AtomMem : Learnable Dynamic Agentic Memory with Atomic Memory Operation

Anonymous ACL submission

Abstract

Equipping agents with memory is essential for solving real-world long-horizon problems. However, most existing agent memory mechanisms rely on static and hand-crafted workflows. This limits the performance and generalization ability of these memory designs, which highlights the need for a more flexible, learning-based memory framework. In this paper, we propose AtomMem , which reframes memory management as a dynamic decision-making problem. We deconstruct high-level memory processes into **fundamental atomic CRUD** (Create, Read, Update, Delete) operations, transforming the memory workflow into a learnable decision process. By combining supervised fine-tuning with reinforcement learning, AtomMem learns an autonomous, task-aligned policy to orchestrate memory behaviors tailored to specific task demands. Experimental results across 3 long-context benchmarks demonstrate that the trained AtomMem-8B consistently outperforms prior static-workflow memory methods. Further analysis of training dynamics shows that our learning-based formulation enables the agent to discover structured, task-aligned memory management strategies, highlighting a key advantage over predefined routines. The project can be accessed at https://anonymous.4open.science/r/AtomMem-86F8.

1 Introduction

Enabling LLM-based agents to accomplish long-horizon and more complex tasks has been a shared goal across both industry and academia (Chen et al., 2025; Erdogan et al., 2025; Wang et al., 2025b). A critical bottleneck in this pursuit is the design of memory mechanisms. Currently, most memory mechanisms of LLM-based agents rely on static, expert-crafted workflows (Xu et al., 2025b; Chhikara et al., 2025; Li et al., 2025). In these systems, memory operations are confined

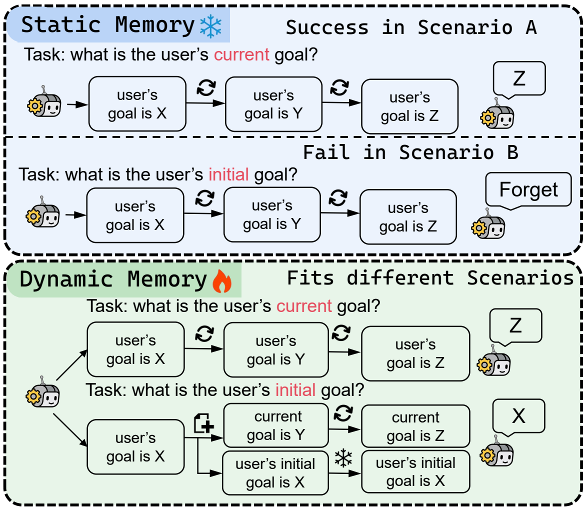

Figure 1: The one-size-fits-all workflow of static memory often fails to adapt to diverse tasks. Instead, a dynamic memory system is needed to determine the optimal memory strategy based on the specific task context.

to rigid, human-designed pipelines, implicitly assuming that a single interaction pattern suffices for diverse scenarios.

The fundamental limitation of these approaches is their "one-size-fits-all" assumption: they impose rigid memory rules for information retention, independent of the varying demands of downstream tasks. Strategies like continuous memory fusion (Xu et al., 2025b) or predefined forgetting schedules (Zhong et al., 2023) may work well in generic scenarios but fail in complex environments. Specifically, continuous memory fusion risks obscuring fine-grained details in precision-sensitive tasks, while rigid forgetting schedules may prematurely discard early yet critical cues in long-horizon reasoning (Wang et al., 2024). As illustrated in Figure 1, such static workflows fail to dynamically adapt to the fluctuating information density inherent in real-world interactions.

Several recent approaches attempt to mitigate this rigidity by introducing content-level flexibility (Wang et al., 2025a; Yu et al., 2025a). However, these methods remain bound by globally fixed workflows. For instance, while MemAgent (Yu et al., 2025a) allows the content of the memory operation to be learned, the update memory action is enforced as a mandatory step in every execution cycle. Consequently, even when new information is sparse or irrelevant, the agent is forced to perform redundant updates. We term this the “content-optimized but workflow-constrained” paradigm, which limits the agent’s ability to allocate cognitive resources effectively.

To address this issue, we propose AtomMem , which reframes memory management of LLM-based agents not as a fixed pipeline, but as a decision-making problem. Drawing inspiration from agent tool learning (Qin et al., 2023; Shen, 2024)—where models learn when to invoke tools based on context, we deconstruct high-level memory processes into their fundamental atoms: the standard CRUD (Create, Read, Update, Delete) operations. This atomization transforms a static memory workflow into a learnable decision process (Sutton et al., 1999; Dietterich, 1999). By training with reinforcement learning (RL), the agent learns a policy over these atomic operations, enabling it to autonomously orchestrate memory behaviors that are adaptively tailored to the specific demands of each decision step.

Across three augmented multi-hop long-context QA tasks, including HotpotQA (Yang et al., 2018), 2WikiMultihopQA (Ho et al., 2020), and Musique (Trivedi et al., 2022), our approach consistently outperforms prior methods reliant on static memory workflows by approximately 2-5 percentage points under the same Qwen3-8B backbone. Moreover, our framework shows an increasing performance advantage as document length grows in Needle-in-a-Haystack evaluations. Together, these results demonstrate that treating memory management as an atomic-level capability optimized via RL is more effective than relying on predefined routines. Beyond overall performance gains, we further uncover an empirical insight into how memory should be managed for QA tasks. The learned policy exhibits a systematic shift in memory operation usage: the frequencies of Create, Update, and Delete operations steadily increase, while reliance on Read actions decreases and stabilizes at a lower level. This suggests that effective memory control for QA tasks benefits from learning structured, task-aligned patterns of memory operation usage, rather than maintaining a fixed or unstructured access strategy.

