SWIFTMEM: TOWARDS LIGHTWEIGHT AND EFFICIENT MEMORY FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-based memory systems enable agents to autonomously manage long-term memories, but may introduce substantial token overhead through unnecessary retrieval and excessive memory returns. We propose SWIFTMEM, a lightweight and token-efficient memory system that uses plain text files and three simple memory tools. SWIFTMEM introduces memory probing to guide retrieval decisions, tag-based query enhancement to improve retrieval quality, and Adaptive Memory Subset Selection to balance retrieval utility and cost. Experiments on two benchmarks show that SWIFTMEM achieves superior memory performance while substantially reducing token consumption compared with existing tool-based memory systems. We provide the source code of SWIFTMEM in `https://anonymous.4open.science/r/SwiftMem`.

1 INTRODUCTION

With the rapid advancement of large language models (LLMs), particularly in tool use capabilities, LLM-based agents are increasingly emerging as the dominant paradigm for deploying foundation models. When tackling complex long-horizon tasks, agents often need to leverage historical experience and external knowledge, making the memory system a fundamental component for enabling such capabilities (Li et al., 2025a; Chhikara et al., 2025; Li et al., 2025b; Packer et al., 2023; Tan et al., 2025; Zhong et al., 2024). Existing memory systems are typically designed as standalone modules that operate independently of the agent's main execution loop. Under this design, the timing and strategy of memory reading and writing are usually predefined by developers, resulting in a separation between memory management and the agent's reasoning process. This disconnect makes it difficult to accommodate the dynamic and evolving interaction requirements of open-ended environments (Xu et al., 2025; Yu et al., 2026). As agent applications continue to scale and task complexity increases, this separation has become a critical bottleneck limiting agents' ability to support long-term interactions. To address these limitations, recent studies have begun to encapsulate memory operations as tools that agents can invoke autonomously during reasoning (Yuan et al., 2026; Yu et al., 2026; Anthropic, 2025). This tool-based interface returns control over memory management to the agent, improving the flexibility, transparency, and consistency of memory operations.

However, given that long-term interactions can lead to a potentially massive memory repository, an obvious question arises: despite the significant advantages offered by memory tools, **do they also introduce unnecessary overhead**? Specifically, is invoking a retrieval tool always necessary? Even when retrieval is necessary, can the tool consistently return precisely the memories that are needed? To investigate these questions, we sampled 20 instances from the LongMemEval dataset (Wu et al., 2024) and evaluated a simple tool-based memory system consisting of one memory writing tool and one memory retrieval tool. As shown in Figure 1, we observe two issues in the experimental results: ❶ agents tend to retrieve even when memory access is unnecessary for the current query; and ❷ agents often choose an inappropriate retrieval scope, causing the returned results to contain substantial irrelevant memories. Clearly, these issues increase token consumption and may interfere with the agent's reasoning trajectory, ultimately degrading response quality. To the best of our knowledge, existing work has not provided an effective solution to these problems. Although simple suppression strategies, such as manually reducing the retrieval tool's top-k value, can partially alleviate token overhead, they also increase the risk that agents will miss or overlook necessary memory (Sun et al., 2026), making such approaches suboptimal.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

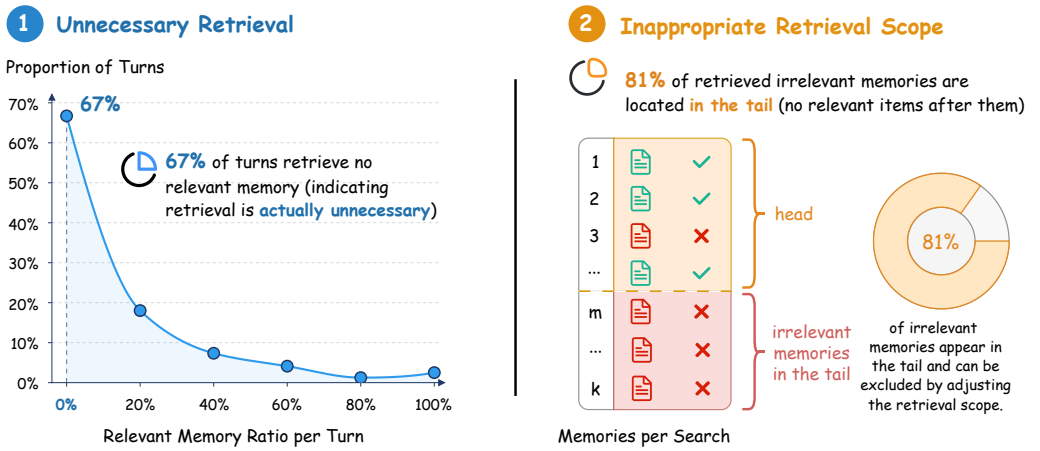


Figure 1: A preliminary experiment of the unnecessary overhead introduced by memory systems using a simple tool-based memory system on LongMemEval. In the left figure, 67% of turns that perform memory retrieval fail to retrieve any relevant memory. Given that the memory repository is relatively small, this suggests that retrieval is unnecessary in these turns. In the right figure, 81% of memories are located at the tail of the similarity-ranked retrieval list and can be excluded by solely adjusting the retrieval scope without affecting relevant memories.

In addition, influenced by earlier memory systems, many tool-based memory frameworks still rely on complex, computationally expensive, and user-opaque components such as embedding models and graph databases. These dependencies are inconsistent with the current trend toward lightweight tool design and make memory inspection and maintenance more difficult for users. Therefore, it is desirable to avoid such complexities while maintaining strong memory performance.

To this end, we propose SWIFTMEM, a lightweight tool-based memory system for agents that uses user-friendly plain text files as the storage medium. SWIFTMEM consists of three tools: `SEARCH`, `VIEW`, and `UPDATE`, which are responsible for searching memory summaries, viewing complete memory contents, and maintaining memories, respectively. We argue that the two aforementioned issues largely stem from the agent's insufficient awareness of the current state of the memory repository, leading to conservative or inaccurate retrieval decisions (Zeng et al., 2026). Our solution is therefore to partially delegate the decisions of whether to retrieve memory and how much memory to retrieve back to the memory system itself. Specifically, we first introduce a memory probing mechanism. After a user query is issued, the system performs a rapid and lightweight probe and injects different prompts into the agent depending on whether potential relevant memories are detected, providing background information and guidance on whether invoking the memory search tool is necessary. Furthermore, we introduce a tag-based query enhancement strategy and an Adaptive Memory Subset Selection (AMSS) algorithm for memory search. The former refines queries using the tags in memory units, while the latter jointly considers multiple factors to adaptively select an optimal memory subset that balances retrieval utility and cost.

