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. Through widely used reinforcement learning (GRPO), AtomMem learns an autonomous, task-aligned policy to orchestrate memory behaviors tailored to specific task demands. Experimental results across 3 long-context QA benchmarks and 2 web benchmarks demonstrate that the trained model 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 workflows. 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., 2025c). 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., 2025; Chhikara et al., 2025; Li et al., 2025). In these systems, memory operations are confined to predefined pipelines rather than decided autonomously by the model.

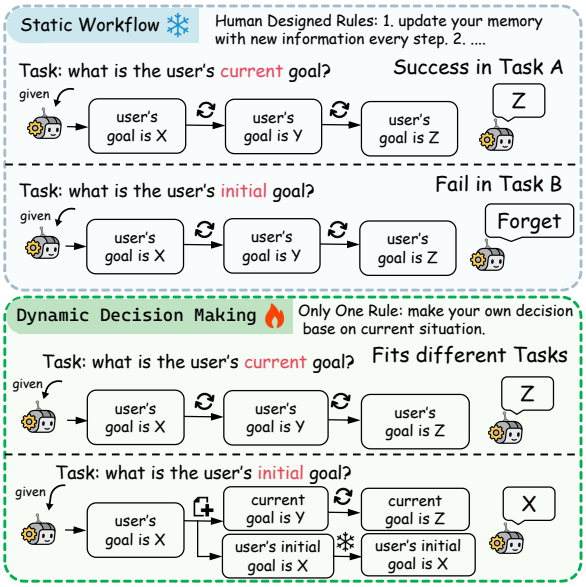

Figure 1: The one-size-fits-all workflow of static memory often fails to adapt to diverse tasks. Instead, a dynamic memory system is needed to determine the optimal memory strategy based on the specific situation.

A core limitation of these approaches lies in their implicit 'one-size-fits-all' assumption: they impose fixed memory management rules for information retention, rather than allowing the model to make autonomous decisions. Strategies like continuous memory fusion (Xu et al., 2025) or predefined forgetting schedules (Zhong et al., 2023) may work well in generic scenarios but fail in complex environments. For example, exponential forgetting schedules may prematurely discard early yet critical cues in long-horizon reasoning (Wang et al., 2024, 2025b). As illustrated in Figure 1, the same static workflow successfully preserves important information in Task A, but fails to do so in Task B. This naturally raises a question: how should we design a more adaptive and effective agent memory system?

To answer this question, we propose AtomMem , which reframes memory management of LLM-

based agents not as a fixed workflow, 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).

A key advantage of this framework is that the effectiveness of memory management is no longer fundamentally bounded by expert rules, but instead by the model's own capacity to make appropriate decisions. By training with reinforcement learning (RL), the agent can acquire experience over these atomic operations and gradually learn a policy for managing memory in a task-aware manner—retaining information that is important for the completion of the task. In this way, memory is no longer treated as a fixed mechanism, but as a kind of policy behavior optimized through interaction with the environment.

We evaluate our method on multiple memory-intensive tasks, which can be broadly divided into two categories: 1) Three multi-hop long-context QA tasks, including HotpotQA (Yang et al., 2018), 2WikiMultihopQA (Ho et al., 2020), Musique (Trivedi et al., 2022), and 2) Two multi-turn web search task including GAIA (Mialon et al., 2023) and WebWalkerQA (Wu et al., 2025). Across all tasks, our approach consistently outperforms prior methods reliant on static memory workflows by approximately 3-8 percentage points under the same Qwen3-8B backbone. 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 these 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. Whereas when the task condition changes, the frequencies follow entirely different trends. This suggests that effective memory control for these tasks benefits from learning task-aligned patterns, rather than maintaining a fixed 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; Wang et al., 2025a). 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 different tasks. For example, architectures designed for QA tasks may be difficult to transfer to agent tasks that interact with the external environment. In contrast, our work moves beyond static rules, aiming to learn a dynamic memory policy directly from different task.