2 Related Works

Static Memory Workflow Early memory mechanisms in LLM-based agents typically relied on heuristic-based static workflows. These memory mechanisms can be categorized into two types: 1) **Imitation-Based**: Imitation-based approaches refer to transferring designs from natural systems or other engineering domains into agent memory architectures. For example, MemoryBank (Zhong et al., 2023) draws an analogy between agent memory and human memory, while MemGPT (Packer et al., 2024) likens the agent’s context to computer memory. 2) **Prior-Based**: Prior-based approaches refer to carefully crafted workflows designed by human experts based on prior knowledge (Rezazadeh et al., 2025; Hu et al., 2023; Qian et al., 2025). Despite their theoretical appeal, these methods share a common limitation: the memory workflow is hard-coded by experts. This rigidity prevents the agent from adapting its memory strategy to the complexity of specific tasks. In contrast, our work moves beyond static rules, aiming to learn a dynamic memory policy directly from data.

Toward Dynamic Memory Management Beyond static rules, research has shifted toward adaptive memory management (Lu and Li, 2025; Yan et al., 2025b,a), which we categorize into three paradigms: **Intuition-Based**: Works like SCM (Wang et al., 2025a), and AgentFold (Ye et al., 2025) introduced decision gates for specific memory actions like memory summarization, fusion, pruning, or folding. While enabling some dynamic control, these designs treat flexibility as an ancillary feature rather than a core principle, resulting in fragmented and incomplete action spaces. **Summarization-Based**: MemAgent (Yu et al., 2025a) and Mem1 (Zhou et al., 2025) utilize step-wise overwriting summaries. Although overwriting can theoretically emulate any atomic operation, the workflow is restricted to a mandatory “update-at-every-step” routine. This ignores information density, forcing redundant updates even when new data is sparse. **Hyperparameter-Based**: Some prior work recognizes the importance of dynamic memory but achieves this dynamism solely by tuning the hyperparameters of a fixed workflow (Zhang et al., 2025; Xu et al., 2025a). As a result, the limited optimization space restricts the agent to learning only

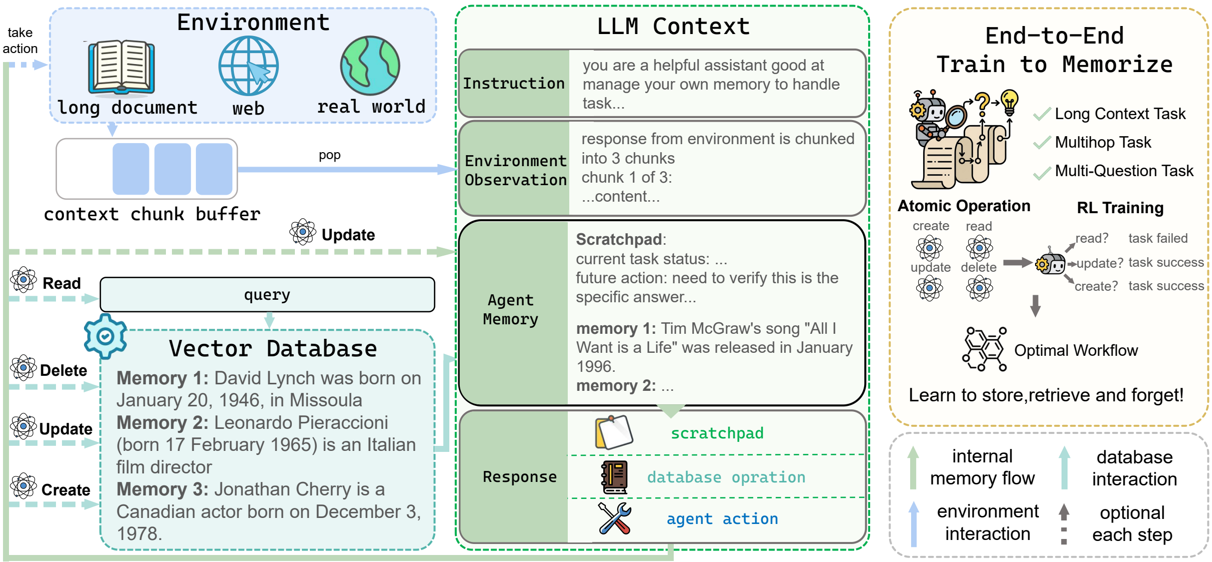

Figure 2: Overview of the AtomMem framework. The agent interacts with long documents, web, or real-world environments while maintaining an external memory. High-level memory workflows are decomposed into atomic CRUD (Create, Read, Update, Delete) operations over a vector database. Through end-to-end reinforcement learning, the agent learns a task-aligned memory management policy that dynamically decides when to store, retrieve, update, or delete information based on task demands.

suboptimal strategies. In contrast, by decomposing memory strategies into atomic operations, we ensure a wide optimization space, increasing the likelihood that the model learns an optimal strategy.

3 Method

In this section, we formulate the memory management of LLM-based agents as a sequential decision-making problem and introduce a complete action space over memory operations.

3.1 Problem Formulation

We formulate the memory management of an LLM-based agent as a Partially Observable Markov Decision Process (POMDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, P, \Omega, \mathcal{O}, \mathcal{R}, \gamma)$. In this framework, the agent must not only interact with the external environment but also manage its internal storage to bridge information gaps over long horizons.

▷ **Global State** $\mathcal{S}$. $s_t \in \mathcal{S}$ at time t consists of two components: $s_t = (s_t^{env}, s_t^{mem})$. Here, s_t^{env} represents the external environment (e.g., stream input progress), and s_t^{mem} represents the current state of the agent's internal memory.

▷ **Action Space** $\mathcal{A}$. An action $a_t \in \mathcal{A}$ is a joint decision $a_t = (a_t^{env}, a_t^{mem})$. While a_t^{env} represents task-specific execution (e.g., answering a question), a_t^{mem} denotes a memory management action chosen from our atomic CRUD space (Create, Read, Update, Delete).

