AtomMem: Learnable Dynamic Agentic Memory with Atomic Memory Operation

Anonymous ACL submission

Abstract

Large language model (LLM)-based agents increasingly face tasks that require reasoning over extremely long range. A key bottleneck is memory design: most prior approaches treat memory as a static, human-designed component, limiting adaptability. Inspired by biological memory evolution, we propose AtomMem, a framework that exposes a set of atomic memory operations—including summarization and CRUD primitives—as tool-like actions. The agent learns a policy over these operations purely via reinforcement learning, using only the task reward. AtomMem allows the agent to dynamically interleave summarization and CRUD to retain important information while managing memory efficiently. We evaluate AtomMem on streaming long-context understanding and web-navigation benchmarks, demonstrating an improvement of 7.4 points over static memory workflows and 2.1 points over the current learnable-memory baseline. Ablation studies show that both summarization and CRUD are essential to train agent memory. The project can be accessed at https://anonymous.4open.science/r/AtomMem-86F8.

1 Introduction

Today's LLM-based agents are increasingly deployed on tasks whose horizons exceed a single context window (Ai et al., 2026; Zheng et al., 2025). A central design in these systems is memory: how an agent retains what is useful across steps and discards what is no longer needed (Chen et al., 2025; Erdogan et al., 2025; Wang et al., 2025b).

We argue that the memory policy of an agent—the full sequence of decisions it makes about its own memory state, including when to compress, when to write, when to retrieve, when to overwrite, and when to forget—should be learned end-to-end from low-level operations, rather than assembled from high-level routines specified in advance. In most current systems (Xu et al., 2025; Chhikara et al., 2025; Li et al., 2025), the bulk of this policy is fixed by the designer: a human decides the workflow, and the model only fills in content within it. As shown in Figure 1, we argue this division of labor fundamentally restricts agent adaptability. A rigid workflow that excels in one scenario often fails in another. To achieve genuine generalizability, the memory policy must be learned from fundamental atomic primitives, rather than constrained to a discrete choice among pre-defined workflows.

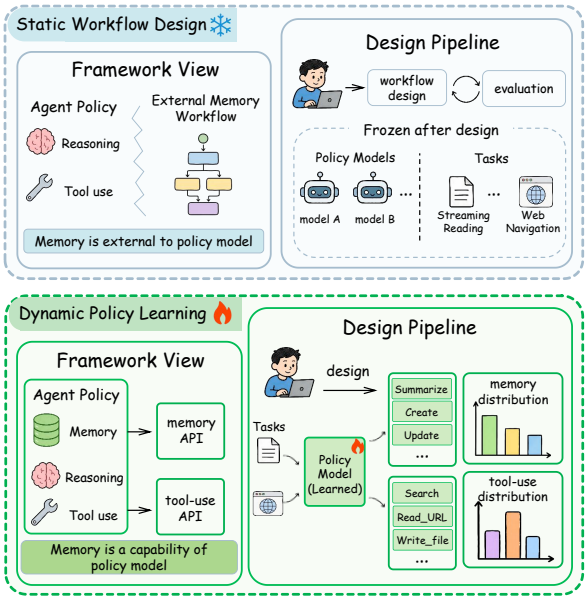

Figure 1: Instead of treating memory as a separate, external component of the agent, we position memory alongside reasoning and tool use, treating it as a essential capability of the policy model.

While recent works have begun exploring learnable memory behaviors, they typically pre-commit the agent's policy to a single class of operations—either continuously rewriting a summarization (e.g., MemAgent (Yu et al., 2025a)) or managing discrete entries in an external store (e.g.,

Memory-R1 (Yan et al., 2025)). In these systems, deciding which class of operations to prioritize remains a human-engineered design-time choice. Our position is the opposite: this choice is itself part of the memory policy, and should also be learned.

We instantiate this view in AtomMem, an agent memory system in which a general set of primitive operations is exposed to a single policy, trained end-to-end with reinforcement learning from task reward. The operations are organized along two dimensions: the object an operation acts on, and the type of operation. We consider two objects: The in-context buffer is single-slot and in-place; it admits one meaningful operation, Update Context, which replaces the buffer with new content. The external store holds addressable entries and admits the standard CRUD operations, with Read treated as a learnable retrieval action rather than a fixed retrieval step. Together, these five operations cover the behavior of existing memory systems, and the policy decides—on a per-step basis—which to invoke.