Experiments on the LongMemEval (Wu et al., 2024) and BEAM (Tavakoli et al., 2025) benchmarks demonstrate that SWIFTMEM can substantially reduce token consumption while improving memory performance, showcasing its significant advantages. Further experiments validate the contributions of its key strategies and demonstrate the potential for further improvements. In summary, our contributions are as follows:

- We propose SWIFTMEM, a lightweight tool-based memory system with high token efficiency that effectively reduces the memory retrieval overhead of LLM agents.
- We design several novel mechanisms, including memory probing, tag-based query enhancement, and adaptive memory subset selection, which collectively contribute to the effectiveness of SWIFTMEM.
- We conduct extensive experiments to demonstrate the effectiveness of SWIFTMEM and its key mechanisms, while identifying promising directions for future research.

2 Related Work

2.1 Evolution of Memory Systems

The evolution of memory mechanisms for LLMs has largely followed two main lines of development (Mei et al., 2025; Hu et al., 2025). The first is the implicit encoding paradigm, which encodes memories and injects them into the model for implicit access through forward computation. The injection medium may consist of hidden states in intermediate layers (Wang et al., 2024; Li et al., 2025b), key-value (KV) caches (Wu et al., 2022), or model parameters (Meng et al., 2022a;b; Yang et al., 2024). This paradigm offers low access latency but incurs high update costs and presents challenges for memory editing and auditing. The second paradigm, explicit storage, has gradually become the dominant approach. Early work primarily adopted simple retrieval-augmented generation (RAG) strategies (Zhong et al., 2024; Xu et al., 2024; Rau et al., 2024), which, however, have limited effectiveness in scenarios involving continuously evolving information and temporal reasoning (Hu et al., 2026; Maharana et al., 2024; Wu et al., 2024). Building on this line of work, subsequent studies introduced memory management based on LLM-generated memory units, enabling memory updates and deletions. Depending on how memory units are organized, these approaches can be further categorized into memory-entry-based (Chhikara et al., 2025; Huang et al., 2025; Tan et al., 2025; Li et al., 2025a) and memory-graph-based (Gutiérrez et al., 2024; Rasmussen et al., 2025; Xu et al., 2025) paradigms, each offering distinct advantages in different scenarios. Nevertheless, most of these approaches design memory systems as standalone modules decoupled from the agent, which is inconsistent with the current trend toward tightly integrated agent architectures.

2.2 Tool-Based Memory Systems

Driven by the rapid development of various agentic systems, a major paradigm shift in recent memory research has been to encapsulate memory operations as tools that agents can invoke autonomously, thereby returning control over when and how memories are read and written to the agent itself. The pioneering work MemGPT (Packer et al., 2023) proposed an operating system-like virtual context management mechanism that moves data between the main context and external storage through function calls. TA-Mem (Yuan et al., 2026) further expanded the tool set, enabling agents to adaptively select different retrieval tools according to the query, including exact matching, semantic similarity search, and temporal filtering. In addition, several studies have drawn on recent advances in agent research and explored the use of training approaches such as reinforcement learning to enhance agents' end-to-end memory management capabilities (Yan et al., 2026; Yu et al., 2026) or their ability to determine when memory retrieval should be performed (Cai et al., 2026). However, existing tool-based systems typically rely on complex and opaque components inherited from standalone memory systems. Moreover, they focus primarily on improving memory management accuracy while overlooking cost control, resulting in several token efficiency issues.

3 Method

As illustrated in Figure 2, the typical workflow of SwiftMem is as follows. A user query is first processed by the pre-query middleware (§3.4) before being passed to the agent. The agent then interacts with the memory repository (§3.1) through the predefined tool set (§3.2) to read and write memories as needed and generates a response. Finally, the post-query middleware (§3.5) performs the necessary post-processing operations.

Table 1: Memory types and corresponding storage locations.

Type	Description	Storage Location
`personal_fact`	User-related factual information	`~/.memories/user`
`preference`	User preferences	`~/.memories/user`
`project_rule`	Project-specific knowledge	`<cwd>/.memories/project`
`temporary`	Session-specific knowledge	`<tmp_dir>/.memories/temporary`
`business_knowledge`	Business-related knowledge	`<busi_dir>/.memories/<busi_id>`

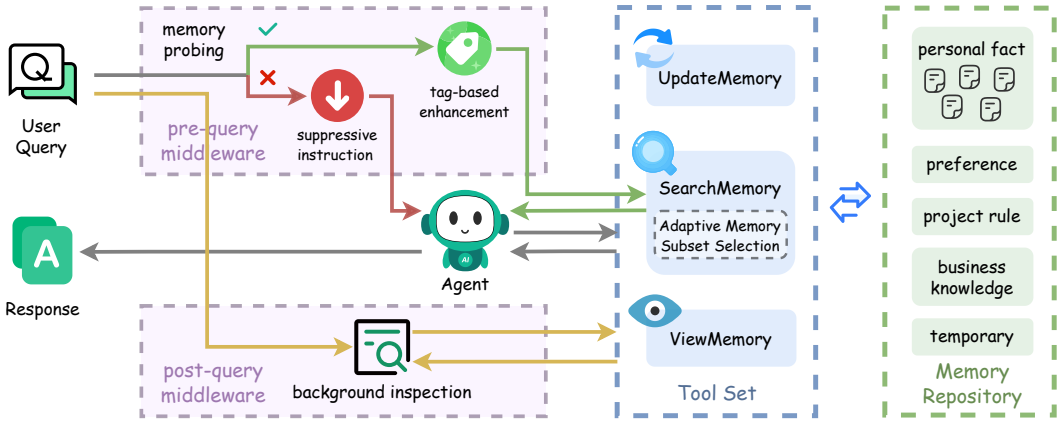


Figure 2: Overview of SWIFTMEM.

3.1 MEMORY FILE STRUCTURE AND ORGANIZATION

SWIFTMEM uses plain text files as the storage medium for memories, with each file representing a single memory unit. The structure of a memory file is illustrated in Figure 3. Among the metadata fields, `type` specifies the category of a memory. Memories of different types are stored in different locations, naturally enabling reasonable memory sharing. For example, memories of type `personal_fact` are stored in the `.memories` directory under the user's home directory, allowing user-specific memories accumulated across different workspaces to be shared seamlessly. The semantics and storage locations of all memory types are summarized in Table 1.

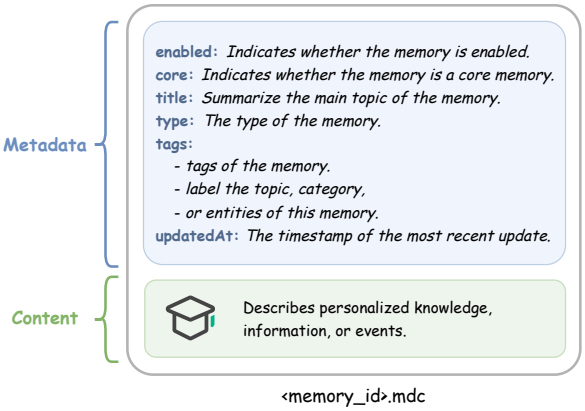


Figure 3: Structure of the memory file.

