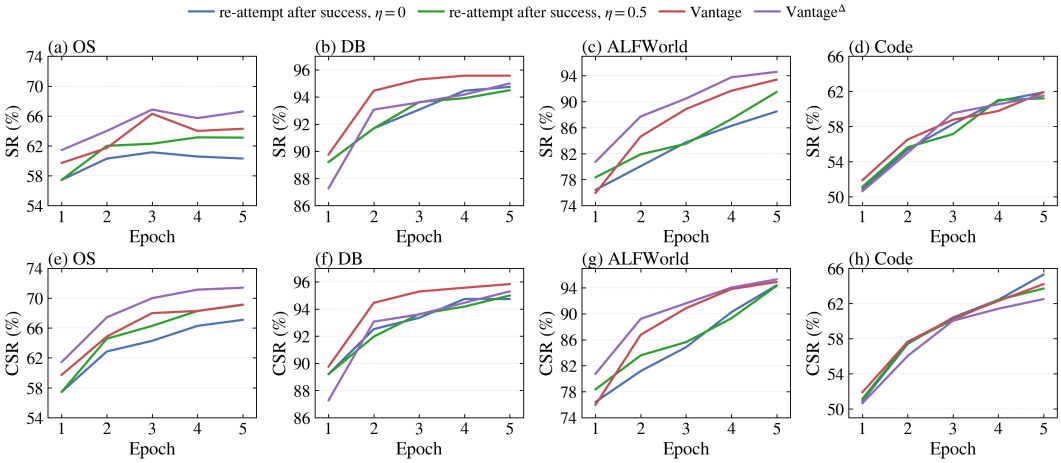

Figure 7: SR and CSR over 5 epochs on OS, DB, ALFWorld, and Code. The four settings are re-attempt after success at $\eta = 0$ and at $\eta = 0.5$, Vantage, and Vantage$^{\Delta}$.

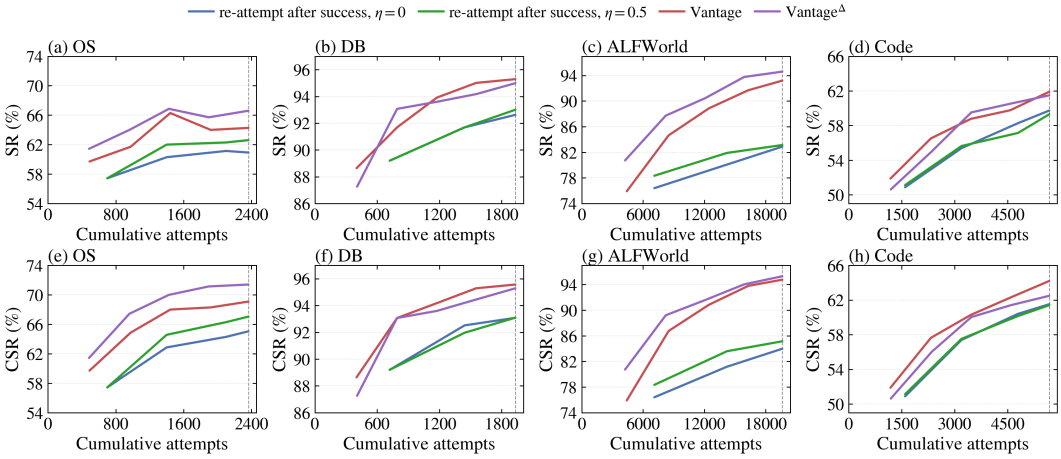

Figure 8: The same four settings, plotted against the cumulative number of attempts. The count includes the re-attempt after failure and leaves out the memory-free attempt used to obtain $y_{\varnothing}$. Each panel ends at the smallest 5-epoch attempt count among the four settings.

$\eta = 0.5$. On OS, $\eta = 0.5$ with one attempt per epoch improves all four metrics over $\eta = 0$, and it is not the leading value on LSR, Frozen SR, or Held-out SR.