We extensively evaluate AtomMem on long-horizon tasks, including Streaming Long-context Understanding (tasks constructed from HotpotQA (Yang et al., 2018), 2WikiMultihopQA (Ho et al., 2020), Musique (Trivedi et al., 2022), scaled to over 100k context at test time) and on web navigation (GAIA (Mialon et al., 2023) and WebWalkerQA (Wu et al., 2025)). AtomMem outperforms static-workflow baselines by 7.4 points and surpasses the strongest learnable-memory baseline by 2.1 points. Crucially, AtomMem at 4B parameters achieves comparable performance with 8B baselines, providing evidence that equipping the policy with a richer, lower-level action space yields greater marginal gains than simply scaling up model parameters within a narrow action space. Furthermore, ablation and dynamic analyses reveal that the learned policy does not degenerate into relying on a single operation class. Instead, it proactively shifts its operations (e.g., favoring Update and Delete under tight memory budgets), demonstrating the emergence of a truly task-adaptive memory strategy rather than a rigid routine.

Our contributions are: 1) We formulate agentic memory management as a dynamic decision-making problem over a unified space of atomic operations, shifting the design paradigm from human-crafted workflows to end-to-end learnable policies. 2) We propose AtomMem, an architecture that integrates in-context buffer rewrites and external store CRUD operations into a single policy, optimized end-to-end via reinforcement learning. 3) We provide empirical evidence for the position: a model trained under this broader action space exceeds models trained under narrower ones, and we reveal how agents autonomously evolve memory strategies.

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 chat 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 downstream tasks.

Learning in Agent Memory As reinforcement learning becomes a common method for fine-tuning LLM behavior (Yu et al., 2025b; Wang et al., 2025b), 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. **Heuristic-Tool-Based**: Memory-As-Action (Zhang et al., 2025) and AgentFold (Ye et al., 2025) introduce memory management tools. But they define only a single tool, such as context pruning or folding. Memory-R1 (Yan et al., 2025) defined ADD, UPDATE, and DELETE, but still not rich and lacks analysis of the action space. The question is whether these tools are themselves well-

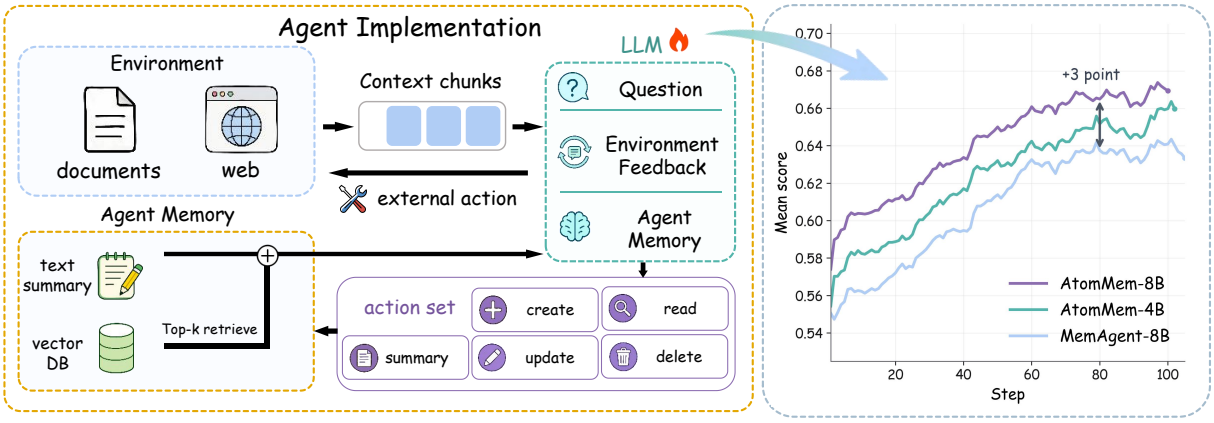

Figure 2: (Left) Overview of the AtomMem framework. The agent interacts with environments while maintaining an external memory. High-level memory workflows are decomposed into atomic (Summary, Create, Read, Update, Delete) operations. Through end-to-end reinforcement learning, the agent learns a task-aligned memory management policy that dynamically decides when to store, retrieve, update, or delete information based on task demands. (Right) AtomMem exhibits a superior optimization curve, with AtomMem-4B can outperform MemAgent-8B.

designed, and how a model would select among them when given the freedom to do so. AtomMem tries to answer this: by unifying multiple memory primitives into an action space and training with RL, we can observe which operations the learned policy actually relies on, providing evidence on whether the action space is well-conceived.

3 Method

We formulate agentic memory management as a sequential decision-making problem and design an action space that exposes memory operations as direct policy decisions. The overview of AtomMem is shown in Figure 2.