3.2 MEMORY TOOL SET

SWIFTMEM consists of three tools: `UpdateMemory`, `SearchMemory`, and `ViewMemory`. Their respective functionalities are described as follows:

- **`UpdateMemory`**: A memory management tool that supports creating, updating, and deleting memories. When creating a new memory, agents need to provide necessary parameters including `title` and `tags`.
- **`SearchMemory`**: A memory summary search tool that supports two modes: `BM25` and `Grep`. Given a query provided by the agent, it returns summaries of all matched memories with `enabled` set to true, sorted by `updated_at`. The summaries only contain brief information such as `memory_id`, `title`, `type`, and `tags`, allowing the agent to determine whether it needs to invoke `ViewMemory` to further access the complete memory content.
- **`ViewMemory`**: A memory viewing tool that returns the full content of the specified memory based on the `memory_id` provided by the agent.

The latter two tools together form a progressive disclosure architecture, which has been widely adopted in agent skill tool design (Zheng et al., 2026; Gao et al., 2026; Li et al., 2026). The key idea of this architecture is to separate the reading of summaries and complete contents into two sequential steps, balancing coverage and efficiency during the selective loading of large-scale resources.

It is worth noting that SWIFTMEM additionally maintains two state variables: the recently exposed memory set and memory activity scores. The former records memories after each invocation of `UpdateMemory` and `SearchMemory`, and prevents these memories from being returned by the `SearchMemory` tool for the following several turns, thereby mitigating redundant search. The latter increases the activity score of a memory when the `ViewMemory` tool is invoked and gradually decays as the conversation progresses, reflecting the degree of recent attention paid to the memory. This metric is multiplied with the BM25 score to obtain a weighted score, which is used by the adaptive memory subset selection algorithm described in Section 3.3.

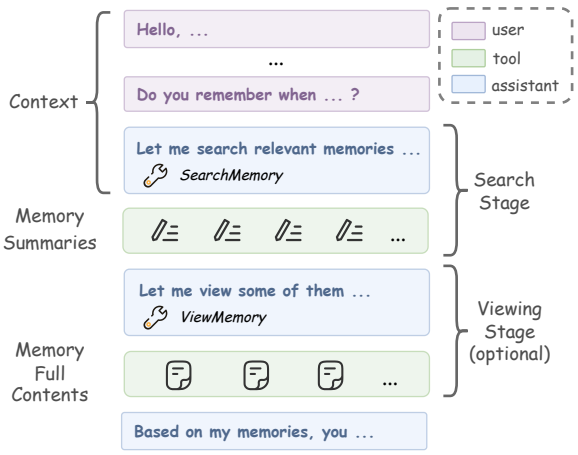


Figure 4: Progressive disclosure workflow of SWIFTMEM.

3.3 ADAPTIVE MEMORY SUBSET SELECTION

To address the challenge that agents struggle to determine an appropriate search scope, we propose an integer-programming-based Adaptive Memory Subset Selection (AMSS) algorithm, which explicitly balances search utility against cost instead of heuristically controlling either the number of searched memories or a similarity threshold. Specifically, the objective of AMSS is to select an optimal subset of memories for each `SearchMemory` call that maximizes the ratio between the expected utility and the expected cost.

We formulate this problem mathematically. Given N candidate memories, we introduce a binary decision variable $x_i, i = 1, \cdots, N$ indicating whether the i-th memory is selected. The following parameters are defined:

- p_i: the probability that the agent subsequently invokes `ViewMemory` to inspect the complete content of the i-th memory. This value is approximated using its weighted score.
- v_i: the utility score of the i-th memory, estimated from its weighted score.
- l_i: The token length of its summary, which determines its cost in the `SearchMemory` output.
- L_i: The token length of its full content, which determines its cost in the `ViewMemory` output.
- M: the total number of tokens in the current context.

As illustrated in Figure 4, the retrieval process consists of two stages: the search stage and the viewing stage. The input token cost of the search stage is fixed at $M + \sum_{i=1}^{N} x_i l_i$. Since the agent may choose not to inspect any of the retrieved memories, the viewing stage (denoted by event V) occurs only probabilistically. Its occurrence probability is $P_V = 1 - \prod_{i=1}^{N}(1 - x_i p_i)$. Accordingly, the expected input token cost of the viewing stage is $M + \sum_{i=1}^{N} x_i l_i + \sum_{i=1}^{N} x_i p_{i|V} L_i$, where we ignore the relatively small token overhead of the `ViewMemory` tool call itself. Combining the two stages yields the total expected cost:

$$C = (1 + P_V)\left(M + \sum_{i=1}^{N} x_i l_i\right) + \sum_{i=1}^{N} x_i p_i L_i.$$

The expected utility is defined as

$$U = \delta + \sum_{i=1}^{N} x_i p_i v_i,$$

where $\delta > 0$ is a base utility constant that discourages retrieval when the expected utility is uniformly low. Under the above definitions, the optimization problem can be formulated as:

$$\max_{\mathbf{x}\in\{0,1\}^N} \frac{U}{C} = \max_{\mathbf{x}\in\{0,1\}^N} \frac{\delta + \sum_{i=1}^{N} x_i p_i v_i}{(1+P_V)\left(M + \sum_{i=1}^{N} x_i l_i\right) + \sum_{i=1}^{N} x_i p_i L_i}.$$

Since the objective function is nonlinear, it cannot be directly solved using standard integer programming solvers. We therefore adopt a hybrid optimization strategy. Specifically, if N is small, we exhaustively enumerate all possible assignments of $\{x_i\}$ and select the one that maximizes the objective function. Otherwise, exhaustive search becomes expensive. In this case, we employ a greedy approximation algorithm: the initial result subset is empty, and at each iteration, we add the memory whose inclusion yields the largest increase in the objective function. The process terminates when adding any remaining memory no longer improves the objective, and the current subset is returned.

3.4 Pre-Query Middleware

Memory Probing and Prompt Injection. After the user issues a query, SwiftMem first performs a lightweight memory probing step to determine whether potentially relevant memories exist. Specifically, the query is matched against the `tags` of all memories to detect any overlap. If a match is found, the query is first enhanced using the tag-based strategy described in the next paragraph, after which relevant memory summaries are searched and injected into the original prompt. Otherwise, a suppressive instruction is injected instead. This provides the agent with additional information before deciding whether to initiate memory retrieval, thereby reducing tool calls. Meanwhile, if the agent determines that the existing information is insufficient or that the instruction is incorrect, it can still invoke the search tool as a fallback.

Tag-Based Query Enhancement. Since BM25 itself lacks mechanisms such as self-attention for dynamically adjusting the importance of different terms based on semantics, certain terms in the query may introduce noise into the search process. This issue is particularly prominent in memory probing, where queries are typically sentences or even paragraphs containing a large number of terms. However, we observe that queries and memories represent the two ends of the search process. Unlike queries, memories are created and updated by the agent with contextual awareness, making it possible to extract representative keywords while filtering out noise. This observation motivates the introduction of the `tags` field in our memory structure. Based on this insight, we match the query against these tags and append the successfully matched tags to the original query, thereby indirectly adjusting the weights of different terms within the query. Compared with strategies that keep the query unchanged and merely increase the weights of memory tags, our approach allows memories whose `content` contains the matched terms to also benefit from the enhanced search signal.