▷ **Transition Function** P. The transition $P(s_{t+1}|s_t, a_t)$ defines how the state evolves. Notably, the internal memory state s_{t+1}^{mem} is directly modified by the agent's memory action (Create/Update/Delete).

▷ **Observation Function** $\mathcal{O}$. The agent does not have direct access to the underlying state s_t. Instead, it receives an observation $o_t \sim \mathcal{O}(s_t, a_{t-1}^{mem})$, which consists of the current environmental input and the memory contents. Crucially, the memory state is not fully observable. Access to memory contents is mediated by explicit **Read** operations, whose execution is decided by the learned policy.

▷ **Observation Space** Ω. An observation $o_t \in \Omega$ represents the agent's visible context at time t, formulated as $o_t = (o_t^{env}, o_t^{mem})$. Here, o_t^{env} is the direct environmental input, and o_t^{mem} contains the memory content (e.g., scratchpad or retrieved entries) determined by the previous action a_{t-1}^{mem}.

To handle environmental inputs o^{env} exceeding the LLM's context window C, we adopt a streaming observation protocol. The input is partitioned into a sequence of (potentially overlapping) fixed-length chunks: $o^{env} = \{c_1, \ldots, c_T\}$, where $|c_t| \le C$. At step t, the agent's observation is restricted to $o_t = \{c_t, o_t^{mem}\}$. This approach ensures the processing of arbitrarily long sequences, contingent on the effective maintenance of task-relevant

information within the memory observation o_t^{mem}.
▷ **Reward Function $\mathcal{R}$.** The agent's objective is to maximize the expected return of the task, defined by the reward function $\mathcal{R}$.

3.2 Memory Mechanism Implementation

The overall architecture of AtomMem is illustrated in Figure 2. Rather than prescribing a fixed memory workflow, we define a set of atomic memory operations that allow the agent to make explicit decisions over memory. Under this formulation, memory is modeled as a persistent, queryable storage whose state evolves according to the agent's learned policy. Specifically, at step t, the memory state is represented as a collection

$$\mathcal{M}_t = \{m_i\}_{i=1}^{N_t}, \tag{1}$$

where each memory entry m_i encodes a piece of stored information.

We expose memory manipulation to the agent through a set of **atomic operations**:

$$\mathcal{A}^{mem} = \{\text{Create}, \text{Read}, \text{Update}, \text{Delete}\}, \tag{2}$$

each corresponding to a primitive state transition over the memory.

At each decision step t, conditioned on the current observation o_t, the agent's policy emits a **sequence** of memory actions:

$$\mathcal{A}_t = a_t^1, \ldots, a_t^{K_t}, \tag{3}$$

which can be viewed as a compositional action within a single environment step. The non-read actions in this sequence are executed sequentially to produce a composed memory state transition:

$$\mathcal{M}_{t+1} = a_t^k(\mathcal{M}_t). \tag{4}$$

where each $a_t^k \in \{\text{Create}, \text{Update}, \text{Delete}\}$.

In contrast, a read operation does not modify the memory state. Instead, it produces a memory observation that is exposed to the agent. Specifically, the **Read** operation exhibits an inherent latency: information requested at step $t-1$ only becomes available at step t. In this case, steps $t-1$ and t should be treated as a single agent-internal transition, as no new environment observation is involved.

Scratchpad To prevent the fine-grained atomic operations from becoming overly fragmented, which could lead to a loss of informational hierarchy and global coherence, we introduce a scratchpad. This scratchpad functions as a centralized memory entry that is **mandatorily retrieved** at every execution step. It is designed to capture the global task state and store pivotal information essential for every step of decision-making. From an atomic perspective, the scratchpad differs from other memory entries only in its retrieval mechanism. Therefore, a unified optimization strategy can be applied to it.

As a result, the agent's observation at each time step is given by

$$o_t = \{o_t^{env}, m_t^{scr}, a_{t-1}(\mathcal{M}_t/m_t^{scr})\}, \tag{5}$$

where m_t^{scr} is the content of the scratchpad, $a_{t-1} \in \{\text{Read}\}$, and $a_{t-1}(\mathcal{M}_t)$ denotes the memory content returned by the READ operation. This formula illustrates two retrieval mechanisms of memory: the deterministic retrieval of m_t^{scr}, and the selective retrieval via the Read action, denoted as $a_{t-1}(\mathcal{M}_t/m_t^{scr})$.

3.3 Optimization Strategy

Since memory operations are realized as structured tokens in the model's vocabulary, optimizing the output sequence likelihood implicitly optimizes the memory policy.

We employ a two-stage training pipeline: (1) We first apply SFT to initialize the model, ensuring it adheres to the API schema and learns basic memory patterns. (2) We further refine the policy using Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to master complex memory management in multi-turn scenarios.

During RL, each training sample corresponds to a multi-step trajectory $\tau = (o_1, a_1, \ldots, o_T, a_T)$, where a_t includes the task-specific action a_t^{env} and the memory operation a_t^{mem}, and the observation o_t is defined in Equation (5). We use task-level success as the reward signal, i.e., no intermediate rewards are provided and only a terminal reward is assigned at the end of the trajectory. After obtaining the reward, we compute the advantage using the following formulation:

$$A_i = r_i - \frac{1}{|G|}\sum_{j \in G} r_j \tag{6}$$

where G denotes the set of trajectories corresponding to repeated executions of the same task, and r_i is the terminal reward of the i-th trajectory. Following Dr.GRPO (Liu et al., 2025), we do not apply normalization to the advantages.

Finally, the advantage is uniformly distributed across all output tokens in the trajectory and optimized according to the following objective:

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G} \rho_\theta^i A_i - \beta \mathbb{D}_{\mathrm{KL}}[\pi_\theta \| \pi_{\mathrm{ref}}]\right] \quad (7)$$

Here, ρ_θ^i denotes the importance sampling ratio for the i-th sample.

Notably, we apply task-level advantages uniformly across all tokens, including memory operations. This enables the agent to jointly optimize memory usage and task performance via RL without external modules.