Reinforcement Learning in Agent Memory As reinforcement learning becomes a common method for fine-tuning LLM behavior (Yu et al., 2025b; Wang et al., 2025d), some works have begun using RL to enhance agent memory, which we categorize into two paradigms: **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. **Heuristic-Tool-Based**: Memory-As-Action (Zhang et al., 2025) and AgentFold (Ye et al., 2025) introduce memory management tools such as context pruning or folding. While providing some dynamic control over memory, the tool designs themselves rely heavily on manual priors. In contrast, we provids the model with only the most atomic memory operations, better highlighting the characteristics of the memory-as-decision-making paradigm.

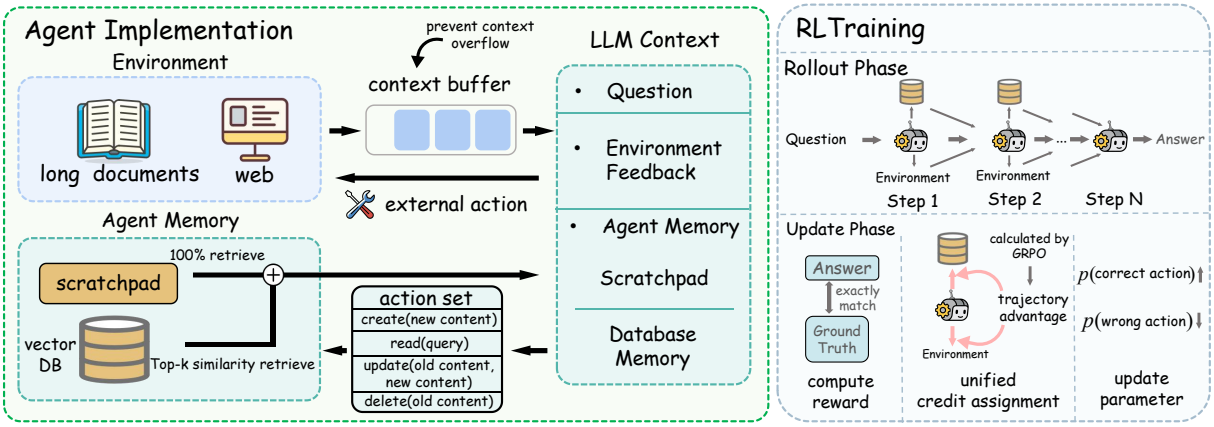

Figure 2: Overview of the AtomMem framework. The agent interacts with environments while maintaining an external memory. High-level memory workflows are decomposed into atomic CRUD (Create, Read, Update, Delete) operations. Through end-to-end reinforcement learning, the agent learns a task-aligned memory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

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 Preliminaries: POMDP for Memory

We model memory management in an LLM-based agent as a Partially Observable Markov Decision Process (POMDP) $(\mathcal{S}, \mathcal{A}, P, \Omega, \mathcal{O}, \mathcal{R}, \gamma)$, where memory is explicitly treated as a controllable component of the environment.

▷ **Global State** $\mathcal{S}$. The global state is $s_t = (s_t^{env}, s_t^{mem})$, comprising the external environment and the internal memory state.

▷ **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., search), a_t^{mem} denotes a memory management action chosen from our atomic CRUD space.

▷ **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 actions.

▷ **Observation Function** $\mathcal{O}$. The agent observes $o_t = (o_t^{env}, o_t^{mem})$. Crucially, memory is not fully observable: o_t^{mem} is determined by previous memory actions (e.g., Read), making memory access an explicit decision variable.

When we formalize memory as a POMDP, it means that we can leverage RL algorithms to optimize the LLM's ability to manage memory. Notably, the agent's memory should be regarded as part of the environment and reset at the start of each independent task, meaning that

$$s_0^{mem} = \varnothing \tag{1}$$

This setting should be distinguished from another type of memory that accumulates experience across different tasks (Zhao et al., 2024; Fang et al., 2025).