3.5 Post-Query Middleware

In our preliminary experiments, we found that in a fully tool-based memory system, agents often focus primarily on responding to user requests and consequently overlook storing important information contained in those requests. To address this issue, we introduce a background memory inspection mechanism as a fallback. To reduce overhead and maintain contextual coherence, this mechanism is executed once every K turns. It sends the queries from the most recent K turns to a dedicated one-off sub-agent equipped with the complete SwiftMem toolset, which autonomously explores the memory repository, analyzes the queries, and performs appropriate memory management operations.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate SwiftMem on two benchmarks for long-horizon conversational memory, LongMemEval (Wu et al., 2024) and BEAM (Tavakoli et al., 2025), using their 115K and 100K subsets, respectively. Both benchmarks feature long conversations with diverse question types and have been widely adopted for evaluating memory systems. Since tool-based memory systems rely

Table 2: Accuracy (%) and average QA input tokens on LongMemEval.

Method	SS-User	SS-Pref.	Multi-Sess.	Know.-Upd.	Temporal	Overall	Avg. Tokens
AgeMem	91.4	43.3	75.2	88.5	77.4	78.6	12,797
TA-Mem	71.4	66.7	46.6	73.1	42.9	55.4	22,342
RAG (top-k=5)	81.4	40.0	72.2	76.9	43.6	63.7	13,841
RAG (top-k=10)	**95.7**	63.3	**83.5**	88.5	59.4	77.7	34,967
SwiftMem	94.3	**73.3**	78.9	**96.2**	**80.5**	**84.5**	**6,362**

Table 3: Rubric score (%) and average QA input tokens on BEAM.

Method	Abst.	Contra.	Event	Instr.	Know.	Pref.	Temp.	Overall	Avg. Tokens
AgeMem	65.0	**59.7**	**55.0**	85.0	62.5	85.6	59.4	67.5	13,346
TA-Mem	60.0	53.4	39.9	75.6	50.0	86.2	61.9	61.0	72,261
RAG (top-k=5)	**83.8**	47.2	35.8	65.6	17.5	76.5	51.2	53.9	23,904
RAG (top-k=10)	70.0	49.4	34.0	66.9	25.0	81.5	**70.6**	56.8	31,105
SwiftMem	77.5	51.9	39.4	**85.0**	**65.0**	**92.5**	61.9	**67.6**	**11,522**

on the agent's actual responses to manage memory, we replay the historical sessions turn by turn rather than adopting the batched processing strategy commonly used in conventional evaluations. Consequently, we exclude question types that depend heavily on information contained in the assistant turns provided by the benchmarks.

Baselines & Backbone Model. We use *DeepSeek-V4-Flash-Preview* as the backbone model and compare SwiftMem with three partially or fully tool-based memory system baselines: AgeMem (Yu et al., 2026), TA-Mem (Yuan et al., 2026), and a simple RAG-based retrieval tool. For a fair comparison, all vector-based similarity measures in the baselines are replaced with the same BM25-based similarity measure adopted by SwiftMem.

Metrics. To comprehensively evaluate the capabilities of SwiftMem, and in particular to validate our claimed contributions, we report two primary metrics: end-to-end QA accuracy/rubric score and input token consumption during QA, measuring effectiveness and efficiency, respectively. In the ablation study, we further include two metrics: the number of `SearchMemory` tool calls and the number of memories searched during QA.

4.2 Main Results

Performance. As shown in Tables 2 and 3, SwiftMem achieves the best overall performance on both benchmarks while consuming the fewest input tokens. This advantage is particularly evident on LongMemEval, where SwiftMem ranks among the top two methods across all five question types while requiring only about half as many input tokens as the second most efficient method, demonstrating outstanding token efficiency. Notably, the RAG baseline incurs relatively high token consumption even with a small `top-k` setting, as it lacks fine-grained mechanisms for processing retrieved memories.

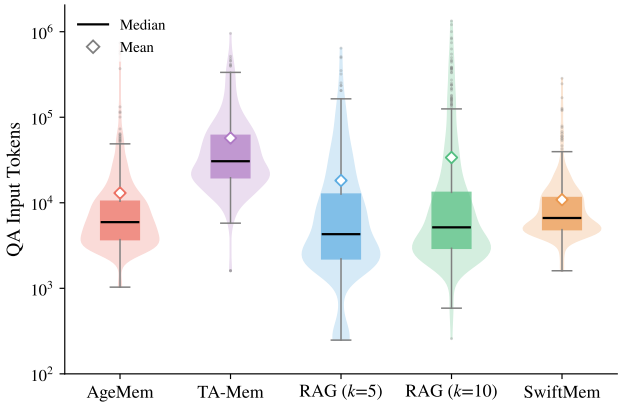


Figure 5: Box-and-violin plots of token consumption for different methods.

Token Analysis. Figure 5 presents the distribution of token consumption for different methods on the two benchmarks. Although the progressive disclosure architecture and

Table 4: Ablation study on LongMemEval.

Method	Acc.(%)	Tokens	Search Calls	Searched Memories		
				Injected	*Tool Recalled*	***Total***
SWIFTMEM	84.5	6,362	1.05	6.28	5.12	11.40
w/o PI	82.2 2.7% ↓	10,583 66.3% ↑	2.53 141.9% ↑	0.00	17.75	17.75 55.6% ↑
w/o AMSS	80.4 4.8% ↓	7,505 18.0% ↑	1.20 15.1% ↑	15.04	3.79	18.82 65.1% ↑
w/o QE	79.7 5.6% ↓	6,907 8.6% ↑	1.15 10.1% ↑	8.78	4.36	13.14 15.2% ↑

prompt injection strategy introduce a relatively higher token overhead for SWIFTMEM at the lower end of the distribution, our design effectively suppresses further token growth. As a result, the 75th percentile is only slightly higher than that of AgeMem, while SWIFTMEM exhibits the fewest outliers beyond the whiskers (accounting for 5.4%). These results demonstrate the robustness of SWIFTMEM in handling extreme cases. In long-term agent deployment, such stability is arguably more beneficial to the user experience than merely achieving a lower average token consumption.

4.3 ABLATION STUDY

To quantify the contribution of each key component in SWIFTMEM, we conduct three ablation studies on LongMemEval by removing the memory probing and prompt injection mechanism (*w/o PI*), the AMSS algorithm (*w/o AMSS*), and the tag-based query enhancement strategy (*w/o QE*), respectively. The results are presented in Table 4 and Figure 6, where the number of searched memories is further divided into memories injected by the memory probing mechanism (*Injected*) and those recalled through tool calls initiated by the agent during execution (*Tool Recalled*).