4 Experiments

In this section, we first introduce our evaluation task and then present the experimental results.

4.1 Evaluation Tasks

We collect commonly used QA datasets: HotpotQA (Yang et al., 2018), 2WikiMultiHopQA (Ho et al., 2020), and MuSiQue (Trivedi et al., 2022) as our data sources. These datasets typically consist of a question paired with several relevant documents, and answering the question requires multi-hop reasoning across these documents. We feed documents to the model, instructing it to memorize information relevant to the question, and finally require the agent to answer using only the memories. To systematically stress-test the memory capabilities of agents, we augment these QA datasets along the following two complementary dimensions.

Long-context Setting Following the RULER (Hsieh et al., 2024) benchmark, we construct arbitrary long-context tasks using the following method: We shuffle the relevant documents and interleave them with a large number of irrelevant documents, constructing a needle-in-a-haystack (NIAH)–style task. This augmentation challenges the agent's ability to identify and remember important information from massive amounts of input. We train on inputs containing 200 documents (28K tokens), and at test time scale the setting to 400 documents (56K tokens) and 800 documents (112K tokens).

Multi-question Setting Following MEM1 (Zhou et al., 2025) and Memory-R1 (Yan et al., 2025b), we provide the model with multiple questions simultaneously. The documents relevant to these questions are shuffled and mixed together before being fed to the agent. After processing all documents, the model is required to answer each question individually. This augmentation strategy challenges the model's ability to manage and maintain multiple semantically independent memories at the same time. Each task contains a randomly sampled number of questions, ranging from 1 to 10. In this setting, the reward obtained for a task is defined as the average of the rewards obtained across its subtasks.

Why do we need dynamic memory management to solve this task? Notably, the relevant documents are interleaved with distractor content and shuffled without regard for their logical dependencies. This shuffling makes the timing and logic order of critical information unpredictable. Consequently, it becomes increasingly difficult for the agent who follows a fixed memory workflow, necessitating a more dynamic management strategy to capture important documents.

4.2 Baselines

We evaluate our method against static baselines with hand-designed strategies (Chhikara et al., 2025; Park et al., 2023, Mem0, Generative Agents) and partially dynamic models that trigger specific actions within fixed workflows (Xu et al., 2025b; Yu et al., 2025a, A-Mem, MemAgent). Comparisons also include standard RAG variants (Gao et al., 2022) and a full-context baseline, with implementation details provided in Appendix A.

4.3 Implementation Details

Models For all agents, we use Qwen3-8B (Yang et al., 2025) as the base model. To ensure a fair comparison with baselines, we disable the thinking mode. For agents that require retrieval, we use Qwen3-embedding-0.6B as the embedding model.

Agent Implementation We implement memory using a vector database as the underlying storage. Under this design, the **Read** operation corresponds to semantic similarity–based retrieval. The query for retrieval is provided by the read action. Each action and its XML format will be detailed in Appendix A. Long texts are split into chunks of 4k tokens and fed to the agent step-by-step. At each step, if a read operation is triggered, the memory module retrieves 6 relevant entries from the database.

Table 1: Results on long-context multi-hop reasoning.

Method	HotpotQA			2WikiMQA			Musique			Avg.
	200doc	400doc	800doc	200doc	400doc	800doc	200doc	400doc	800doc	
Direct Answer	63.5	66.7	62.0	55.7	52.2	49.2	42.8	42.3	41.9	52.9
RAG	67.8	65.0	63.1	46.5	44.8	40.0	38.5	40.2	37.1	49.2
HyDE	70.0	68.6	66.6	47.5	40.7	43.7	46.7	43.9	45.6	52.6
Generative Agents	38.8	24.9	10.0	12.3	4.0	2.0	19.8	14.9	8.4	15.0
Mem0	38.2	35.5	33.9	24.2	17.9	18.3	14.0	13.4	11.2	23.0
A-Mem	73.5	71.8	70.4	62.7	60.9	57.1	47.1	46.3	41.6	59.0
MemAgent	76.5	74.4	71.1	65.8	60.7	57.7	54.7	**50.0**	44.5	61.7
AtomMem (ours)										
SFT	65.9	66.9	60.1	52.8	52.8	55.0	47.8	43.5	40.0	53.9
RL	**77.8**	**76.9**	**72.9**	**67.5**	**65.8**	**62.5**	**55.1**	49.1	**48.5**	**64.0**

SFT Implementation SFT is applied to equip the model with stable instruction-following behavior and basic task completion capability. Specifically, we conduct supervised fine-tuning on 4K prompt–completion pairs sampled from HotpotQA via rejection sampling using DeepSeek-V3.1 (DeepSeek-AI et al., 2025), serving primarily as a lightweight initialization. For MemAgent, to rule out performance differences caused by training-induced misalignment, we also perform SFT with the same amount of data.

RL Implementation For SFT, we only use the model trained on HotpotQA to ensure stable format-following, but for RL, we train on each dataset individually to obtain task-specific policies. We adopt a fully on-policy RL strategy, where each rollout is used for a single update. We use exactly match (EM) between the model answer and the ground-truth as the reward, to prevent potential reward hacking. The RL hyperparameters are also kept identical across trained agents (MemAgent and AtomMem). Additional hyperparameters are provided in the Appendix A.

4.4 Main Results

All data points are averaged over three repeated runs to ensure numerical stability. The main experimental results are shown in Table 1. We highlight the following observations:

(1) **AtomMem achieves superior performance and robust scalability across varying task scales.** It outperforms all trained and untrained baselines on average. Notably, in the 800-document setting—a $4\times$ extension of the training context—our model maintains a significant performance lead. This indicates that the agent has learned a content-aware memory policy capable of mitigating information overload as environmental noise increases.