3.2 Why Atomic CRUD Operations?

We adopt CRUD (Create, Read, Update, Delete) as the atomic memory action space $\mathcal{A}$ based on three foundational properties.

▷ **Completeness.** CRUD constitutes a universal set of operations capable of synthesizing state-transition in memory. Any valid memory state can be reached from the current state through a series of CRUD operations. This guarantees that, under ideal optimization, the agent's memory can achieve its maximal potential performance.

▷ **Atomic Minimality.** Any higher-level memory tool design can be viewed as invoking a structured combination of CRUD operations. In contrast, the invocation of CRUD primitives themselves cannot be further decomposed into other finer-grained tool designs.

▷ **Task-agnosticness** The CRUD action framework is not tied to any specific downstream task. Its ability to complete tasks relies entirely on the LLM's decision-making capability at a given step, which is itself optimizable. Consequently, this action set constitutes a potential foundation for a general-purpose agent memory.

In summary, rather than proposing isolated memory tools, we focus on a complete and general-purpose operator set that governs memory state

evolution.

3.3 Memory Mechanism Implementation

We model memory at step t as a dynamic set

$$\mathcal{M}_t = \{m_i\}_{i=1}^{N_t}, \tag{2}$$

where m_i encodes a stored memory entry.

Memory manipulation is exposed as a learnable action space

$$\mathcal{A}^{mem} = \{\text{Create}, \text{Read}, \text{Update}, \text{Delete}\}, \tag{3}$$

where each primitive defines a state transition operator over $\mathcal{M}_t$.

At each decision step t, conditioned on observation o_t, the policy generates a sequence of memory actions

$$\mathcal{A}_t = \{a_t^1, \ldots, a_t^{K_t}\}, \tag{4}$$

which forms a compositional macro-action within a single environment step. Non-read operations are executed sequentially, yielding a composed transition

$$\mathcal{M}_{t+1} = a_t^{K_t} \circ \cdots \circ a_t^1(\mathcal{M}_t), \tag{5}$$

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

Hybrid Memory Retrieval The Read operation does not alter the memory state. Instead, it retrieves information from $\mathcal{M}_t$ and produces a memory observation: content requested at step $t-1$ forms the observation at step t. We implement a hybrid retrieval mechanism that combines deterministic retrieval with selective query-based retrieval:

▷ **Deterministic Retrieval (Scratchpad).** A special memory entry m_t^{scr} is retrieved at every step. This scratchpad captures the global task state and preserves pivotal information necessary for stepwise decision-making. Functionally, it is identical to other memory entries, differing only in its mandatory retrieval schedule.

▷ **Selective Retrieval.** The agent generates a textual query q_t as a tool parameter, and relevant memory entries are retrieved based on semantic similarity. Formally, if $\mathcal{M}_t = m_1, m_2, \ldots, m_N$ is the set of memory entries at step t, the retrieved set $\hat{\mathcal{M}}_t$ is:

$$\hat{\mathcal{M}}_t = \text{TopK}\left(\{m_i \in \mathcal{M}_t \mid \text{sim}(q_{t-1}, m_i)\}\right) \tag{6}$$

Under this unified formulation, the agent's observation is given by

$$o_t = \{o_t^{env}, m_t^{scr}, \hat{\mathcal{M}}_t\}, \tag{7}$$

where $\hat{\mathcal{M}}_t$ denotes selective retrieval, and m_t^{scr} corresponds to deterministic retrieval.

The hybrid memory retrieval ranks information by importance: critical information is maintained in the scratchpad, while potentially useful information is stored in the vector database. Experiments show that this multi-path retrieval setup enhances both the effectiveness and robustness of the agent's memory.

3.4 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 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 (7). 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:

$$A_i = r_i - \frac{1}{|G|} \sum_{j \in G} r_j \tag{8}$$

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:

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G} \rho_\theta^i A_i - \beta \mathbb{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]\right] \tag{9}$$

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.