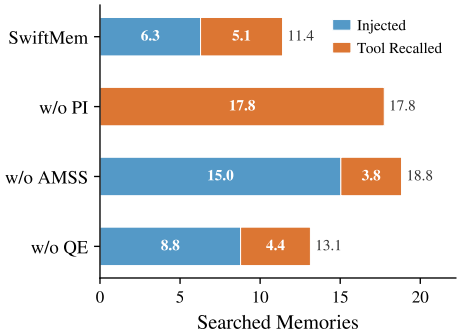


Figure 6: Number of searched memories for different variants.

Overall, removing any of these components leads to lower accuracy while consuming more tokens, demonstrating their effectiveness. Furthermore, SWIFTMEM requires fewer `SearchMemory` invocations and searches fewer memories than all three variants, even when memories searched through prompt injection are counted. This directly confirms that our design effectively alleviates both excessive retrieval invocations and inappropriate retrieval scope.

Among the three variants, *w/o QE* causes the largest drop in accuracy while introducing the smallest increase in token consumption. This is likely because, without query enhancement, prompt injection introduces more noisy memories that occupy the limited budget and crowd out relevant memories, thereby leading to reasoning errors. In contrast, both disabling prompt injection entirely (*w/o PI*) and injecting more memories by removing AMSS (*w/o AMSS*) substantially increase the total number of searched memories, resulting in significantly higher token consumption. These findings suggest that neither extreme—delegating retrieval control entirely to the framework nor entirely to the model—is optimal. Instead, combining the strengths of both yields the best trade-off.

4.4 MARGINAL RETURNS ANALYSIS

To further highlight the advantages of SWIFTMEM, we conduct a marginal returns analysis. Intuitively, retrieving more memories may improve answer accuracy, but it inevitably consumes more tokens and incurs higher costs. Moreover, as accuracy continues to improve, achieving an additional increment in accuracy typically requires increasingly greater cost, a phenomenon known as the law of diminishing marginal returns. Therefore, a steeper decline in marginal returns indicates lower efficiency in converting additional retrieval cost into performance gains.

We compare this trend between SWIFTMEM and the best baseline, AgeMem, on LongMemEval, with the results shown in Figure 7. Since the retrieval cost of SWIFTMEM cannot be directly controlled by adjusting parameters such as `top-k`, we indirectly vary its token consumption through system prompts. Specifically, in addition to the original configuration, we introduce two variants for both methods: *high* and *max*. Their system prompts respectively encourage the agent to ① perform additional retrievals after the initial attempts fail, and ② continue retrieving until it is certain that no relevant memory exists. In this way, we increase the retrieval cost by encouraging more retrieval rounds.

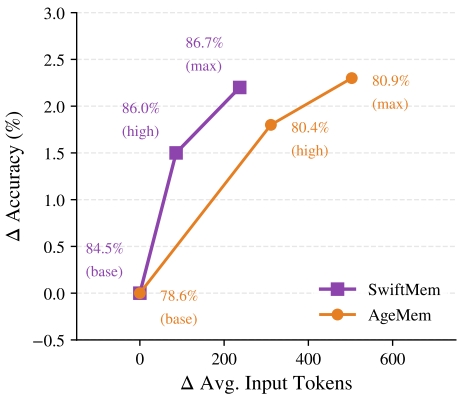


Figure 7: Accuracy as a function of token consumption for SWIFTMEM and AgeMem.

The results show that, compared with AgeMem, SWIFTMEM exhibits a substantially slower decline in marginal returns. This further demonstrates that SWIFTMEM achieves higher token efficiency through more effective system design and more precise memory retrieval.

5 LIMITATIONS

Generalizability. We choose BM25 in SWIFTMEM to achieve lightweight deployment and minimize external dependencies, but this inevitably comes with some performance limitations. For example, BM25 cannot effectively match semantically equivalent words with different lexical forms, although the use of `tags` can partially alleviate this limitation. Therefore, replacing BM25 with an embedding-based retriever presents a trade-off. In this case, some designs in SWIFTMEM, such as tag-based query enhancement, may no longer be optimal, and how to optimize token consumption under such a paradigm remains an open question for future exploration.

Evaluation Representativeness. Existing memory evaluation benchmarks (Maharana et al., 2024; Jiang et al., 2025) typically design the queries in evaluation turns as questions that clearly require memory to answer. However, they rarely consider scenarios commonly encountered in real-world interactions where the need for memory is implicitly embedded in the query. For example, a query such as "Recommend me some movies" may implicitly require the user's memory of movie preferences. Moreover, existing benchmarks rarely include queries for which relevant memory does not exist or memory retrieval is unnecessary. These scenarios are precisely where SWIFTMEM is expected to offer the greatest advantages. Therefore, we argue that current evaluation results may not fully capture the true token efficiency gains of SWIFTMEM in practical use.

Memory Scope. Currently, SWIFTMEM focuses only on memories involving personal information or domain knowledge contained in user queries, without considering experiential memories accumulated by the agent while performing tasks or answering questions. This limitation reduces the practical applicability of SWIFTMEM to some extent. We make this choice primarily because experiential memories may involve research topics such as skill (Shen & Tamkin, 2026) and self-evolution (Gao et al., 2025; Fang et al., 2025). Effectively integrating a memory system with these capabilities is a complex problem, and incorporating them into the current work would distract from our main research focus. Nevertheless, we plan to explore this direction in future work to further extend and improve SWIFTMEM.

6 CONCLUSION

We present SWIFTMEM, a lightweight and token-efficient tool-based memory system for LLM agents. By combining memory probing, tag-based query enhancement, and adaptive memory subset selection, SWIFTMEM effectively reduces unnecessary retrieval and memory exposure while maintaining strong memory performance. Experiments on LongMemEval and BEAM demonstrate that SWIFTMEM achieves competitive accuracy with fewer input tokens. Further analyses validate the effectiveness of its key components.

AI USAGE DISCLOSURE

In this work, we used generative AI tools to assist with implementing the system code. We **have not** used generative AI tools for generating synthetic data sets, developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, assisting with translation, cleaning or reformatting datasets, supporting qualitative or thematic data analysis, or interpreting results. Additionally, we used generative AI tools for creating and editing software code and editing the paper to improve readability. We have reviewed all AI-assisted work: all AI-assisted code was reviewed, tested, and validated by the authors, and all AI-edited text was checked by the authors for technical accuracy. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The complete source code of SWIFTMEM, together with the evaluation pipeline, all prompts, and configuration files, is publicly available in the anonymous repository at `https://anonymous.4open.science/r/SwiftMem`. The design of the memory file structure and the memory tools, the hyperparameter settings of the AMSS algorithm and the activity score mechanism, and all prompts used by the pre-query and post-query middleware are described in the appendix. The experimental setup, including the backbone model, the benchmark subsets, the excluded question types, and the token counting protocol, is described in the Experiments section and the appendix. Both benchmarks used in this paper are publicly available, and all reported results can be reproduced with the released code.

REFERENCES

Anthropic. Memory tool. `https://platform.claude.com/docs/zh-CN/agents-and-tools/tool-use/memory-tool`, 2025. Accessed: 2026-07-16.