3.1 Problem Formulation

We model an LLM-based agent as a Partially Observable Markov Decision Process (POMDP) $(\mathcal{S}, \mathcal{A}, P, \Omega, \mathcal{O}, R, \gamma)$. We factor the system state into an environment component and a memory component:

$$s_t = (s_t^{\text{env}}, s_t^{\text{mem}}), \quad s_0^{\text{mem}} = \varnothing, \tag{1}$$

where s_t^{env} evolves according to task dynamics influenced by the agent's actions, and s_t^{mem} is modified only by the agent's memory actions $a_t^{\text{mem}} \in \mathcal{A}^{\text{mem}}$. At each step, the agent issues a joint action

$$a_t = (a_t^{\text{env}}, a_t^{\text{mem}}), \tag{2}$$

simultaneously deciding how to act in the environment and how to update memory.

This factorization is the formal expression of our position. Because s_t^{mem} is determined entirely by a_t^{mem}, and a_t^{mem} is produced entirely by the policy, no human-designed workflow intervenes between policy decisions and memory state. The memory policy is therefore end-to-end learnable from task reward alone: prior work that fixes a workflow effectively restricts a_t^{mem} to a designer-specified subset of $\mathcal{A}^{\text{mem}}$, while AtomMem imposes no such restriction.

Memory is partially observable. The agent's observation factors as

$$o_t = (o_t^{\text{env}}, o_t^{\text{mem}}), \quad o_t^{\text{mem}} \neq s_t^{\text{mem}} \text{ generally.} \tag{3}$$

When memory consists only of a single in-context buffer, the entire memory state fits in the policy's context and $o_t^{\text{mem}} = s_t^{\text{mem}}$. When memory includes a large external store, however, the policy can observe only what is explicitly retrieved at each step. This partial observability is what makes the formulation a POMDP rather than an MDP, and it is also why retrieval must itself be a policy decision: unlike prior work that treats retrieval as a fixed step external to the policy (Yan et al., 2025; Zhang et al., 2025), we include the retrieval action (Read) in $\mathcal{A}^{\text{mem}}$, allowing the same policy that chooses what to store and modify to also choose what to retrieve.

3.2 Action Space

The remaining question is what operations the agent can perform on memory—that is, what $\mathcal{A}^{\text{mem}}$

Table 1: Comparison of action spaces in different memory systems.

Memory System	Learnable	Target Object	Supported Actions
Amem (Xu et al., 2025)	✗	Vector DB	Tag, Update
mem0 (Chhikara et al., 2025)	✗	Vector DB	ADD, Update, Delete, None
MemAgent (Yu et al., 2025a)	✓	Text Buffer	Summarize
Memory-As-Action (Zhang et al., 2025)	✓	Text Buffer	Prune
AgentFold (Ye et al., 2025)	✓	Text Buffer	Fold
Memory-R1 (Yan et al., 2025)	✓	Vector DB	ADD, Update, Delete, None
AtomMem (ours)	✓	**Both**	Create, **Read**, Update, Delete, Summarize

contains. We organize the action space along two dimensions: the object an action operates on, and the type of operation. We consider two kinds of objects.

The **in-context buffer** is a single-slot state that is modified in place and fully visible to the policy at every step. Because the buffer carries no internal addressing and is rewritten as a whole, only the **Update Context** operation on it is meaningful: replace the buffer with new content. In practice this is most often realized as **Summarize**—compressing the current context into a shorter representation and overwriting the in-context buffer. Notably, we set the in-context buffer as a non-addressable memory, so operations such as prune and fold cannot act on it. These operations use conversation turns as addresses, functioning partially like a database, whereas we employ a separate database module.

The **external store** is a persistent collection of addressable entries that can be individually manipulated. The store admits the standard CRUD primitives:

(1) **Create**: insert a new entry into the store.

(2) **Read**: emit a query and retrieve the top-k entries most relevant to it, making them observable in o_t^{mem}.

(3) **Update**: modify the content of an existing entry, identified by its address.

(4) **Delete**: remove an entry, identified by its address.

As shown in Table 1, two choices distinguish our action space from prior work. First, we expose both the in-context buffer and the external store to a single policy, rather than committing to one of them in advance: the policy decides, at every step, whether to act on the buffer, on the store, or on both. Second, we include **Read** as a learnable action in $\mathcal{A}^{\text{mem}}$—prior database-style approaches (Yan et al., 2025; Zhang et al., 2025) define only the writing operations (Add, Update, Delete) and treat retrieval as fixed. In our view, this fragmented design is unnecessary.

3.3 Training and Implementation

We train the joint policy $\pi(a_t|o_t)$ end-to-end with Group Relative Policy Optimization (GRPO) (Shao et al., 2024), using only the final task reward as supervision.