Table 12: **Weight η on OS.** LSR / CSR, Frozen SR, and Held-out SR (%). The left block uses 10 epochs and one attempt per epoch; the right block uses 5 epochs and does not re-attempt after success. $\eta = 0$ gives $r = y$; $\eta = 1$ uses only the outcome change. MemRL$^{\Delta}$ and Vantage$^{\Delta}$ are the entries at $\eta = 0.5$. Best within each block in **bold**; a tie is marked on each tied entry.

	One attempt per epoch (10 epochs)			**No re-attempt after success (5 epochs)**		
η	LSR / CSR	Frozen	Held-out	LSR / CSR	Frozen	Held-out
0	57.4 / 65.7	57.1	53.3	64.3 / 69.1	64.3	56.0
0.25	**59.4 / 68.3**	**59.1**	53.3	64.0 / 71.1	63.4	55.3
0.5	58.3 / **68.3**	58.9	54.0	**66.6 / 71.4**	**66.3**	**60.0**
0.75	59.1 / 66.9	58.9	**56.0**	63.4 / 70.6	63.1	56.0
1	58.3 / 66.0	58.3	54.7	63.1 / 70.0	63.1	54.7

D.3 Additional backbones

The main experiments use a frozen Qwen3-8B model. Table 13 reports MemRL, MemRL$^{\Delta}$ ($\eta = 0.5$), and Vantage with a frozen Qwen2.5-7B-Instruct backbone, the same two-phase retriever, and the same summarizer f_{sum}. MemRL and MemRL$^{\Delta}$ use 10 epochs; Vantage uses 5 epochs. Held-out SR is one attempt per held-out query, with the bank fixed and writing and utility updates disabled. Frozen SR is the SR on the training queries in Frozen-bank evaluation: the bank is fixed from the last epoch and each query is attempted once, with writing and utility updates disabled. Vantage has a higher CSR, Frozen SR, and Held-out SR than MemRL and MemRL$^{\Delta}$ on every domain. LSR is reported for reference and is not the comparison with 10-epoch MemRL or MemRL$^{\Delta}$. Vantage's unweighted averages are 75.1 CSR, 71.8 Frozen SR, and 61.7 Held-out SR. MemRL's are 71.5, 64.7, and 58.3; MemRL$^{\Delta}$'s are 73.0, 66.3, and 59.2.

Table 13: **Qwen2.5-7B-Instruct.** LSR / CSR, Frozen SR, and Held-out SR (%). MemRL and MemRL$^{\Delta}$ use 10 epochs; Vantage uses 5 epochs. Averages are unweighted means over the four domains.

Method	Code	OS	DB	Planning	Average
LSR / CSR					
MemRL (10 epochs)	49.1 / 53.4	54.0 / 59.4	90.0 / 90.3	65.5 / 82.8	64.7 / 71.5
MemRL$^{\Delta}$ (10 epochs)	51.0 / 54.8	54.3 / 62.9	90.0 / 90.3	69.2 / 84.1	66.1 / 73.0
Vantage (5 epochs)	51.5 / 55.8	59.4 / 63.7	92.0 / 92.8	84.5 / 87.9	71.9 / 75.1
Frozen SR					
MemRL (10 epochs)	48.9	54.3	90.0	65.6	64.7
MemRL$^{\Delta}$ (10 epochs)	51.2	54.3	90.3	69.5	66.3
Vantage (5 epochs)	51.6	59.4	92.0	84.3	71.8
Held-out SR					
MemRL (10 epochs)	31.9	49.3	79.9	72.1	58.3
MemRL$^{\Delta}$ (10 epochs)	33.9	50.0	80.6	72.1	59.2
Vantage (5 epochs)	35.1	53.3	82.0	76.4	61.7

D.4 Memory bank size and attempt cost

Vantage writes at most twice on a query in an epoch and does not re-attempt after a first-attempt success. Each with-memory attempt writes once, so the two counts in Table 14 coincide. Vantage after 5 epochs therefore stores fewer memories than MemRL or MemRL$^{\Delta}$ after 10 epochs, and fewer than the 5-epoch setting that also re-attempts after success. Vantage after 10 epochs can exceed those banks: a query may still be re-attempted after failure, so with 10 epochs a query can be attempted 20 times. In Table 14 it exceeds them on every domain.