Yuxuan Cai, Wei Li, Jie Zhou, Qin Chen, Xin Li, Bo Zhang, and Liang He. Ask only when needed: Proactive retrieval from memory and skills for experience-driven lifelong agents. *arXiv preprint arXiv:2604.20572*, 2026.

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

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

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

Yudong Gao, Zongjie Li, Yuanyuan Yuan, Zimo Ji, Pingchuan Ma, and Shuai Wang. Skillreducer: Optimizing llm agent skills for token efficiency. *arXiv preprint arXiv:2603.29919*, 2026.

Bernal J Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. *Advances in neural information processing systems*, 37:59532–59569, 2024.

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

Zhanghao Hu, Qinglin Zhu, Runcong Zhao, Di Liang, Hanqi Yan, Yulan He, and Lin Gui. Beyond rag for agent memory: Retrieval by decoupling and aggregation. *arXiv preprint arXiv:2602.02007*, 2026.

Zhengjun Huang, Zhoujin Tian, Qintian Guo, Fangyuan Zhang, Yingli Zhou, Di Jiang, and Xiaofang Zhou. Licomemory: Lightweight and cognitive agentic memory for efficient long-term reasoning. *arXiv preprint arXiv:2511.01448*, 2025.

Bowen Jiang, Yuan Yuan, Maohao Shen, Zhuoqun Hao, Zhangchen Xu, Zichen Chen, Ziyi Liu, Anvesh Rao Vijjini, Jiashu He, Hanchao Yu, et al. Personamem-v2: Towards personalized intelligence via learning implicit user personas and agentic memory. *arXiv preprint arXiv:2512.06688*, 2025.

Hao Li, Chenghao Yang, An Zhang, Yang Deng, Xiang Wang, and Tat-Seng Chua. Hello again! llm-powered personalized agent for long-term dialogue. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5259–5276, 2025a.

Yanchao Li, Wanhao Liu, Ben Gao, Jiaqing Xie, Zhehong Ai, Na Zou, Yuqiang Li, and Tianfan Fu. Skillsinjector: Dynamic skill context construction for llm agents. *arXiv preprint arXiv:2605.29794*, 2026.

Zhiyu Li, Shichao Song, Hanyu Wang, Simin Niu, Ding Chen, Jiawei Yang, Chenyang Xi, Huayi Lai, Jihao Zhao, Yezhaohui Wang, et al. Memos: An operating system for memory-augmented generation (mag) in large language models. *arXiv preprint arXiv:2505.22101*, 2025b.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753*, 2024.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334*, 2025.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in gpt. *Advances in neural information processing systems*, 35:17359–17372, 2022a.

Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer. *arXiv preprint arXiv:2210.07229*, 2022b.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: Towards llms as operating systems. 2023.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: a temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

David Rau, Shuai Wang, Hervé Déjean, and Stéphane Clinchant. Context embeddings for efficient answer generation in rag. *arXiv preprint arXiv:2407.09252*, 2024.

Judy Hanwen Shen and Alex Tamkin. How ai impacts skill formation. *arXiv preprint arXiv:2601.20245*, 2026.

Chung-En Sun, Linbo Liu, Ge Yan, Zimo Wang, and Tsui-Wei Weng. Llm agents already know when to call tools–even without reasoning. *arXiv preprint arXiv:2605.09252*, 2026.

Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, et al. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8416–8439, 2025.

Mohammad Tavakoli, Alireza Salemi, Carrie Ye, Mohamed Abdalla, Hamed Zamani, and J Ross Mitchell. Beyond a million tokens: Benchmarking and enhancing long-term memory in llms. *arXiv preprint arXiv:2510.27246*, 2025.

Yu Wang, Yifan Gao, Xiusi Chen, Haoming Jiang, Shiyang Li, Jingfeng Yang, Qingyu Yin, Zheng Li, Xian Li, Bing Yin, et al. Memoryllm: Towards self-updatable large language models. *arXiv preprint arXiv:2402.04624*, 2024.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024.

Yuhuai Wu, Markus N Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. *arXiv preprint arXiv:2203.08913*, 2022.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. Recomp: Improving retrieval-augmented lms with context compression and selective augmentation. In *The Twelfth International Conference on Learning Representations*, 2024.

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

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

Hongkang Yang, Zehao Lin, Wenjin Wang, Hao Wu, Zhiyu Li, Bo Tang, Wenqiang Wei, Jinbo Wang, Zeyun Tang, Shichao Song, et al. Memory3: Language modeling with explicit memory. *arXiv preprint arXiv:2407.01178*, 2024.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885*, 2026.

Mengwei Yuan, Jianan Liu, Jing Yang, Xianyou Li, Weiran Yan, Yichao Wu, and Penghao Liang. Tamem: Tool-augmented autonomous memory retrieval for llm in long-term conversational qa. In *2026 9th International Conference on Advanced Algorithms and Control Engineering (ICAACE)*, pp. 2684–2688. IEEE, 2026.

Yirong Zeng, Shen You, Yufei Liu, Qunyao Du, Xiao Ding, Yutai Hou, Yuxian Wang, Wu Ning, Haonan Song, Dandan Tu, et al. The tool-overuse illusion: Why does llm prefer external tools over internal knowledge? *arXiv preprint arXiv:2604.19749*, 2026.

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuai Zhu, Yong Wu, Tianze Xu, Baohua Dong, Hangcheng Zhu, Ruohui Huang, et al. Skillrouter: Skill routing for llm agents at scale. *arXiv preprint arXiv:2603.22455*, 2026.

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

A Implementation Details of SwiftMem

A.1 Memory File

The following is a concrete example of a memory file:

```
---
core: false
createdAt: '2026-08-04T22:22:35.839543'
enabled: true
tags:
- travel
```

```
- Suica
- transit
- Shinjuku
- Tsukiji
- Nikko
- Narita
title: Suica card for Tokyo transit
type: personal_fact
updatedAt: '2026-08-04T22:22:35.839543'
---

On May 22, 2023, the user mentioned they just got a Suica card for
    their Tokyo trip and plans to use it for public transit, including
     getting from Shinjuku to Tsukiji Fish Market and to the Nikko
    National Park tour meeting point, and from Narita Airport to
    Shinjuku Station.
```