Reward design Because $\mathcal{A}^{\text{mem}}$ is not directly supervised—there is no ground-truth label for which memory operation is "correct" at each step—the reward signal must come entirely from task outcome. For streaming long-context understanding (SLU) tasks, where reference answers are available in short form, we use exact match (EM) between the agent's final answer and the reference. For web navigation tasks, where correct answers can be expressed in many forms, we use an LLM-as-a-judge reward computed by DeepSeek-V3.2 (DeepSeek-AI et al., 2025).

Memory mechanism We instantiate the external store as a FAISS vector database. Each entry, once Created, is assigned a stable index used as its address; Update and Delete operate on entries by this index. Read emits a natural-language query, which is embedded and compared to entry embeddings by cosine similarity; the top $k = 6$ entries are returned by default. We instantiate the in-context buffer as a text segment placed in the policy's input at every step. Summarize fully overwrites this segment; if no Summarize action is issued in a given step, the segment is retained unchanged.

4 Experiments

Building on the defined external memory structure above, we perform policy optimization experiments to evaluate and refine the agent's use of memory.

4.1 Evaluation Tasks

We conduct experiments on two types of tasks: streaming long-context understanding and web nav-

Table 2: Results on long-context understanding benchmarks and multi-turn web benchmarks.

Model	Method	SLU-HQA		SLU-2Wiki		SLU-Musi		WW	GAIA	Avg.
		200doc	800doc	200doc	800doc	200doc	800doc			
Qwen3-4B	Full Context	60.6	47.7	54.3	34.3	36.6	32.8	25.7	20.3	39.0
	Vanilla RAG	58.7	57.2	44.5	37.9	36.5	32.1	26.0	19.7	39.1
	Generative Agents	17.3	9.2	11.0	2.1	19.6	7.5	23.4	17.0	13.4
	Mem0	28.3	23.0	20.0	11.8	9.7	8.3	26.0	19.1	18.3
	A-Mem	67.8	64.0	51.3	46.8	37.0	34.8	27.3	22.5	43.9
	MemAgent	74.3	62.5	60.1	48.5	52.2	44.5	34.1	24.8	50.1
	AtomMem (ours)	**77.6**	**65.4**	**64.6**	**54.9**	**55.3**	**48.1**	**34.3**	**26.2**	**53.3**
Qwen3-8B	Full Context	63.5	62.0	55.7	49.2	42.8	41.9	29.5	23.3	46.0
	Vanilla RAG	67.8	63.1	46.5	40.0	38.5	37.1	24.0	20.4	42.2
	Generative Agents	38.8	10.0	12.3	2.0	19.8	8.4	29.5	22.3	17.9
	Mem0	38.2	33.9	24.2	18.3	14.0	11.2	28.3	25.2	24.2
	A-Mem	73.5	70.4	62.7	57.1	47.1	41.6	29.0	30.1	51.4
	MemAgent	76.5	71.1	65.8	57.7	54.7	44.5	**50.0**	33.0	56.7
	AtomMem (ours)	**78.1**	**72.9**	**67.5**	**62.5**	**55.1**	**48.5**	48.7	**37.4**	**58.8**

igation.

4.1.1 Streaming Long-context Understanding

Streaming Long-context Understanding (SLU) was proposed by MemAgent and has been widely adopted in works such as MemAct (Zhang et al., 2025). Its core idea is to enable agents to simulate human reading by processing extremely long texts through multiple rounds of input. Our long texts are drawn from three widely used datasets: HotpotQA (Yang et al., 2018), 2WikiMultiHopQA (Ho et al., 2020), and MuSiQue (Trivedi et al., 2022).

Streaming Adaptation We shuffle the relevant documents and interleave them with irrelevant documents, constructing a needle-in-a-haystack–style (NIAH) task (Hsieh et al., 2024). The documents are concatenated into chunks. The model processes one chunk at a time per step until all documents have been read. At the end of the process, the model relies on its memory to answer the questions. We train on inputs containing 200 documents (about 28K tokens), and at test time scale the setting to 800 documents (about 112K tokens). Also, following MEM1 (Zhou et al., 2025), we provide the model with multiple questions simultaneously. 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) for RL training and evaluated our model on the GAIA (Mialon et al., 2023) and WebWalkerQA (Wu et al., 2025). In addition to the memory management API, we provide two external tools—Google search engine and Jina URL Reader—to access the internet.

4.2 Agent Implementation

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). For the implementation of MemAgent, we used the same training hyperparameters. Comparisons also include standard RAG and a full-context baseline. We use Qwen3-4B and 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. We implement memory using a FAISS vector database as the underlying storage. Relevance between a query and each document is computed using cosine similarity, and the top 6 memory entries are retrieved. All the implementation details are provided in Appendix A.