Table 14: **Bank size and attempts.** Stored memories / with-memory attempts. The second line of 5-epoch Vantage is the reduction versus 10-epoch MemRL. Memory-free attempts are not counted.

Method	Code	OS	DB	Planning	Average
MemRL (10 epochs)	7980 / 7980	3500 / 3500	3610 / 3610	35530 / 35530	12655 / 12655
MemRL$^{\Delta}$ (10 epochs)	7980 / 7980	3500 / 3500	3610 / 3610	35530 / 35530	12655 / 12655
re-attempt after success (5 epochs)	7980 / 7980	3500 / 3500	3610 / 3610	35530 / 35530	12655 / 12655
Vantage (5 epochs)	5668 / 5668 (29%/29%)	2473 / 2473 (29%/29%)	1953 / 1953 (46%/46%)	20091 / 20091 (43%/43%)	7546 / 7546 (40%/40%)
Vantage (10 epochs)	11132 / 11132	4925 / 4925	3826 / 3826	38911 / 38911	14699 / 14699

The second-line percentages for Vantage after 5 epochs are reductions relative to MemRL after 10 epochs on the same domain, and apply equally to attempts and stored memories. Summing the four domains gives 30,185 versus 50,620, a 40% reduction. The 40% figure is the reduction of the summed attempt counts.

D.5 Re-attempts after success

The setting that re-attempts after success does so even when $y^{(1)} > 0$: it retrieves again from the updated bank, acts, updates, and writes. Vantage and Vantage$^{\Delta}$ do not make that re-attempt. Table 15 reports, for re-attempt after success at $\eta = 0$, how often the re-attempt also succeeds, $\Pr(y^{(2)} > 0 \mid y^{(1)} > 0)$. A re-attempt after a successful first attempt almost always succeeds again, in at least 98.3% of cases on every domain: 98.3% on Code, 98.6% on OS, 99.6% on DB, and 98.3% on Planning, because the memories credited by the first attempt are ranked higher and largely retrieved again (Section 3.3). Such a re-attempt is unnecessary: as an unchanged success it measures no outcome change, and it would only credit those memories a second time and write a similar memory.

Table 15: **Re-attempts after success** on the 5-epoch setting that also re-attempts after a successful first attempt ($\eta = 0$). The share that also succeed is $\Pr(y^{(2)} > 0 \mid y^{(1)} > 0)$. Entries are percentages except the counts of such re-attempts. Vantage does not execute them.

	Code	**OS**	**DB**	**Planning**
Re-attempts after success	1611	897	1510	13426
Also succeeds (%)	98.3	98.6	99.6	98.3

E Memory-bank analysis

This section records the counts behind Figures 4 and 6. In Section 4.4, P is a query's success rate over ten memory-free attempts and $y \in \{0, 1\}$ is its outcome on one with-memory attempt: an improvement is $y - P > 0$, and a degradation is $y - P < 0$. The counts in Section E.1 use that rule.