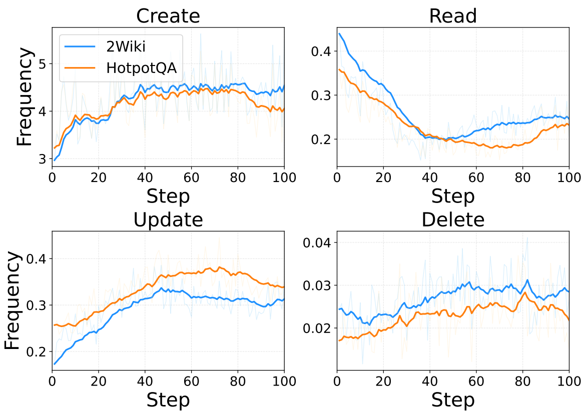

Figure 3: The frequency of the memory operations during the RL training. The y-axis represents the average number of memory API calls made by the model.

(2) **RL training substantially optimizes the agent's memory policy, resulting in large performance gains.** After RL training, AtomMem improves by nearly 10 percentage points on average across different task settings. This improvement indicates that directly optimizing memory decisions with task-level feedback is critical for effective long-context reasoning. In particular, RL enables the agent to refine when and how memory operations are applied under noisy and extended contexts, leading to markedly stronger end performance.

4.5 Training Dynamic Analysis

In this section, we provide a detailed analysis of the RL training dynamics of AtomMem .

As shown in Figure 3, RL training induces systematic changes in the agent's memory operation usage. Specifically, we have the following findings:

(1) **The model's behavior shifts from under-managed to task-aligned memory usage.** Early in training, the model over-relies on Read actions and

Table 2: Ablation study of memory operations and memory components. Percentage values in brackets represent the relative performance decrease.

Method	HotpotQA	2WikiMQA	Musique
AtomMem	77.8	67.5	55.1
Selective Memory Operations			
w/o Update	71.4 (-6.4)	62.6 (-4.9)	47.9 (-7.2)
w/o Delete	76.5 (-1.3)	67.8 (+0.3)	54.2 (-0.9)
Memory Components			
w/o scratchpad	71.8 (-6.0)	56.3 (-11.2)	46.0 (-9.1)
w/o storage	69.2 (-8.6)	59.4 (-8.1)	43.9 (-11.2)
w/o Both	25.6 (-52.2)	27.1 (-40.4)	12.1 (-43.0)

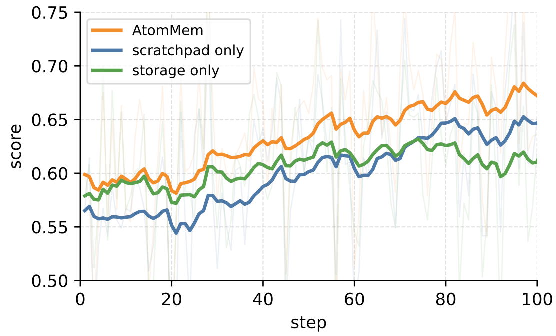

Figure 4: Training curves for optimizing a single component and for jointly optimizing all the components.

largely neglects memory maintenance, leading to redundant retrievals. As training progresses, Read usage decreases sharply, while Create, Update, and Delete actions increase substantially. This transition indicates that the model learns to maintain a compact, task-relevant memory by preserving useful information, revising outdated entries, and removing redundancy.

(2) **While the frequency of Update actions remains low, they represent the critical few that significantly influence the agent's overall performance.** We conduct an ablation study as shown in Table 2, demonstrating that removing Update operations leads to a substantial performance drop across all benchmarks, indicating that selectively revising existing memories is critical for maintaining accurate and compact representations as new evidence arrives. In contrast, disabling Delete has only a marginal impact, suggesting that explicit memory removal is less crucial under the current task distribution, where most stored entries are non-conflicting, and memory capacity is sufficient.

4.6 Ablation and Analysis

In this section, we conduct ablation studies on the various memory components of AtomMem and examine the impact of hyperparameters.

Ablation of Memory Component This experiment investigates the contribution of components to the final performance. The ablation results are reported in Table 2. We can see that:

(1) **AtomMem exhibits robustness to the removal of individual memory components.** Removing either the scratchpad or the external memory storage leads to a moderate performance drop, whereas removing both results in a catastrophic degradation exceeding 40 points. This suggests that when one component is unavailable, the learned policy can still rely on the remaining component to preserve most task-relevant information, rather than collapsing entirely. This indicates that AtomMem is robust to component-level failures.

(2) **Both the memory storage and the scratchpad contribute substantially to the final performance of AtomMem .** Removing either component leads to a consistent performance drop of 5–10 points across all benchmarks. This indicates that the information preserved by the scratchpad and the external memory storage differ fundamentally in domain and usage, such that neither can be fully substituted by the other.

To verify this, we trained another two variants of AtomMem from scratch: scratchpad-only and storage-only. The results are shown in Figure 4. From the experimental results, we observe that **the other two variants do not achieve performance comparable to AtomMem .** The scratchpad-only variant remains consistently below AtomMem during training, whereas the storage-only variant benefits marginally from RL. This indicates that our design effectively raises the performance ceiling of the agent. The significant gap between AtomMem and its variants suggests that the synergy between the scratchpad and memory storage is a structural necessity for handling complex tasks.

Effect of Hyper-Parameters In this experiment, we investigate the effect of several key hyperparameters of AtomMem , including chunk size and retrieve number. The chunk size determines the length of the text segment processed by the agent at each step, while the retrieve number specifies how many entries are retrieved from storage at each step. The experimental results are shown in Table 3.