4.3 Main Results

The main experimental results are shown in Table 2. All values correspond to the average@3 on the test set. We highlight the following observations:

(1) AtomMem achieves superior average performance and robust scalability across varying task scales. It outperforms all trained and untrained baselines on average. The baseline method most similar to ours is MemAgent. On average, AtomMem outperforms MemAgent by 2.1 percentage points, demonstrating that AtomMem provides a better starting point for learning a memory policy.

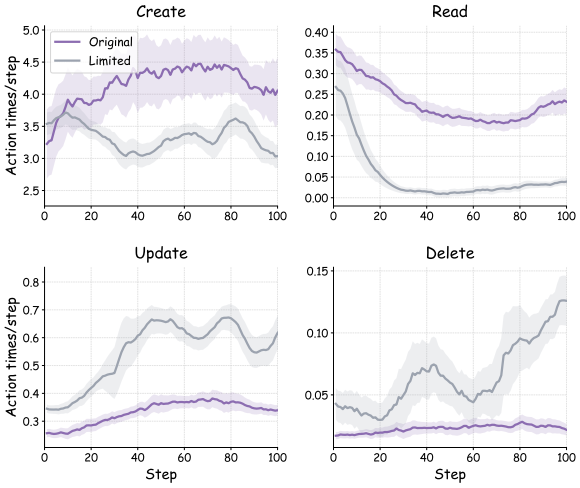

Figure 3: The frequency of the memory operations during the RL training on SLU Tasks. The y-axis represents the average number of memory API calls per step made by the model. For instance, a value of 4 indicates that the LLM create 4 memory entries each step.

(2) In the Streaming Long-context Understanding tasks, AtomMem's advantage over MemAgent becomes larger as the input size increases from 200 to 800 documents. On HQA, the performance gap grows from 1.6 to 1.8 points; on 2Wiki, from 1.7 to 4.8 points; and on MuSi, from 0.4 to 4.0 points. This demonstrates AtomMem's advantage in retaining useful information over longer-horizon tasks, a key property that agent memory aims to achieve.

4.4 Evolution Dynamic Analysis

Beyond the performance gains, it is crucial to discover how the memory policy evolves over the course of training.

(1) As training progresses, the agent's memory retrieval and storage patterns undergo a systematic shift, shown in Figure 3. Read usage decreases sharply, while Create, Update, and Delete actions increase substantially. This transition indicates that the model naturally learns a task-relevant memory policy rather than relying on original pattern. Specifically, the agent evolves to executing more proactive database operations, and shifts from blind, high-frequency Read to more efficient, targeted Read.

(2) Create operation is invoked about 4 times per step on average, whereas Delete is rarely used. Ablation studies in Table 3 show that removing Update causes a severe performance drop, confirming that revising existing memories is critical, while disabling Delete has only a marginal impact

Table 3: Ablation study of memory operations. Percentage values in brackets represent the relative performance decrease.

Method	HotpotQA	2WikiMQA	Musique
AtomMem	78.1	67.5	55.1
w/o Summarize	71.8 (-6.3)	56.3 (-11.2)	46.0 (-9.1)
w/o Update	71.4 (-6.7)	62.6 (-4.9)	47.9 (-7.2)
w/o Delete	76.9 (-1.2)	67.3 (-0.2)	54.2 (-0.9)

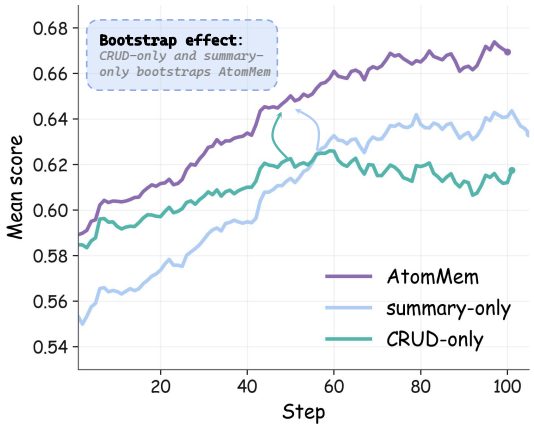

Figure 4: Training curves on SLU task for optimizing AtomMem and several of its ablation variants.

To investigate the necessity of the Delete operation, which prevents memory from growing unbounded, we introduced a strict capacity constraint of 20 entries to the vector database. The experimental results are illustrated in Figure 3. Under this constraint, the model dynamically shifts its strategy: Create are suppressed, while the frequencies of Update and Delete increase significantly. Notably, compared to the unconstrained setting, Read usage in the early training phases is even more severely suppressed, experiencing only a marginal recovery in the later stages. We attribute this sharper drop in Read usage to a full-database bottleneck in early training. When the small database fills up quickly, new write operations fail; consequently, the agent cannot save new information, making the Read operation always fails. Only in later stages, as the model learns to clear space, does the Read action slowly recover.