Table 1: Results on long-context QA benchmarks and multi-turn web benchmarks.

Method	HotpotQA		2WikiMQA		Musique		GAIA	WebWalker	Avg.
	200doc	800doc	200doc	800doc	200doc	800doc			
Training-free Methods									
Full Context	63.5	62.0	55.7	49.2	42.8	41.9	23.3	29.5	46.0
Vanilla RAG	67.8	63.1	46.5	40.0	38.5	37.1	20.4	24.0	42.2
Generative Agents	38.8	10.0	12.3	2.0	19.8	8.4	22.3	29.5	17.9
Mem0	38.2	33.9	24.2	18.3	14.0	11.2	25.2	28.3	24.2
A-Mem	73.5	70.4	62.7	57.1	47.1	41.6	30.1	29.0	51.4
Trained Methods									
MemAgent	76.5	71.1	65.8	57.7	54.7	44.5	33.0	**50.0**	56.7
AtomMem w/o RL	65.9	60.1	52.8	55.0	47.8	40.0	35.2	45.6	50.3
AtomMem (ours)	**77.8**	**72.9**	**67.5**	**62.5**	**55.1**	**48.5**	**37.4**	48.7	**58.8**

4 Experiments

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

4.1 Evaluation Tasks

4.1.1 Long Context Benchmarks

We collect 3 QA datasets: HotpotQA (Yang et al., 2018), 2WikiMultiHopQA (Ho et al., 2020), and MuSiQue (Trivedi et al., 2022) as our data sources. All three datasets include training and test splits. We perform RL training on the training split and evaluation on the test split. We feed the document to the model, instructing it to memorize information relevant to the question, and finally require the agent to answer using only the memories. We augment the difficulty of these QA datasets along the following two 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 (about 28K tokens), and at test time scale the setting to 800 documents (about 112K tokens).

Multi-question Setting Following MEM1 (Zhou et al., 2025) and Memory-R1 (Yan et al., 2025), 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.

4.1.2 Web Benchmarks

In multi-turn web search scenarios, we chose the Asearcher (Gao et al., 2025) open-source dataset for training and evaluated our model on the GAIA (Mialon et al., 2023) and WebWalkerQA (Wu et al., 2025) datasets. In addition to the memory management API, we provide two external tools—Google search engine and Jina URL Reader—to enable the model to access the internet. The maximum number of web tool calls per task is set to 40 to fully showcase the contribution of the agent's memory.

4.2 Baselines

We evaluate our method against static baselines with hand-designed strategies mem0 (Chhikara et al., 2025), A-Mem (Xu et al., 2025), GenerativeAgents (Park et al., 2023). We also compared our approach with MemAgent (Yu et al., 2025a), another paradigm that employs RL for training Agent Memory. For the implementation of MemAgent, we used the same training hyperparameters and random seed. Comparisons also include standard RAG 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. For agents that require retrieval, we use Qwen3-embedding-0.6B as the embedding model.

Agent Implementation We implement memory using a FAISS vector database as the underlying storage. 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.

RL Implementation We adopt a fully on-policy RL strategy, where each rollout is used for a single update. For QA tasks, we use exactly match (EM) between the model answer and the ground truth as the reward, and for web tasks, we use an LLM-as-a-judge reward. Additional hyperparameters are provided in the Appendix A.

4.4 Main Results

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× 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.

(2) **RL training substantially optimizes the agent's memory policy, resulting in large performance gains.** After RL training, AtomMem improves by nearly 9 percentage points on average across different task settings. This improvement indicates that directly optimizing memory decisions with task-level feedback is critical for long-horizon tasks. In particular, RL enables the agent to refine when and how memory operations are applied, 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 on QA tasks 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 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.

