GeMS-SC: Runtime State-Conditioned Graph-Memory Invocation for Long-Horizon Agents

Anonymous ACL submission

Abstract

Long-term memory is a key capability for large language model (LLM) agents to complete long-horizon tasks. Existing methods have improved external memory through structured memory organization and memory management. However, during multi-step task execution, agents still lack a step-level mechanism for runtime graph-memory invocation that adapts graph-native memory behavior to the current execution state and hands off memory results to downstream task execution. To address this problem, we propose GeMS-SC, a Graph-Native Memory Skill framework with state-conditioned guidance. GeMS-SC introduces CASE, a Cross-Attentive State-Skill Encoder that models state-skill interactions and produces step-level memory guidance. The resulting state-conditioned guidance is injected into the base model context to coordinate graph-native memory tool calls with subsequent agent decisions. Experiments on LoCoMo, LongMemEval, ALFWorld, and MemoryAgentBench demonstrate the effectiveness of GeMS-SC in state-conditioned graph-memory invocation across long-term language memory and dynamic interaction tasks.

1 Introduction

In long-horizon tasks, large language model (LLM) agents must reuse historical information across multiple decision steps. Such agents often generate actions or tool calls based on the current task state, historical context, and external feedback, and then update subsequent behavior according to newly received observations (Yao et al., 2023). In this process, memory is not merely static context appended before generation. Instead, it shapes state estimation, tool call selection, and the interpretation of new observations. As task horizons grow, simply enlarging the context window is insufficient for reliable evidence recovery, outdated-state correction, and runtime decisions about when and how memory should influence subsequent execution (Packer et al., 2023; Hu et al., 2025b; Wu et al., 2025a).

Existing methods have improved external memory along two directions: structured memory organization and memory management. One line of work studies how memory should be organized, representing long-term history as relational, graph-structured, or hierarchical structures to improve memory representation, evidence retrieval, and state tracking (Xu et al., 2025; Zhang et al., 2025a; Rasmussen et al., 2025; Anokhin et al., 2024; Gutiérrez et al., 2025; Wu et al., 2025c; Li et al., 2025; Lei et al., 2025; Xu et al., 2026; Sun et al., 2026; Salama et al., 2025). Another line of work studies how memory should be managed, formulating memory addition, update, deletion, retrieval, or skill-based maintenance as callable or learnable memory management mechanisms (Packer et al., 2023; Chhikara et al., 2025; Yan et al., 2025; Zhang et al., 2026; Tan et al., 2025; Zhang et al., 2025b). These studies show that long-term memory is not only a storage problem, but also an external state that can be represented, accessed, updated, and reused during agent execution.

However, these advances still leave the runtime invocation of graph-structured memory under-characterized in multi-step agent execution. When external memory is organized through nodes, edges, and relational paths, the agent's decision at each step is no longer simply whether to retrieve or update an individual memory entry. Instead, it must decide how to operate over structural objects in the graph according to the current execution state, and how to hand off the returned local graph results to subsequent execution. A step may require tracing an evidence subgraph from entity or event anchors, following update or replacement edges to identify the currently valid state, maintaining invalidated relations and provenance within a conflict subgraph, or compressing an oversized local subgraph into a structured view that can be injected into con-

text. These graph-native behaviors involve different graph traversal patterns, relation-update procedures, and result organization formats, and therefore affect downstream reasoning, memory updates, tool-argument construction, and task/environment actions. Since graph-memory tool calls are embedded in multi-step execution rather than performed as an isolated pre-generation retrieval stage, an inappropriate invocation may cause the agent to propagate outdated states along incorrect relational paths, miss critical support chains, or carry unresolved conflicts into later actions and memory updates. Multi-step agents therefore require a step-level mechanism for runtime graph-memory invocation: given the current subtask, the previous observation, and the recent execution trajectory, the mechanism selects appropriate graph-native memory guidance and coordinates graph-memory tool calls with downstream reasoning, answer generation, memory updates, or task/environment actions.

To address this problem, we propose GeMS-SC, a Graph-Native Memory Skill framework with state-conditioned guidance for runtime memory invocation in long-horizon agents. GeMS-SC introduces CASE, a Cross-Attentive State-Skill Encoder, to model interactions between execution states and candidate Graph-Native Memory Skills. Rather than retrieving a skill by static embedding similarity, CASE jointly contextualizes execution-state fragments and skill descriptions, and produces step-level memory guidance through Skill-to-State Cross-Attention. The selected guidance is injected as a concise prompt for the next decision step.

Our main contributions are as follows:

- We identify runtime graph-memory invocation as a step-level control problem in long-horizon agent execution. GeMS-SC addresses this problem by coordinating graph-native memory tool calls with downstream reasoning, answer generation, memory updates, and task/environment actions according to the current execution state.

- We introduce CASE, a Cross-Attentive State-Skill Encoder for step-level memory guidance selection. Rather than retrieving skills by static embedding similarity, CASE jointly contextualizes execution-state fragments and Graph-Native Memory Skill descriptions, and produces state-conditioned skill representations through Skill-to-State Cross-Attention.