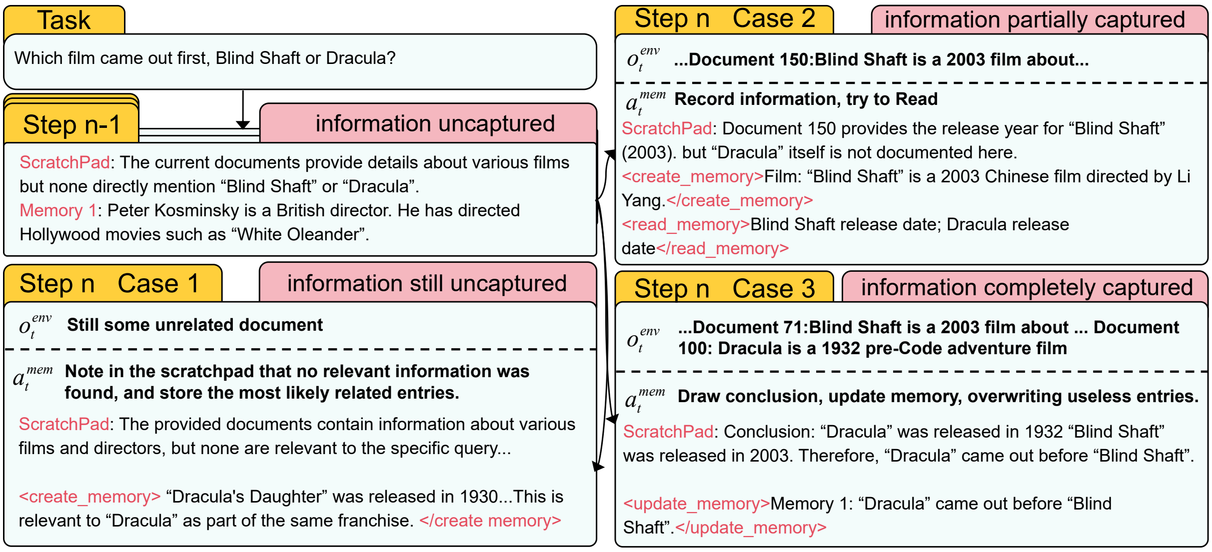

Figure 5: A case illustrates that the model adopts different memory management strategies (a_t^{mem}) when facing different task contexts o_t^{env}. It demonstrates the dynamic nature of AtomMem .

Table 3: Hyperparameter Analysis of Chunk Size (C) and Retrieve Number (K).

C	K	HotpotQA	2WikiMQA	Musique	Avg.
2048	3	76.2	64.2	49.9	63.4
2048	6	76.8	67.6	52.6	65.7
2048	12	75.0	67.3	52.6	65.0
4096	3	74.2	64.8	54.6	64.5
4096	6	76.9	67.5	55.1	66.5
4096	12	77.4	69.6	54.5	67.2
8192	3	76.4	65.2	53.0	64.9
8192	6	78.7	67.9	54.0	66.9
8192	12	76.5	67.5	52.8	65.6

From the results, we find that:

(1) **The retrieval size K must match the task's memory demand.** Reducing K from 6 to 3 causes a clear performance drop, while increasing it to 12 brings little benefit. This is because the evaluated benchmarks require only 2–4 hop reasoning, for which retrieving about six documents is sufficient.

(2) **AtomMem is robust to chunk size.** Performance remains consistent across different chunk sizes, due to the base model's strong long-context understanding and reinforcement learning that enables effective information extraction at varying granularities.

5 Case Study

In this section, we analyze the model's responses on a case-by-case basis to understand what memory workflow the model has learned. As illustrated in Figure 5, we present three scenarios at step n that demonstrate the agent's learned ability to adapt its memory workflow based on the observation o_t^{env}.

▷ **Case 1**: When o_t^{env} contains unrelated documents, the agent uses the scratchpad to log the absence of relevant info and only stores potentially related background entries.

▷ **Case 2**: When o_t^{env} provides partial information (e.g., the release date of a single film), the agent commits the newly found evidence to memory and proactively generates a `<read_memory>` request to retrieve the missing piece.

▷ **Case 3**: In the scenario where all required information is present, the agent synthesizes the retrieved facts within the scratchpad to derive the final answer and uses `<update_memory>` to overwrite useless entries with the conclusion.

Together, these cases illustrate that the agent has learned a **context-sensitive** memory workflow, dynamically deciding when to ignore, retrieve, update, or consolidate memories based on the informational sufficiency of the current observation.

6 Conclusion

In this paper, we propose AtomMem , which reframes agentic memory management as a dynamic decision-making problem by deconstructing complex workflows into atomic CRUD operations. By optimizing this learnable decision process, AtomMem moves beyond the limitations of static, "one-size-fits-all" memory pipelines. Experimental results and training dynamics demonstrate that this approach enables a task-aligned memory policy.

Limitation

Despite its effectiveness, RL optimization is computationally intensive. Training an agent model to convergence requires approximately 2 to 3 days on an 8-GPU cluster. This computational overhead may become a bottleneck when scaling our approach to even longer-horizon or noisier tasks.

Ethical Statement

All data used in this work are sourced from open-source datasets and do not contain personal or private information. The LLM is used solely for writing and sentence refinement. This work does not pose any potential ethical or societal risks.

References

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krahenbuhl. 2025. Reinforcement learning for long-horizon interactive llm agents. *ArXiv*, abs/2502.01600.

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

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Thomas G. Dietterich. 1999. Hierarchical reinforcement learning with the maxq value function decomposition. *Preprint*, arXiv:cs/9905014.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 15419–15462. PMLR.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Precise zero-shot dense retrieval without relevance labels. *Preprint*, arXiv:2212.10496.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *Preprint*, arXiv:2404.06654.

Chenxu Hu, Jie Fu, Chenzhuang Du, Simian Luo, Junbo Zhao, and Hang Zhao. 2023. Chatdb: Augmenting llms with databases as their symbolic memory. *Preprint*, arXiv:2306.03901.

Zhiyu Li, Chenyang Xi, Chunyu Li, Ding Chen, Boyu Chen, Shichao Song, Simin Niu, Hanyu Wang, Jiawei Yang, Chen Tang, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhen Tao, Huayi Lai, Hao Wu, Bo Tang, Zhengren Wang, Zhaoxin Fan, Ningyu Zhang, Linfeng

Zhang, Junchi Yan, Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofen Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. 2025. Memos: A memory os for ai system. *Preprint*, arXiv:2507.03724.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025. Understanding r1-zero-like training: A critical perspective. *Preprint*, arXiv:2503.20783.