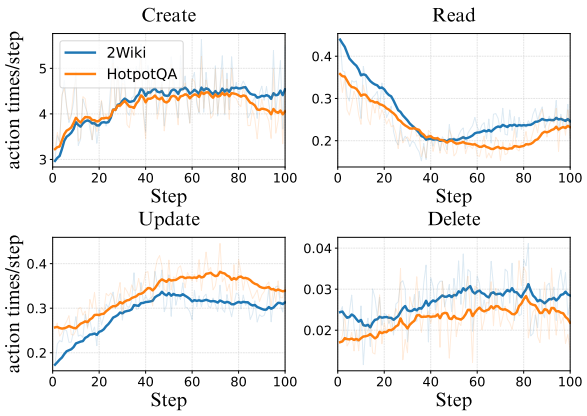

Figure 3: The frequency of the memory operations during the RL training on QA Tasks. The y-axis represents the average number of memory API calls per step made by the model. For instance, a value of 4 indicates that the LLM create 4 memory entries each step.

(2) **While the frequency of Update action remains low compared to Create action, 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, which are largely information-accumulation tasks with non-conflicting facts. To verify this, we added experiments in Appendix C, showing that when the maximum number of memory entries is limited, the frequencies of these actions exhibit different trends of change, while the importance of actions such as Update and Delete becomes greater.

4.6 Ablations

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

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)	67.3 (-0.2)	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)

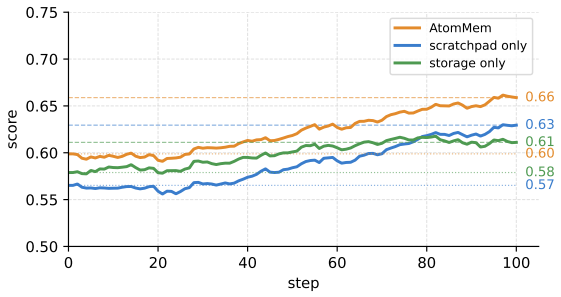

Figure 4: Training curves on HotpotQA for optimizing AtomMem and several of its ablation variants.

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. 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.

Sensitivity to Embeddings We evaluate the impact of embedding models on task performance. The experimental results are shown in the table 4. Frow which we find that:

(1) **The learned embeddings consistently outperforms random selection across all three tasks.** Random selection suffers an average drop of 7.4 percentage points compared to Qwen3-embedding-0.6B. This significant gap indicates that similarity-based matching plays an important role in task performance.

(2) **The Qwen3 embeddings show a clear trend of improvement as model size increases**. Qwen3-embedding-0.6B achieves an average score of 66.5, while the 4B and 8B variants reach 67.2 and 68.2, respectively. This indicates that higher-capacity embedding models can better capture semantic information relevant to the tasks. Overall, the results