A.2 MEMORY TOOLS

A.2.1 UPDATEMEMORY

The input schema of this tool is shown below:

```
{
    "type": "object",
    "properties": {
        "action": {
            "type": "string",
            "enum": ["create", "update", "delete"],
            "description": "The action to perform on the knowledge base.
            ↪  Defaults to 'create' if not provided. Must be `create`
            ↪  if there is no existing memory."
        },
        "existing_knowledge_id": {
            "type": "string",
            "description": "Required if action is 'update' or 'delete'.
            ↪  The ID of **existing memory** to update instead of
            ↪  creating new memory."
        },
        "knowledge_to_store": {
            "type": "string",
            "description": "The specific memory to be stored. Required
            ↪  for 'create' and 'update' actions."
        },
        "title": {
            "type": "string",
            "description": "The title of the memory to be stored. This
            ↪  can be used to look up and retrieve the memory later.
            ↪  Required for 'create' and 'update' actions."
        },
        "memory_type": {
            "type": "string",
            "enum": ["preference", "personal_fact", "project_rule",
            ↪  "business_knowledge", "temporary"],
            "description": "The type of memory. 'preference': user
            ↪  long-term preferences; 'personal_fact': user personal
            ↪  facts; 'project_rule': project-level coding/workflow
            ↪  rules; 'business_knowledge': domain terminology,
            ↪  metrics, formulas; 'temporary': session-only context.
            ↪  Defaults to 'project_rule' if not specified."
        },
        "tags": {
            "type": "array",
```

```
            "items": {"type": "string"},
            "description": "Tags that label the topic, category, or key
            ↪  entities of this memory. Each tag should be concise (1-3
            ↪  words, preferably single words). Examples: 'travel',
            ↪  'python', 'budget', 'fitness', 'meeting'. Must not be
            ↪  empty for 'create' action."
        },
        "core": {
            "type": "boolean",
            "description": "Whether this memory is core and should
            ↪  always be injected into the context. Only set to true
            ↪  for constraints that must be followed in most tasks.
            ↪  Defaults to false."
        }
    },
    "required": ["action"]
}
```

A.2.2 SEARCHMEMORY

The input schema of this tool is shown below:

```
{
    "type": "object",
    "properties": {
        "query": {
            "type": "string",
            "description": "A single query string. Must be at least 2
            ↪  characters. e.g. 'coffee', 'ice cream'."
        },
        "queries": {
            "type": "array",
            "items": {"type": "string"},
            "description": "One or more query strings. Each query must
            ↪  be at least 2 characters and is executed independently;
            ↪  the union of all results is returned. e.g. ['coffee',
            ↪  'ice cream']."
        },
        "mode": {
            "type": "string",
            "enum": ["bm25", "grep"],
            "description": "Search mode: 'bm25' for relevance-based
            ↪  ranking, 'grep' for keyword matching with '*' wildcard
            ↪  support. Defaults to 'bm25'. For grep mode, do not use
            ↪  '*' alone as the entire query | include at least one
            ↪  concrete keyword, e.g. 'coffee*' rather than just '*'."
        },
        "memory_type": {
            "type": "string",
            "enum": ["all", "preference", "personal_fact",
            ↪  "project_rule", "business_knowledge", "temporary"],
            "description": "Filter by memory type. Defaults to 'all'."
        },
        "top_k": {
            "type": "integer",
            "description": "Max number of candidates from initial
            ↪  retrieval before smart recall. -1 means no limit.
            ↪  Defaults to -1. When use_smart_recall is true, it is
            ↪  recommended not to set this parameter."
        },
        "use_smart_recall": {
            "type": "boolean",
            "description": "When true, applies smart recall after
            ↪  initial retrieval to optimize the result set. Defaults
            ↪  to False."
```

```
            }
        },
        "required": []
}
```

After each invocation of `UpdateMemory` and `SearchMemory` by the agent, SWIFTMEM records the relevant memories in a session-level persistent file. For `UpdateMemory`, these are the memories that were created or updated, while for `SearchMemory`, they are the memories returned by the tool. SWIFTMEM then prevents `SearchMemory` from returning these memories during the subsequent five interaction turns of the current session, thereby reducing redundant retrieval.

A.2.3 VIEWMEMORY

The input schema of this tool is shown below:

```
{
    "type": "object",
    "properties": {
        "id": {
            "type": "string",
            "description": "A single memory ID to view full content."
        },
        "ids": {
            "type": "array",
            "items": {"type": "string"},
            "description": "One or more memory IDs to view full
            ↪  content."
        }
    },
    "required": []
}
```

Whenever the agent invokes `ViewMemory` to inspect the full content of a memory m_i, SWIFTMEM increases its activity score according to the following formula:

$$A_i = \min(\alpha A_i, A_{\max}),$$

where α is the amplification factor and $A_{\max}$ is the maximum activity score, set to 2 and 16, respectively. At the end of each conversation turn, the activity scores of all memories undergo a decay:

$$A_i = \max(\gamma A_i, A_{\min}),$$

where γ is the decay factor and $A_{\min}$ is the minimum activity score, set to 0.95 and 1, respectively. All memories are initialized with an activity score of 1.0. These scores are stored in a global persistent file, enabling activity information to be shared across different sessions.

The activity score is combined with the original BM25 scores to obtain the weighted memory score, calculated as follows:

$$\bar{S} = (w_1 S_1 + w_2 S_2) \cdot (1 + \log_2 A),$$

where S_1 and S_2 denote the original BM25 scores between the query and the memory summary and full memory content, respectively, while w_1 and w_2 are their corresponding weights, set to 0.3 and 0.7, respectively. The logarithm of the activity score is used to prevent it from exerting an excessive influence on the final score.

A.3 AMSS ALGORITHM

A.3.1 PARAMETER SOURCES

The parameters of the integer programming problem in AMSS are obtained as follows:

- p_i: Computed as $\frac{\bar{S}_i}{(S_{\max}+h)}$, normalizing the weighted score to the range $(0, 1)$. Here, $S_{\max} = \max(\bar{S}_1, \ldots, \bar{S}_N)$, and h is a smoothing factor introduced to prevent the probability from reaching 1, set to 0.1.

Algorithm 1 Exhaustive Search for AMSS

Require: Number of candidate memories N; objective function $f(\cdot)$
Ensure: Selected memory subset S

```
best_obj ← 0
best_mask ← 0
for mask ← 0 to 2^N − 1 do
    S' ← {i | 0 ≤ i < N, mask & (1 ≪ i) ≠ 0}
    obj ← f(S')
    if obj > best_obj then
        best_obj ← obj
        best_mask ← mask
    end if
end for
S ← {i | 0 ≤ i < N, best_mask & (1 ≪ i) ≠ 0}
return S
```

- v_i: Directly set to $\bar{S}_i$ as the utility of memory m_i.
- l_i, L_i: To avoid the complexity of adapting to the tokenizers of different backbone models, we uniformly use the `o200k_base` tokenizer from `tiktoken` and take the length of the resulting token sequence.
- M: At the end of each conversation turn, the input and output token counts are obtained from the LLM API and accumulated into a session-level variable tracking the total token consumption.