E.1 Improvement and degradation by retrieved utility

Figure 4 groups with-memory attempts on OS by the final utility Q of the retrieved memory, and reports the share that are improvements rather than degradations. P is that query's success rate over 10 memory-free attempts. $y \in \{0, 1\}$ is the outcome on one with-memory attempt: an improvement is $y - P > 0$, and a degradation is $y - P < 0$. The remaining attempts have $y - P = 0$: a failure at $P = 0$, or a success at $P = 1$. The figure drops unchanged attempts, so each bar is the improvement share among improvements and degradations in that bin. The dashed line is that share pooled over the six plotted bins, Eq. (4), not an unweighted mean of the six bar heights. The measurement is whether this learned utility tracks memory advantage. The bins use the Q stored at the end of the run: the epoch-10 snapshot for MemRL and the epoch-5 snapshot for Vantage and Vantage$^{\Delta}$. Bins are $[0.4, 0.5)$, $[0.5, 0.6)$, $[0.6, 0.7)$, $[0.7, 0.8)$, $[0.8, 0.9)$, and $[0.9, \infty)$, so a final $Q > 1$ is counted in 0.9–1.0. Attempts with $Q < 0.4$, or with no retrieved memory, are omitted. On every domain the counted memory is the Top-1 retrieval, ranked by Q. MemRL contributes one retrieval per query per epoch. Vantage and Vantage$^{\Delta}$ contribute the first attempt and the re-attempt after failure. Memory-free attempts are for this analysis only; Vantage does not run them during runtime learning. On OS and DB the environment returns 0 on failure; on BigCodeBench and ALFWorld it returns -1. In the counts above, a success is recorded as $y = 1$ and a failure as $y = 0$.

Pooled shares. Write n_b^{imp} and n_b^{deg} for the improvement and degradation counts in bin b, read from Table 16 by dropping the middle, unchanged, entry of each cell:

$$\frac{\sum_b n_b^{\text{imp}}}{\sum_b \left(n_b^{\text{imp}} + n_b^{\text{deg}}\right)}. \tag{4}$$

On OS this is $371/482 = 77.0\%$ for MemRL, $306/350 = 87.4\%$ for Vantage, and $309/353 = 87.5\%$ for Vantage$^{\Delta}$; the first two are the dashed lines in Figure 4. The MemRL bin 0.9–1.0 is $152/170 = 89\%$ improvements. The Vantage bins 0.8–0.9 and 0.9–1.0 are $69/74 = 93\%$ and $85/91 = 93\%$. The same ratio on the other domains is $650/751 = 86.6\%$ / $464/534 = 86.9\%$ / $407/464 = 87.7\%$ on Code, $684/760 = 90.0\%$ / $425/466 = 91.2\%$ / $386/420 = 91.9\%$ on DB, and $5136/5618 = 91.4\%$ / $4150/4435 = 93.6\%$ / $4501/4881 = 92.2\%$ on ALFWorld (MemRL / Vantage / Vantage$^{\Delta}$). The pooled shares of Vantage and Vantage$^{\Delta}$ stay close on every domain.

Their unweighted average over the four domains is 89.8% for both, above MemRL's 86.3%. On OS, Code, and ALFWorld both exceed MemRL. On DB, Vantage is 91.2% and Vantage$^{\Delta}$ is 91.9%, against 90.0% for MemRL.

Table 16: **Improvement, unchanged, and degradation counts.** Each cell is improvement / unchanged / degradation. $Q < 0.4$ is omitted. MemRL uses the epoch-10 snapshot; Vantage and Vantage$^{\Delta}$ use the epoch-5 snapshot. OS rows with $Q \geq 0.4$ are those in Figure 4.