Junfeng Lu and Yueyan Li. 2025. Dynamic affective memory management for personalized llm agents. *Preprint*, arXiv:2510.27418.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Memgpt: Towards llms as operating systems. *Preprint*, arXiv:2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael Bernstein. 2023. Generative agents: Interactive simulacra of human behavior.

Hongjin Qian, Zheng Liu, Peitian Zhang, Kelong Mao, Defu Lian, Zhicheng Dou, and Tiejun Huang. 2025. Memorag: Boosting long context processing with global memory-enhanced retrieval augmentation. *Preprint*, arXiv:2409.05591.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

Alireza Rezazadeh, Zichao Li, Wei Wei, and Yujia Bao. 2025. From isolated conversations to hierarchical schemas: Dynamic tree memory representation for llms. *Preprint*, arXiv:2410.14052.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Zhuocheng Shen. 2024. Llm with tools: A survey. *Preprint*, arXiv:2409.18807.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297. ACM.

Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Musique: Multi-hop questions via single-hop question composition. *Preprint*, arXiv:2108.00573.

Bing Wang, Xinnian Liang, Jian Yang, Hui Huang, Shuangzhi Wu, Peihao Wu, Lu Lu, Zejun Ma, and Zhoujun Li. 2025a. Scm: Enhancing large language model with self-controlled memory framework. *Preprint*, arXiv:2304.13343.

Yu Wang, Yifan Gao, Xiusi Chen, Haoming Jiang, Shiyang Li, Jingfeng Yang, Qingyu Yin, Zheng Li, Xian Li, Bing Yin, Jingbo Shang, and Julian McAuley. 2024. Memoryllm: towards self-updatable large language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. 2025b. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *Preprint*, arXiv:2504.20073.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. 2025c. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *Preprint*, arXiv:2504.20073.

Haoran Xu, Jiacong Hu, Ke Zhang, Lei Yu, Yuxin Tang, Xinyuan Song, Yiqun Duan, Lynn Ai, and Bill Shi. 2025a. Sedm: Scalable self-evolving distributed memory for agents. *Preprint*, arXiv:2509.09498.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025b. A-mem: Agentic memory for llm agents. *Preprint*, arXiv:2502.12110.

B. Y. Yan, Chaofan Li, Hongjin Qian, Shuqi Lu, and Zheng Liu. 2025a. General agentic memory via deep research. *Preprint*, arXiv:2511.18423.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. 2025b. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. *Preprint*, arXiv:2508.19828.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao

Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *Preprint*, arXiv:1809.09600.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. 2025. Agentfold: Long-horizon web agents with proactive context management. *Preprint*, arXiv:2510.24699.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. 2025a. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent. *Preprint*, arXiv:2507.02259.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. 2025b. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Zeyu Zhang, Quanyu Dai, Rui Li, Xiaohe Bo, Xu Chen, and Zhenhua Dong. 2025. Learn to memorize: Optimizing llm-based agents with adaptive memory framework. *Preprint*, arXiv:2508.16629.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2023. Memorybank: Enhancing large language models with long-term memory. *Preprint*, arXiv:2305.10250.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. 2025. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. *Preprint*, arXiv:2506.15841.

A Implementation Details

In this section, we list the training hyperparameters, which are shared across all training agents. All training is conducted on NVIDIA A800 GPUs.

A.1 SFT Hyperparameters

For all training agents, we sample 4k prompt–completion pairs from DeepSeek-V3.1, which, under our 200-document setting, correspond to approximately 300 complete trajectories. We use the TRL training framework with a batch size of 16 and train on this data for three epochs. To prevent data leakage, the SFT dataset, RL dataset, and evaluation dataset are strictly isolated.

A.2 RL Hyperparameters

All key RL training hyperparameters are shown in Table 5.

A.3 Agent Implementations

A.3.1 Action Space Protocol

As shown in Table 4, we define four atomic CRUD operations for long-term memory management, each associated with a structured XML schema and explicit parameters. The `Create` operation inserts new content as a standalone memory entry into the vector database. `Read` takes a textual query as input and retrieves the top-k most relevant entries based on vector similarity. `Update` specifies a unique memory identifier along with revised content, enabling selective modification of existing entries. Finally, `Delete` removes a memory entry by its identifier, permanently clearing it from storage. Together, these operations provide fine-grained and interpretable control over memory creation, access, refinement, and removal.

A.3.2 LLM Inference Hyperparameters

The Qwen3 series recommends using a temperature above 0.6 during inference to avoid repetitive outputs and unstable reasoning; therefore, we set the inference temperature of all agents to 0.7. Meanwhile, top-p is set to 1 and top-k is disabled.

A.4 Baseline Implementations

In this work, we use the following baseline:

(1) RAG & HyDE: Each document is individually stored in the vector database (without chunking). During retrieval, for each question, the question itself is used as the query to retrieve six docu-

Table 4: Atomic CRUD Operations for Long-Term Memory Management

Operation	XML Tag Schema	Functionality
Create	<create_memory>{content}</create_memory>	Add new entry to the vector database
Read	<read_memory>{query}</read_memory>	Retrieve top-k relevant entry
Update	<update_memory>{memory id: content}</update_memory>	Modify an existing entry by its identifier
Delete	<delete_memory>memory id</delete_memory>	Permanently remove an entry

Table 5: Reinforcement Learning Hyperparameters

Hyperparameter	Value
RL algorithm	GRPO
Base model	Qwen3-8B
Batch size	16
Rollout Group Size	16
Learning rate	1e-6
Clip Range High	0.28 (Yu et al., 2025b)
Clip Range Low	0.2
Entropy Loss coefficient	0
KL Loss coefficient	0 (Wang et al., 2025c)
Training framework	Verl (Sheng et al., 2025)
Hardware	NVIDIA A800

Table 6: Wall Clock Running Time for Different Methods

Method	Wall Clock Time (second/task)
AtomMem	97.6
MemAgent	49.7
A-Mem	1278.1
Generative Agents	208.0
Mem0	247.8

ments, which are then concatenated and fed to the model for answering.