Just as biological organisms evolve to adapt to their environments, AtomMem dynamically adjusts its memory policy to fit specific tasks.

4.5 Action Space Ablations

The design of the action space determines how the model can select its memory policy. We experi-

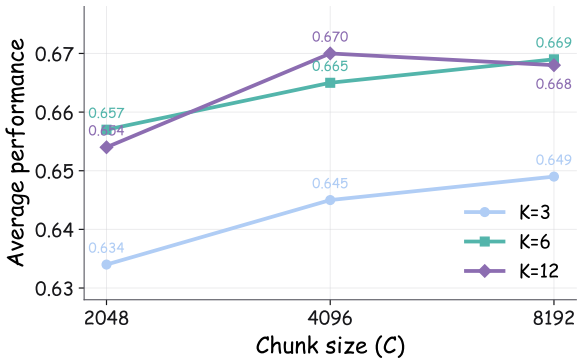

Figure 5: Hyperparameter Analysis of Chunk Size (C) and Retrieve Number (K). The y-axis represents the average score in HQA, 2Wiki and Musique.

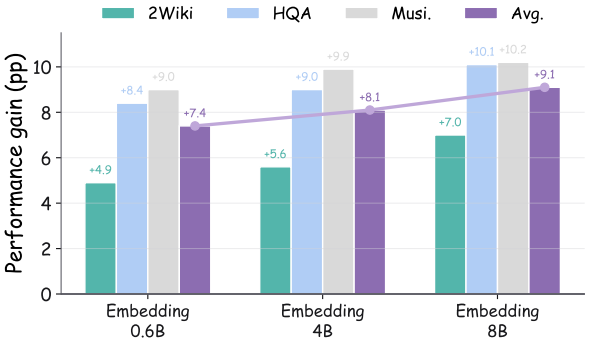

Figure 6: Sensitivity Analysis of Embedding Models. The y-axis represents the relative performance improvement of similarity-based matching compared to randomly selecting memory.

Table 4: Ablation study of memory operations and memory components. Percentage values in brackets represent the relative performance decrease.

Action Space	HotpotQA	2WikiMQA	Musique
AtomMem	78.1	67.5	55.1
CRUD-only	71.8 (-6.3)	62.3 (-5.2)	51.0 (-4.1)
summary-only	76.5 (-1.6)	65.8 (-1.7)	54.7 (-0.4)
no memory	25.6 (-52.5)	27.1 (-40.4)	12.1 (-43.0)

mented with various action spaces and ultimately settled on AtomMem's effective action space. The training curve is shown in Figure 4 and the evaluation results are reported in Table 4. From which we can see that:

(1) CRUD-only variant's optimization effect was limited. As training progressed, the model's performance improvement was less than other action space. We posit that CRUD-only lacks a mechanism to prioritize the importance of information. Even information that is critical for each step is only retrieved at the time of a Read action, rather than being kept in memory to consistently support coherent task execution.

(2) Summary-only is the design of MemAgent, but its performance is significantly lower than that of AtomMem. A per-step summarization action can continuously retain information that the model considers important in memory (addressing the shortcomings of CRUD-only). However, relying solely on a per-step overwrite memory suffers from persistent, irreversible information loss, which in turn degrades the model's performance (Shu et al., 2026).

Taken together, **AtomMem successfully leverages the complementary strengths of both, achieving a bootstrap effect.** This structure is much like the complementary mechanism of short-term and long-term memory. Summarization enables more flexible information flow, akin to short-term memory, while the CRUD-based database provides stable long-term storage.

4.6 Hyper-Parameters Analysis

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 Figure 5. 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 2–4 hop reasoning, for which retrieving about six documents is sufficient.

(2) For different values of K, we observe that increasing the chunk size C leads to performance gains, especially for $K = 3$ and $K = 6$. This suggests that, in agent deployment, splitting the context into smaller chunks may degrade the agent's long-range performance. For example, for a $64k$ text, splitting it into 16 $4k$ chunks performs worse than splitting it into eight $8k$ chunks. A better deployment strategy should fully utilize the model's native context length.