Final Q	0.4–0.5	0.5–0.6	0.6–0.7	0.7–0.8	0.8–0.9	0.9–1.0
OS						
MemRL (10 epochs)	74/183/51	23/101/11	50/122/11	33/85/9	39/130/11	152/514/18
Vantage (5 epochs)	64/145/20	24/22/5	31/55/4	33/65/4	69/114/5	85/259/6
Vantage$^{\Delta}$ (5 epochs)	71/146/17	99/297/13	39/16/4	31/6/4	52/10/4	17/1/2
DB						
MemRL (10 epochs)	6/33/16	12/36/3	14/63/5	40/72/5	28/72/6	584/2208/41
Vantage (5 epochs)	4/41/1	19/55/3	13/29/2	21/49/3	136/282/8	232/816/24
Vantage$^{\Delta}$ (5 epochs)	31/130/10	167/924/12	33/47/3	50/7/4	101/8/5	4/0/0
BigCodeBench						
MemRL (10 epochs)	59/216/9	40/195/9	67/182/5	60/186/11	159/477/20	265/1065/47
Vantage (5 epochs)	53/251/18	81/186/8	66/201/10	78/277/12	91/235/10	95/246/12
Vantage$^{\Delta}$ (5 epochs)	42/282/11	35/87/7	41/117/8	61/110/5	61/74/4	167/172/22
ALFWorld						
MemRL (10 epochs)	38/56/13	156/210/12	227/99/6	1330/661/133	1323/620/136	2062/2818/182
Vantage (5 epochs)	367/235/61	209/410/47	1321/711/39	1062/664/49	309/1337/53	882/1377/36
Vantage$^{\Delta}$ (5 epochs)	956/1736/130	661/687/57	934/609/96	744/497/39	289/213/42	917/322/16

Figure 9 uses the share of improvements in each bin of final Top-1 utility on OS, the same share as the bars in Figure 4. The left bar is the absolute difference, in percentage points, between that share under Vantage and under Vantage$^{\Delta}$. The right bar is the absolute difference between that share under Vantage and under MemRL. The left bar is smaller in every bin: 4.5 against 17.0, 5.6 against 15.1, 2.1 against 6.6, 0.6 against 10.6, 0.4 against 15.2, and 3.9 against 4.0 in 0.9–1.0. The mean of the left bars is 2.9 and the mean of the right bars is 11.4. Vantage and Vantage$^{\Delta}$ therefore stay closer than Vantage and MemRL on the share of improvements across final Q.

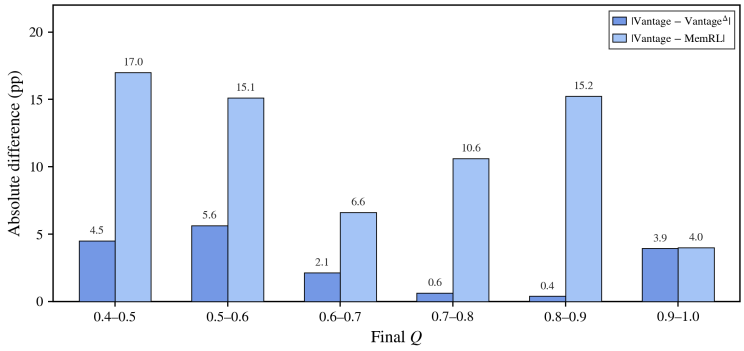

Figure 9: **Absolute difference of retrospective improvement shares on OS.** Bins are final Top-1 utility, as in Figure 4. Left: absolute difference, in percentage points, between the share of improvements under Vantage and under Vantage$^{\Delta}$. Right: absolute difference between the share of improvements under Vantage and under MemRL.

E.2 Distinct sequences

Not re-attempting after success raises the ratio of distinct command or SQL sequences to writes. This subsection also counts how many distinct command or SQL sequences those writes contain. To isolate that choice, Vantage is compared with the 5-epoch setting that also re-attempts after success. Both use 5 epochs and the same re-attempt after failure; they differ only in whether a successful first attempt is followed by a re-attempt in the same epoch. The analysis is restricted to OS and DB, whose trajectories record the executed shell commands and SQL statements. The counts use those statements, not the text of stored summaries.

Measurement. For each write, extract the sequence of statements the environment executed: shell commands on OS and SQL statements on DB. Each statement is normalized to Unicode NFKC, trimmed, and has internal whitespace collapsed. Prompt text, fixed demonstrations, comments, placeholder lines, observations, utility values, and the stored summary are excluded. The same extraction is used for every method. Writes with an empty sequence are discarded.

Two writes are duplicates if and only if they have the same source query, the same outcome, and identical sequences. A difference in statement order is sufficient for the writes not to be duplicates.