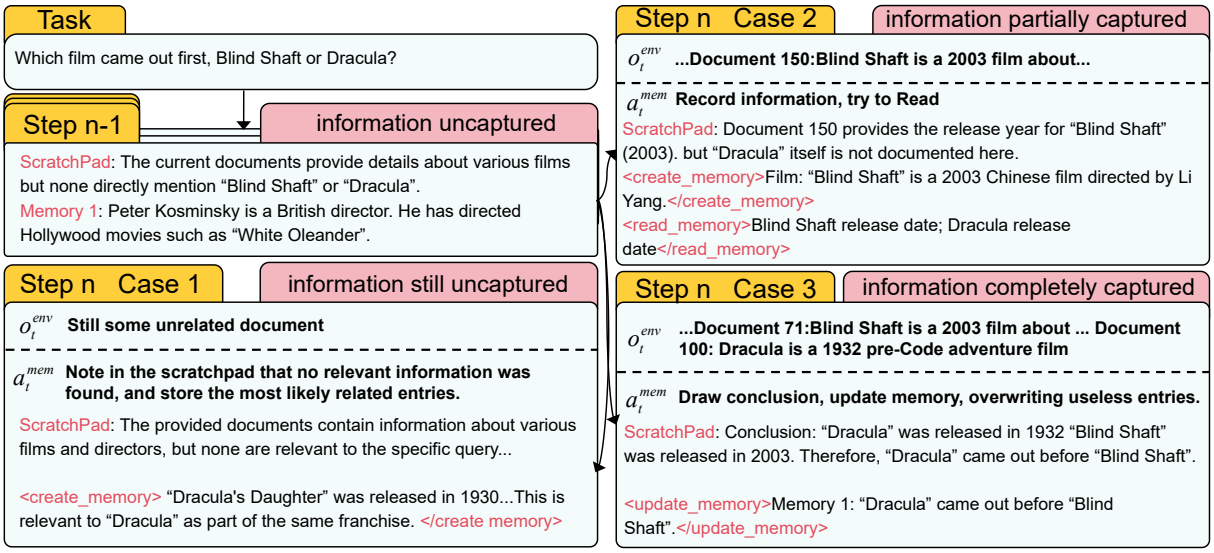

Figure 5: A case illustrates that the model adopts different memory management strategies (a_t^{mem}) when facing different task contexts o_t^{env}. It demonstrates the dynamic nature of AtomMem .

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

Table 4: Sensitivity Analysis of Embedding Models.

Embedding Model	HQA	2Wiki	Musi.	Avg.
Random Select	68.5	62.6	46.1	59.1
Qwen3-embedding-0.6B	76.9	67.5	55.1	66.5
Qwen3-embedding-4B	77.5	68.2	56.0	67.2
Qwen3-embedding-8B	78.6	69.6	56.3	68.2

highlight the importance of choosing an appropriate embedding model for downstream performance.

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}. The example is from HotpotQA, where the LLM made different decisions to complete the task depending on the timing and order in which the key documents appeared.

▷ **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.

In reinforcement learning, task-level advantages are typically evenly distributed across all actions. However, in reality, there should exist more precise methods for assigning advantages, aiming to measure the contribution of each memory entry to the successful completion of a task. We do not explore this direction in the current work for two reasons. First, developing a new RL algorithm for a single downstream task (agent memory) would be unnecessary, as we have already demonstrated that conventional RL methods can effectively optimize performance. Second, accurately evaluating the value of each memory entry is not a trivial problem, and a clear methodology for doing so remains elusive. We leave this problem for future work.

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.

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.

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.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. 2025. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*.

Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu, and Yi Wu. 2025. Beyond ten turns: Unlocking long-horizon agentic search with large-scale asynchronous rl. *Preprint*, arXiv:2508.07976.

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.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. *Preprint*, arXiv:2311.12983.

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.

Yu Wang, Dmitry Krotov, Yuanzhe Hu, Yifan Gao, Wangchunshu Zhou, Julian McAuley, Dan Gutfreund, Rogerio Feris, and Zexue He. 2025b. M+: Extending memoryllm with scalable long-term memory. *Preprint*, arXiv:2502.00592.

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.

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. 2025d. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *Preprint*, arXiv:2504.20073.

Jialong Wu, Wenbiao Yin, Yong Jiang, Zhenglin Wang, Zekun Xi, Runnan Fang, Linhai Zhang, Yulan He, Deyu Zhou, Pengjun Xie, and Fei Huang. 2025. Webwalker: Benchmarking llms in web traversal. *Preprint*, arXiv:2501.07572.

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

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. 2025. 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.

Yuxiang Zhang, Jiangming Shu, Ye Ma, Xueyuan Lin, Shangxi Wu, and Jitao Sang. 2025. Memory as action: Autonomous context curation for long-horizon agentic tasks. *Preprint*, arXiv:2510.12635.

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

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 RL Hyperparameters

All key RL training hyperparameters are shown in Table 7.

A.2 Agent Implementations

A.2.1 Action Space Protocol

As shown in Table 5, 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.2.2 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.3 Baseline Implementations

In this work, we use the following baseline:

(1) RAG: 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 documents, which are then concatenated and fed to the model for answering.