Sensitivity to Embeddings We evaluate the impact of embedding models on task performance. The experimental results are shown in the Figure 6. Frow which we find that:

(1) The learned embeddings consistently out-

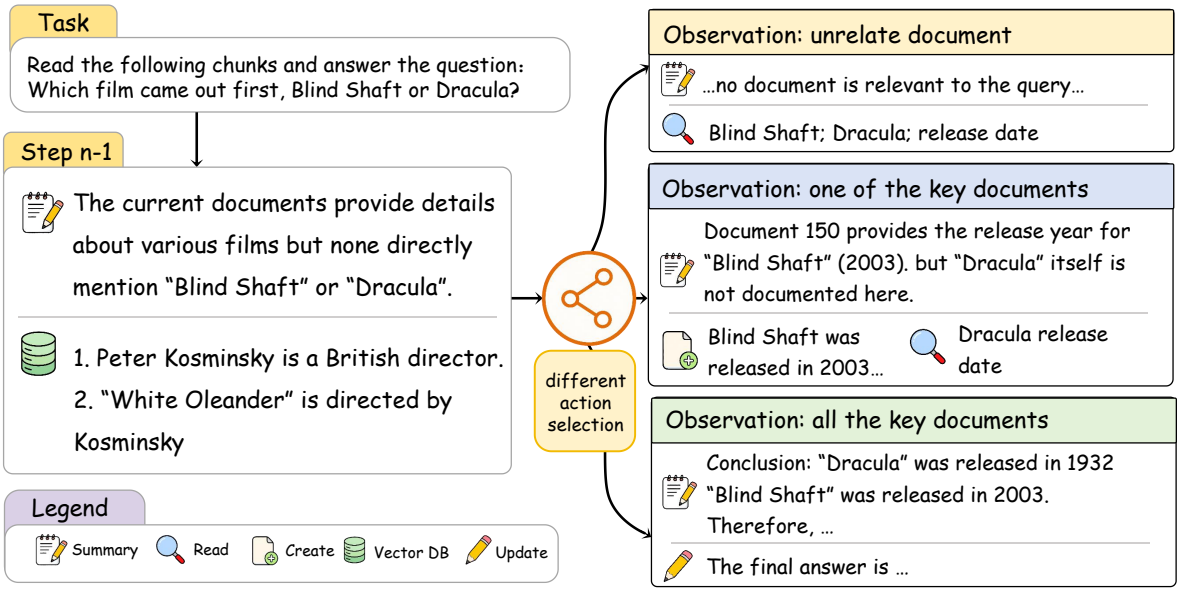

Figure 7: A case illustrates that the model adopts different memory action strategies when facing different observations. It demonstrates the dynamic-decision nature of AtomMem.

performs random selection across all three tasks. Random selection suffers an average drop of 7.4 percentage points compared to Qwen3-embedding-0.6B. This performance improvement does not stem from the memory policy itself, but rather from the underlying hardware infrastructure, indicating that part of the memory system's capability still relies on better hardware implementation.

(2) The Qwen3 embeddings show a clear trend of improvement as model size increases. Qwen3-embedding-0.6B achieves an average performance gain of 7.4, while the 4B and 8B variants reach 8.1 and 9.1, respectively. This indicates that higher-capacity embedding models can better capture semantic information relevant to the tasks. Overall, the results highlight the importance of choosing an appropriate hardware implementation for AtomMem before training a memory policy.

5 Case Study

We analyze the model's responses on a case-by-case basis to understand what memory policy the model has learned. As illustrated in Figure 7, we present three scenarios at step n that demonstrate the agent's learned ability to adapt its action based on the observation.

▷ **Case 1**: When o_t^{env} contains unrelated documents, the agent uses the summary to log the absence of relevant info and tries to call Read to retrieve relevant entries from the database.

▷ **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 action 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 to overwrite the current retrieved entry with the conclusion.

Together, we use these cases to illustrate how the model selects memory actions to handle different situations. Summary is more like a paraphrasing of the current task state along with simple reasoning, allowing the model to flexibly track task progress over time. In contrast, the database primarily stores declarative facts or knowledge, which do not change as the task progresses and are therefore better suited for stable long-term storage.

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 Summarize, Create, Read, Update, Delete operation. By optimizing this learnable decision process, AtomMemmoves beyond the limitations of static memory pipelines. Experimental results indicate that AtomMem provides a better RL action space, and reinforcement learning enables the memory policy to adapt to different environments.

Limitation

In AtomMem, task-level advantages are 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

Qingyao Ai, Yichen Tang, Changyue Wang, Jianming Long, Weihang Su, and Yiqun Liu. 2026. Memorybench: A benchmark for memory and continual learning in llm systems. *Preprint*, arXiv:2510.17281.

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.

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.

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.

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.

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.

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.

Yiheng Shu, Saisri Padmaja Jonnalagedda, Xiang Gao, Bernal Jiménez Gutiérrez, Weijian Qi, Kamalika Das, Huan Sun, and Yu Su. 2026. Remem: Reasoning with episodic memory in language agent. *Preprint*, arXiv:2602.13530.