Let n be the number of remaining writes. For each source query, count the distinct command or SQL sequences among writes with a success and among writes with a failure. Let n_{seq} be the sum of those two counts over source queries. The reported ratio is n_{seq}/n. Tables and figures report $100\, n_{\text{seq}}/n$.

Not re-attempting after success raises the ratio of distinct sequences to writes. Because the ratio depends on both counts, Table 17 reports n_{seq} and n separately. Relative to the 5-epoch setting that re-attempts after success, Vantage raises $100\, n_{\text{seq}}/n$ from 59.94 to 73.14 on OS ($+13.20$ points) and from 17.76 to 32.80 on DB ($+15.04$ points).

Table 17: **Distinct sequences and writes.** Success and failure are the numbers of distinct sequences among writes with a success and writes with a failure, summed over source queries. n_{seq} is their sum and n is the number of writes with a nonempty command or SQL sequence. The last column is $100\, n_{\text{seq}}/n$. MemRL$^{\Delta}$ uses $\eta = 0.5$ on OS and DB.

	Method	Epochs	Success	Failure	n_{seq}	n	$100\, n_{\text{seq}}/n$
OS	MemRL	10	704	1320	2024	3220	62.86
	MemRL$^{\Delta}$	10	633	1169	1802	3196	56.38
	re-attempt after success	5	647	1270	1917	3198	59.94
	Vantage	5	456	1148	1604	2193	73.14
DB	MemRL	10	409	190	599	3489	17.17
	MemRL$^{\Delta}$	10	414	158	572	3486	16.41
	re-attempt after success	5	414	205	619	3486	17.76
	Vantage	5	407	193	600	1829	32.80

The 5-epoch setting that re-attempts after success writes after either first-attempt outcome, so its attempt count matches MemRL after 10 epochs. Its ratio remains close to MemRL after 10 epochs on both domains (59.94 versus 62.86 on OS and 17.76 versus 17.17 on DB). Vantage and that setting share 5 epochs and the re-attempt after failure. A re-attempt after success writes another memory for a query that already succeeded, often with the same sequence. MemRL$^{\Delta}$ keeps one attempt per epoch and does not reproduce the Vantage increase on either domain.

The within-domain comparison is the relevant one: Vantage exceeds the setting that re-attempts after success on both domains. The ratio is computed from command or SQL sequences, not from the text of stored summaries, and does not attribute outcomes to individual memories.

F Prompts

Write, retrieve, and generation templates follow the prompt family of MemRL (Zhang et al., 2026). Vantage and Vantage$^{\Delta}$ inherit that summarizer and two-phase retrieval; which attempts are executed does not change the prompts. What changes across benchmarks is the retrieved-context wrapper and the action interface. The blocks below are the strings sent to the backbone, not logs of runtime learning. The message-order lists are the schematic traces of each domain's interaction loop. Outcomes come from the task-specific evaluator (unit tests on BigCodeBench; environment success flags on OS, DB, and ALFWorld), not from an auxiliary judge.

F.1 SHARED EXPERIENCE SUMMARIZATION

Successful and failed trajectories use the same prompt family. The outcome y comes from the task-specific evaluator, not from the summarizer. The high-level script and failure-reflection prompts are shared; only the serialized trajectory is benchmark-specific.

Shared write prompts

High-level script (success).

```
Analyze the following detailed task trajectory and create a concise,
high-level script that captures the essential steps and decision
points.

The script should be:
1. Generic enough to apply to similar tasks
2. Specific enough to provide useful guidance
3. 3-5 high-level steps maximum
4. Focus on the strategy and key decisions, not detailed actions

Trajectory:
{trajectory}

High-level script:
```

Failure reflection.

```
Task: {task_description}

Failed trajectory:
{failed_trajectory}

This task failed. Analyze what went wrong and suggest improvements
for future similar tasks.
Focus on:
1. Incorrect assumptions
2. Steps to improve
3. What to avoid next time

Provide a brief reflection:
```