A.3.2 Derivation of Expected Cost

Derivation of P_V. The viewing stage occurs if the agent inspects at least one of the retrieved memories. For each memory m_i, the probability that it does not trigger a `ViewMemory` call is $1 - x_i p_i$. Assuming that these events are independent, the probability that none of the selected memories is viewed is

$$\prod_{i=1}^{N}(1 - x_i p_i).$$

Therefore, the probability that at least one selected memory is viewed is

$$P_V = 1 - \prod_{i=1}^{N}(1 - x_i p_i).$$

Derivation of C. The search stage is always executed, and its input token cost is

$$C_S = M + \sum_{i=1}^{N} x_i l_i.$$

The viewing stage is executed with probability P_V. It reuses the current context and the search results, and additionally includes the full content of each memory that is viewed. Thus, its conditional input token cost given that the viewing stage occurs is

$$C_V = M + \sum_{i=1}^{N} x_i l_i + \sum_{i=1}^{N} x_i p_{i|V} L_i,$$

where $p_{i|V}$ denotes the conditional probability that memory m_i is viewed given that the viewing stage occurs. Hence, the expected total input token cost is

$$\begin{aligned} C &= C_S + P_V C_V \\ &= (1 + P_V)\left(M + \sum_{i=1}^{N} x_i l_i\right) + P_V \sum_{i=1}^{N} x_i p_{i|V} L_i. \end{aligned}$$

Algorithm 2 Greedy Approximation for AMSS

Require: Number of candidate memories N; objective function $f(\cdot)$
Ensure: Selected memory subset S

$S \leftarrow \emptyset$
$current_obj \leftarrow f(S)$
while $|S| < N$ **do**
  $best_add \leftarrow -1$
  $best_obj \leftarrow current_obj$
  for $i \leftarrow 0$ **to** $N - 1$ **do**
    if $i \notin S$ **then**
      $S' \leftarrow S \cup \{i\}$
      $obj \leftarrow f(S')$
      if $obj > best_obj$ **then**
        $best_obj \leftarrow obj$
        $best_add \leftarrow i$
      end if
    end if
  end for
  if $best_add = -1$ **then**
    break
  end if
  $S \leftarrow S \cup \{best_add\}$
  $current_obj \leftarrow best_obj$
end while
return S

By the definition of conditional probability,

$$P_V p_{i|V} = p_i.$$

Therefore,

$$C = (1 + P_V) \left(M + \sum_{i=1}^{N} x_i l_i \right) + \sum_{i=1}^{N} x_i p_i L_i.$$

A.3.3 Problem Solving

When $N \leq 8$, AMSS solves the optimization problem using exhaustive search, as shown in Algorithm 1. Otherwise, it employs a greedy approximation strategy, with the corresponding pseudocode presented in Algorithm 2.

A.4 Pre-Query Middleware

A.4.1 Memory Probing and Prompt Injection

Memory probing computes the intersection between the set of tokens obtained by tokenizing the query and the token sets obtained by tokenizing the tags of all memories. If a non-empty intersection exists, memory retrieval is performed and the following prompt is injected:

```
<partial_memories>
{memories_str}
</partial_memories>
<memory_hint>
The above are partial memory summaries potentially relevant to the
    query. If you need the full content of any listed memory, call the
     ViewMemory tool with its ID. If you need to look for other
    memories not listed above, call the SearchMemory tool to search
    for more. Note: memories already present in the <partial_memories>
     list will not be returned repeatedly by the SearchMemory tool.
```

Algorithm 3 Tag-Based Query Enhancement

Require: Query q; candidate memories $\mathcal{M}$
Ensure: Enhanced query q'

$Q \leftarrow \text{Tokenize}(q)$
if $Q = \emptyset$ **then**
  return q
end if
$T \leftarrow \emptyset$
for each memory $m \in \mathcal{M}$ **do**
  $\mathcal{K} \leftarrow \text{Tags}(m)$
  for each tag $t \in \mathcal{K}$ **do**
    $T_t \leftarrow \text{Tokenize}(t)$
    if $T_t \cap Q \neq \emptyset$ **then**
      $T \leftarrow T \cup \{t\}$
    end if
  end for
end for
if $T = \emptyset$ **then**
  return q
end if
$q' \leftarrow q \parallel \text{Join}(T)$
return q'

```
</memory_hint>
```

Otherwise, the following suppressive instruction is injected:

```
<memory_hint>
Evidence suggests there are likely no memories associated with this
    query, so memory search is likely unnecessary. However, if the
    query clearly asks for unknown personal information, please call
    SearchMemory to search for relevant memories.
</memory_hint>
```

It is worth noting that the prompt used for injection is the original prompt rather than the enhanced prompt, as the enhancement is only meaningful for BM25-based retrieval and does not provide additional benefits for LLMs.

A.4.2 Tag-Based Query Enhancement

The tag-based query enhancement algorithm is presented in Algorithm 3.

A.5 Post-Query Middleware

In the background memory inspection mechanism, we set $K = 5$ to maintain sufficient semantic coherence while preventing memory management omissions caused by providing too much information to the subagent at once. The system prompt of the subagent is as follows:

```
You are an excellent memory management assistant. You can use the
    following tools:
- mcp_memory_SearchMemory: search existing memories and return
    matching memory summaries
- mcp_memory_ViewMemory: view the full content of a memory by its ID
- mcp_memory_UpdateMemory: create, update, or delete memories
```

```
You will be given several user queries (each prefixed with a date) and
     memories already created from them. Determine whether any new
    memories should be created, or whether any existing memories need
    to be updated or deleted.

Important guidelines:
1. When creating memories about events, always include the date or
    time information mentioned in the query (e.g., 'last Sunday', '
    March 15', '2 weeks ago').
2. When a memory's subject changes state, update the existing memory
    to reflect the new state AND briefly note the previous state. For
    example: 'User likes romance movies the most (previously science
    fiction movie).'
```

And the input instruction is shown below:

```
<queries description=The 5 user queries to review>
{queries_str}
</queries>
<extracted_memories description=Memories already extracted from the
    above queries>
{updates_str}
</extracted_memories>
Please carefully review the queries and extracted memories above. The
    extracted memories were created or updated based on these queries
     check if they are complete. If any user/project/business-specific
     information/knowledge/event is missing, use SearchMemory to find
    related memories if needed, and ViewMemory to read full content.
    Then use UpdateMemory to create or update accordingly. Keep your
    response concise.
```

It is worth noting that, in addition to the user queries from the most recent five turns, we also provide the memories that have already been created or updated by the main agent during these five turns, thereby preventing redundant memory creation or updates.

B Experimental Details

B.1 Experimental Setup

For LongMemEval, we exclude the `single-session-assistant` question type, which accounts for 11.2% of the benchmark. For BEAM, we exclude three question types, namely `summarization`, `information_extraction`, and `multi_session_reasoning`, which together account for 30% of the benchmark. After these exclusions, the two benchmarks contain a total of 724 questions.

The input token count is calculated as the total number of input tokens consumed by all model-tool interactions during the QA turns.

B.2 Ablation Study

The replacement strategies used by the three ablation variants after removing their corresponding mechanisms are as follows:

- *w/o PI*: No prompt injection is performed, and the original user query from the benchmark is directly provided to the agent.
- *w/o AMSS*: No additional retrieval scope control is applied, except that the agent optionally uses `top_k` parameter to control the number of returned memories and memories with a BM25 score of 0 are excluded.

- *w/o QE*: No query enhancement is performed, and the original query is directly used to calculate BM25 scores.

B.3 Marginal Returns Analysis

The additional system prompts used by the *high* and *max* variants are as follows:

- *high*: "If the required memory cannot be found, try additional searches."
- *max*: "If the required memory cannot be found, continue searching until you are certain that no relevant memory exists."