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.

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.

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.

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

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 AtomMemand use official examples to construct the memory library. During retrieval, we adopt the same

Algorithm 1 Pseudo Code of AtomMem.

Require: Task environment $\mathcal{E}$, policy model π_θ, maximum steps T
Require: External store $D_0 \leftarrow \varnothing$, in-context buffer $B_0 \leftarrow \varnothing$
Initialize environment observation $o_0^{\text{env}} \leftarrow \mathcal{E}.\text{reset}()$
Initialize retrieved memory $R_0 \leftarrow \varnothing$
for $t = 0, 1, \ldots, T-1$ **do**
  Construct memory observation $o_t^{\text{mem}} \leftarrow (B_t, R_t)$
  Form policy input $o_t \leftarrow (o_t^{\text{env}}, o_t^{\text{mem}})$
  Sample joint action $(a_t^{\text{env}}, a_t^{\text{mem}}) \sim \pi_\theta(\cdot \mid o_t)$
  Execute environment action:
    $o_{t+1}^{\text{env}}, r_t, \text{done} \leftarrow \mathcal{E}.\text{step}(a_t^{\text{env}})$
  Execute memory action a_t^{mem}:
  if $a_t^{\text{mem}} = \text{Summarize}(c)$ **then**
    $B_{t+1} \leftarrow c$ ▷ overwrite the in-context buffer
    $D_{t+1} \leftarrow D_t, \quad R_{t+1} \leftarrow R_t$
  else if $a_t^{\text{mem}} = \text{Create}(m)$ **then**
    $D_{t+1} \leftarrow D_t \cup \{m\}$
    $B_{t+1} \leftarrow B_t, \quad R_{t+1} \leftarrow R_t$
  else if $a_t^{\text{mem}} = \text{Read}(q)$ **then**
    $R_{t+1} \leftarrow \text{RetrieveTopK}(D_t, q)$
    $B_{t+1} \leftarrow B_t, \quad D_{t+1} \leftarrow D_t$
  else if $a_t^{\text{mem}} = \text{Update}(i, m)$ **then**
    $D_{t+1} \leftarrow \text{UpdateEntry}(D_t, i, m)$
    $B_{t+1} \leftarrow B_t, \quad R_{t+1} \leftarrow R_t$
  else if $a_t^{\text{mem}} = \text{Delete}(i)$ **then**
    $D_{t+1} \leftarrow D_t \setminus \{D_t[i]\}$
    $B_{t+1} \leftarrow B_t, \quad R_{t+1} \leftarrow R_t$
  else
    $B_{t+1} \leftarrow B_t, \quad D_{t+1} \leftarrow D_t, \quad R_{t+1} \leftarrow R_t$
  end if
  if done **then**
    break
  end if
end for
Update π_θ with task reward using reinforcement learning

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., 2025b)
Training framework	Verl (Sheng et al., 2025)
Hardware	NVIDIA A800
Random Seed	42

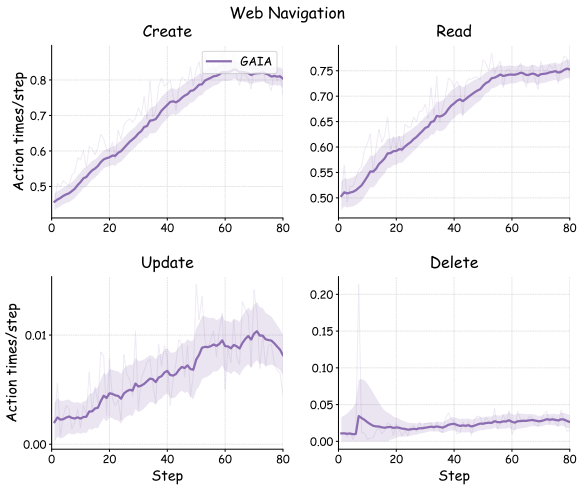

Figure 8: The frequency of the memory operations during the RL training on Web Tasks.

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 Web Navigation Action Rate

In Figure 8, we show the action-rate curves on web tasks. Unlike action rate on SLU tasks, on web tasks, the model invokes Create and Read actions increasingly often as training progresses, while Update actions remain extremely rare. A common pattern, however, is that Create is strongly encouraged throughout training, suggesting that increasingly proactive memory construction is a key driver of performance improvement. The differences between tasks also demonstrate the adaptability of AtomMem.

D SLU Task Prompt

In SLU tasks, the agent's system prompt and the prompt for its memory fields are shown below.

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>

Answer 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}