Stored memory templates. Success stores the first block; failure stores the second.

```
SCRIPT:
{script}

TRAJECTORY:
{trajectory}

TASK REFLECTION:
Task: {task_description}

What went wrong:
{reflection}

Failed approach:
{failed_trajectory}
```

F.2 RETRIEVED-CONTEXT FORMATTING

Phase-B returns the k_2 candidates with the highest ψ_i. They are serialized and prepended as a system block. BigCodeBench uses a numbered code-example list; OS/DB split success and failure memories; ALFWorld injects memories after the few-shot dialogue. On OS/DB, the stored `Failed approach` tail is stripped before injection.

BigCodeBench: retrieved context

```
# Relevant Code Examples from Memory

## Example 1 [SUCCESS]
Task: {task_id}
[MEMORY TYPE] SUCCESS_PROCEDURE
[TASK]
{task_description}

[EXECUTION TRAJECTORY]
{script_and_trajectory}

## Example 2 [FAILURE]
Task: {task_id}
[MEMORY TYPE] FAILURE_REFLECTION
[TASK]
{task_description}

[REFLECTION]
{reflection}
```

OS / DB: retrieved context

```
[Retrieved Memory Context]

=== SUCCESSFUL EXPERIENCES (Learn from these) ===
[SUCCESS 1] [TYPE: {mem_type}]
{content}

=== FAILED EXPERIENCES (Avoid these mistakes) ===
[FAILURE 1] [TYPE: {mem_type}]
{content}
```

ALFWorld: retrieved context

```
In addition to the example, you have the following memories from your
own past experiences. Use them to help you if they are relevant:

--- SUCCESSFUL MEMORIES (Examples to follow) ---
{formatted_success_memories}

--- FAILED MEMORIES (Examples to avoid or learn from) ---
{formatted_failed_memories}
```

F.3 Generation and action interfaces

Each domain has its own legal action syntax. The backbone remains Qwen3-8B; only the system prompt and output grammar change. OS and DB share one system prompt; the strict-format block is task-specific and is appended after the retrieved-context block.

BigCodeBench: system prompt and message order

System prompt.

```
You are an expert Python programmer solving BigCodeBench coding tasks.

You may receive a [Retrieved Memory Context] block with past
experiences from similar problems.
```

```
These are **references for learning**, not guaranteed solutions:
- [MEMORY TYPE] SUCCESS_PROCEDURE: A successful approach from a
similar task---learn the implementation pattern.
- [MEMORY TYPE] FAILURE_REFLECTION: A failed attempt with
lessons---avoid similar mistakes.

Use the memories as inspiration, but always analyze your current task
independently and adapt your approach based on its specific
requirements. Generate clean, correct Python code.

Hard constraints for BigCodeBench:
- Do NOT change the required function signature, return type, or
required exception types/messages.
- Do NOT wrap specific exceptions into generic ones; keep the exact
exception class and message if specified.
- Import every module you use; remove unused imports; do not rely on
implicit imports.
- Avoid broad try/except (e.g., `except Exception`) unless the task
explicitly requires it.
- Avoid any network calls or extra file I/O beyond what the task
specifies.
- Keep code deterministic: no randomness, time-based logic, or
unnecessary logging.
```

Message order. The system prompt still names a `[Retrieved Memory Context]` block; the injected wrap is the code-example header above.

```
1) system: system prompt, then the retrieved-context block if nonempty
2) user: BigCodeBench Instruct prompt
3) assistant: a single Python solution
```

OS / DB: system prompt and action grammar

Shared system prompt.

```
You are an execution-focused AI agent solving database and
operating-system tasks.

You may receive a [Retrieved Memory Context] block with past
experiences from similar problems.
These are **references for learning**, not guaranteed solutions:
- [MEMORY TYPE] SUCCESS_PROCEDURE: A successful approach from a
similar task---learn the pattern.
- [MEMORY TYPE] FAILURE_REFLECTION: A failed attempt with
lessons---avoid similar mistakes.

Use the memories as inspiration, but always analyze your current task
independently and adapt your approach based on its specific
requirements.
```