(2) Full Context: We use YaRN scaling to extend the context of Qwen3-8B to 128K tokens to accommodate the 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

Table 5: 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 6: Efficiency Comparison: Wall Clock Time, Average Output Token, and Model Calls per Task

Method	Wall Clock Time (s/task)	Avg. Tokens	Avg. LLM Calls	Avg. Retrieve Calls
AtomMem (ours)	97.6	570.5	10.9	10.9
MemAgent	49.7	264.1	8.0	0.0
Mem0	247.8	431.7	12.9	88.6
Generative Agents	416.0	494.6	375.9	418.0
A-Mem	662.4	237.7	400.0	402.0

Table 7: 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., 2025d)
Training framework	Verl (Sheng et al., 2025)
Hardware	NVIDIA A800
Random Seed	42

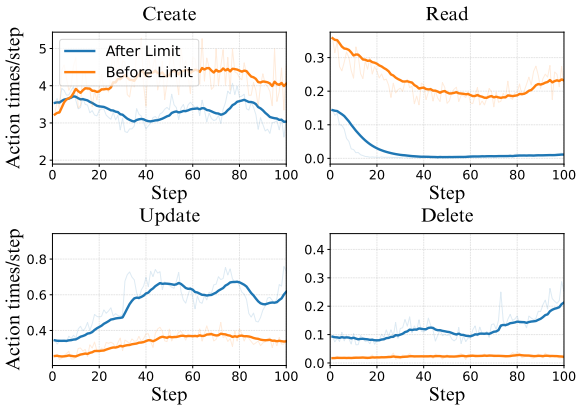

Figure 6: The frequency of memory operations before and after the limitation of database capacity.

same strategy as RAG: each question is queried separately, and the retrieved results are concatenated before being fed to the model.

B Efficiency Analysis

In this section, we provide a simple efficiency analysis. 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:

(1) **AtomMem and MemAgent achieve 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. However, the inference latency of AtomMem is slightly higher than that of MemAgent, due to 1) the increased prompt length caused by the integration of multiple tools, and 2) the addition of a database component, whose retrieval incurs extra latency.

C Memory Capacity Limitation Experiment

We limit the number of memory entries in the memory store and retrain the agent on the HotpotQA. This allows us to observe changes in action rates, such as whether the importance of the delete action increases. Overall, compared with the original experiment, this experiment differs in two aspects: (1) the memory capacity is limited to 20 entries, and any entries exceeding this capacity are discarded; (2) the fine-tuned prompt informs the model of the database limitation and encourages it to make greater use of the Delete operation. The change in action rates is shown in the Figure 6. We observed the following:

(1) The frequencies of the Update and Delete operations increased significantly during training, as these operations do not add entries to the memory.

This is consistent with our expectations.

(2) The frequencies of the Create and Read operations decreased significantly. The decrease in Create operations is intuitive, as the model gradually learns to create entries only within the memory capacity. The frequency of Read operations dropped almost to zero, with only a slight rebound near the end of training. We attribute this to that the large number of entries discarded in the early stage. This led to a substantial loss of useful information, causing the model to learn that subsequent Read operations could not retrieve useful content. In the later stages of training, the model learned to retain only useful information in the database, which in turn led to a very slight recovery in the frequency of Read operations.

D 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 7 and Figure 8.

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_scratchpad>.
Four kinds of memory actions for the database are available: <read_memory>: 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.
<create_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.
<update_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:
<read_memory>dance partner; Yulia Zagoruychenko. </read_memory>
action example 2:
<create_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.
<create_memory>
action example 3:
<update_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.
</update_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_scratchpad>
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—read_memory**, **create_memory**, **update_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_scratchpad>
<read_memory>
short-term vs long-term memory mechanism; atomic operations; memory workflow
</read_memory>
<create_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.
</create_memory>

Figure 7: System prompt for the task.

Agent Prompt

User: This is the question you need to solve:
prompt
This is your scratchpad 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 8: memory prompt for the task.