(2) Direct Answer: We use YaRN scaling to extend the context of Qwen3-8B to 128K tokens to accommodate the 400-document and 800-document settings. All questions are input to the model simultaneously, and it is required to answer them sequentially.

(3) mem0 & Amem& Generative Agents: We follow the same chunking strategy as AtomMem and use official examples to construct the memory library. During retrieval, we adopt the same strategy as RAG: each question is queried separately, and the retrieved results are concatenated before being fed to the model.

B Efficiency Analysis

Efficiency is not the main focus of our study, as there are many opportunities for optimization and parallelization within each agent framework. Nevertheless, we provide a simple efficiency analysis here. The notable differences still demonstrate that AtomMem achieves optimal performance at comparatively high efficiency. The result is shown in Table 6.

Analyzing the experimental results, we make the following observations: **AtomMem and MemAgent achieve much higher processing efficiency compared to other agent memory workflows.** This is mainly because the other workflows invoke the LLM multiple times for each input, and this serialized process significantly reduces the efficiency of the memory mechanism, making it nearly unscalable.

C Prompt

In this section, we present the prompt structure that remains constant throughout the agent's execution. The agent's system prompt and the prompt for its memory fields are shown in Figure 6 and Figure 7.

(Agent Prompt)

System Prompt:
You are presented with a section of an article and a previous memory. Please learn the provided section carefully and manage your memory to answer questions.
Your short-term memory is a step-wise updated summary, while your long-term memory is a vector database that can be updated through atomic operations.
short-term memory is updated using <update_memory>,you should use this every response.
Four kinds of memory actions for the database are available: <update_query>: The system maintains a query for memory retrieval. You can modify it via query operations. You will not get any memory unless you give a query.
<add_memory>: creates a new entry in the memory. You do not need to repeatedly add the memory shown to you or enter the index of the memory.
<modify_memory>: updates the existing entry. You need to enter the memory index "Memory i:" to specify which memory to modify. To manage large volumes of memory effectively, you should prioritize using the "modify" function MORE frequently, rather than relying solely on "add" operations.
<delete_memory>: delete a memory. You need to enter the memory index "Memory i" to specify which memory to delete. You must delete duplicate memory entries!
Use paired XML tags as action markers so you can perform multiple actions—such as adding several memories—in a single response.
action example 1:
<update_query>dance partner; Yulia Zagoruychenko. </update_query>
action example 2:
<add_memory>
Document 10 indicates that the dance event took place in Moscow in October and that Yulia participated in it. I need to focus more on who else attended this event or who traveled to Moscow in October, in order to infer who Yulia's dance partner might be.
</add_memory>
action example 3:
<modify_memory>
Memory 1: The current article provides updated competition records showing that Riccardo Cocchi is now partnered with Emily in the 2025 season, while no recent evidence confirms his continued partnership with Yulia Zagoruychenko. This conflicts with the previous memory stating that Yulia's partner is Riccardo. Since the information clearly supersedes the earlier record, the correct action is to modify the existing memory to reflect that Yulia's current dance partner is unknown as of 2025, and mark the entry for re-verification.
</modify_memory>
action example 4:
It can be observed that Entry 2 and Entry 6 are largely duplicated. Since Entry 6 is more recent, I choose to delete Entry 2.
<delete_memory>
Memory 2
</delete_memory>
Response example:
<update_memory>
1. Memory Framework Overview
The provided section describes a **memory-augmented reasoning framework** in which the agent maintains two complementary memory systems. **Short-term memory** functions as an evolving, step-wise summary that tracks the most immediate context, while **long-term memory** resides in a vector database capable of being updated through atomic operations.
2. Memory Operations
The text further introduces **four atomic memory actions—update_query**, **add_memory**, **modify_memory**, and **delete_memory**. These collectively define how the agent: - retrieves information through query management, - adds new knowledge, - updates outdated or inaccurate entries, and - removes redundant or conflicting information.
3. Functional Role of the Two Memory Types The mechanisms emphasize the importance of **active and continuous memory management**. Short-term memory captures high-level situational summaries that guide immediate reasoning, while long-term memory stores **fine-grained, reusable knowledge** that can support future inference, compensate for missing context, and improve the agent's overall consistency across steps. </update_memory>
<update_query>
short-term vs long-term memory mechanism; atomic operations; memory workflow
</update_query>
<add_memory>
Long-term memory note: The system's memory architecture explicitly separates short-term and long-term roles. Short-term memory is updated every step and acts as a compressed running summary of what the agent has just read, inferred, or decided. In contrast, the long-term memory is a vector-database-backed store meant to hold more detailed, fine-grained, and broadly relevant information—such as definitions, recurring concepts, protocol rules, and any knowledge that may be useful across multiple future queries. The long-term store is maintained through atomic operations (query updating, adding new facts, modifying older entries, and deleting redundant ones), making it flexible and continually improvable as new context appears.
</add_memory>

Figure 6: System prompt for the task.

Agent Prompt

User: This is the question you need to solve:
prompt
This is your short-term memory from the previous turn.
{short_memory}
This is the current query to retrieve memory from the database:
{query}
This is the current memory related to the query:
{long_memory}
Tips: DO NOT repeatedly update the query. If you don't have the desired memory, it means the entry does not exist in the knowledge base. AVOID using update_query multiple times within a single response; instead, you can use a long and composite query to retrieve documents for different questions. The query matches documents based on semantic embeddings, and composite queries are best composed of keywords.
This is the article:
{chunk}

Figure 7: memory prompt for the task.