OS action grammar.

```
STRICT OUTPUT FORMAT (LLB-OS, do not violate):
1) After your reasoning, include exactly ONE action line:
   - Act: bash
   - Act: finish
2) If Act: bash, the next lines MUST be a ```bash fenced code block
with your Bash commands. Do not include any other code blocks.
3) If Act: finish, it must be the last line (no code blocks, no extra
text).
4) Do NOT use `Action:` in OS tasks (use `Act:` only).
```

DB action grammar.

```
STRICT OUTPUT FORMAT (LLB-DB, do not violate):
```

```
1) After your reasoning, include exactly ONE action line:
   - Action: Operation
   - Action: Answer
2) If Action: Operation, put exactly ONE SQL statement in the FIRST
fenced code block using ```sql, on a single line. Do not add any
extra text after that block.
3) If Action: Answer, include `Final Answer: ...` on the next line
and do not add extra text after that.
```

ALFWorld: system prompt and message order

System prompt.

```
Interact with a household to solve a task. Imagine you are an
intelligent agent in a household environment and your target is to
perform actions to complete the task goal. At the beginning of your
interactions, you will be given the detailed description of the
current environment and your goal to accomplish.
For each of your turn, you will be given the observation of the last
turn. You should first think about the current condition and plan for
your future actions, and then output your action in this turn. Your
output must strictly follow this format:"Thought: your thoughts.
Action: your next action".

The available actions are:
1. go to {recep}
2. take {obj} from {recep}
3. move {obj} to {recep}
4. open {recep}
5. close {recep}
6. use {obj}
7. clean {obj} with {recep}
8. heat {obj} with {recep}
9. cool {obj} with {recep}
where {obj} and {recep} correspond to objects and receptacles.
After your each turn, the environment will give you immediate
feedback based on which you plan your next few steps. if the
envrionment output "Nothing happened", that means the previous action
is invalid and you should try more options.

Your response should use the following format:

Thought: <your thoughts>
Action: <your next action>
```

Message order.

```
1) system: ALFWorld system prompt
2) user/assistant: few-shot dialogue (first user turn prefixed with
"Here is an example of how to solve the task:")
3) system: optional retrieved memories
4) user: Now, it's your turn to solve a new task.
   {task_description}
5) loop: Observation -> Thought/Action
```

G LIMITATIONS

Vantage improves utility learning by changing which attempts are executed. It sets $\eta = 0$, so the reward of an attempt is its outcome, and within an epoch a query is re-attempted only after the first attempt fails. Several limitations remain, and each points to a direction left open.

Scope of the study. The question in this paper is what the reward is computed from, and which attempts are run. Retrieval and the utility update stay those of MemRL: a similarity pool followed by the similarity–utility score in Eq. (1), and the update in Eq. (2). Other memory architectures and other retrievers answer a different question, namely how a memory is stored and how it is selected. They are left out. Attaching the same attempt rule to those designs is a natural next step.

Dependence on the outcome label. The reward of an attempt is the outcome returned by the environment. At $\eta = 0$ this reward is $r = y$. When $\eta > 0$, Eq. (3) still takes that outcome as an input. Dense process rewards, learned critics, and LLM-as-a-judge labels are not used here. Replacing the outcome with one of those signals is left for future work.

One reward, and a bank that keeps every write. Every memory shown to f_{llm} in an attempt receives the same reward, and every attempt is written into the bank. The update therefore gives no separate reward to each retrieved memory, and retrieval can still return a redundant one. The bank grows with the number of attempts: 5-epoch Vantage stores fewer memories than 10-epoch MemRL, while 10-epoch Vantage can store more. Separating the reward among the retrieved memories, and removing or merging stored memories, are left open.