- We evaluate GeMS-SC on LoCoMo, LongMemEval, ALFWorld, and MemoryAgentBench. Experimental results demonstrate the effectiveness of GeMS-SC in state-conditioned graph-memory invocation across long-term language memory and dynamic interaction tasks.

2 Related Work

2.1 Structured Memory Organization

External memory enables LLM agents to store, access, and maintain historical information over long-horizon interactions (Hu et al., 2025b; Salama et al., 2025). Early systems usually represent memory as textual entries, user facts, event records, or summary fragments. MemoryBank maintains user facts and historical events in long-term dialogue scenarios (Zhong et al., 2024). Memory OS organizes short-term, mid-term, and long-term memory along the temporal dimension, enabling information with different levels of stability to flow and be reused within the system (Kang et al., 2025; Ong et al., 2025). These methods provide useful external memory capabilities, but textual entries and summary fragments are often insufficient to explicitly represent dependencies, state changes, and evidence chains in long-horizon interactions (Anokhin et al., 2024; Rasmussen et al., 2025; Lei et al., 2025).

Recent studies therefore introduce structured, graph-based, and hierarchical memory representations (Xu et al., 2025; Zhang et al., 2025a; Rasmussen et al., 2025; Anokhin et al., 2024; Gutiérrez et al., 2025; Wu et al., 2025c; Li et al., 2025; Lei et al., 2025; Xu et al., 2026; Sun et al., 2026). A-MEM dynamically builds associations among memory entries, organizing discrete historical fragments into an evolving relational network (Xu et al., 2025). G-Memory divides long-term interaction history into an interaction graph, a query graph, and an insight graph, so that raw evidence, task states, and high-level experiences can be represented at different levels (Zhang et al., 2025a). StructMem transforms historical interactions into structured memory units to support more stable retrieval, update, and state maintenance (Xu et al., 2026). H-MEM adopts a hierarchical memory architecture and introduces index mechanisms among sub-memories at different granularities, improving long-horizon retrieval and cross-granularity access efficiency (Sun et al., 2026). These works im-

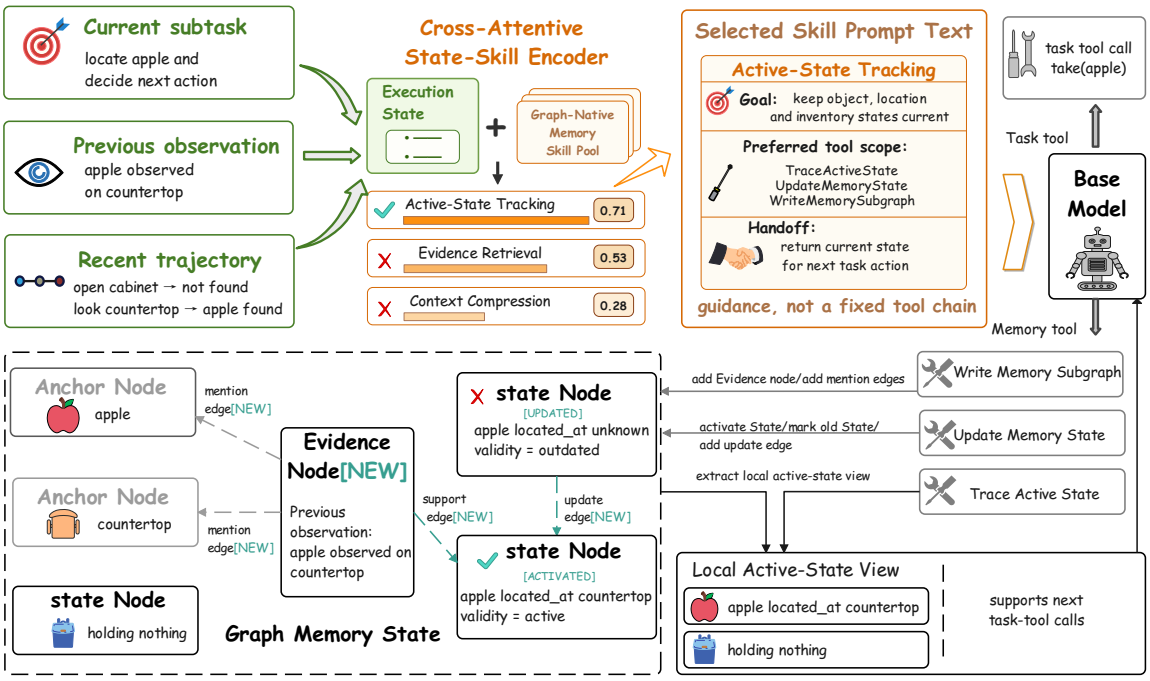

Figure 1: Overview of GeMS-SC. Given the execution state, CASE models state-skill interactions and injects the resulting guidance. The base agent still generates concrete actions. In the illustrated ALFWorld step, ACTIVE-STATE TRACKING guides memory tool calls that add an evidence node, create typed edges, update an object-location state, and return a local active-state view for the next task-tool call.

prove how external memory is represented and accessed, whereas GeMS-SC focuses on how graph-structured memory is invoked during step-by-step agent execution.

2.2 Memory Management and Control

Memory management and control focuses on how models read, write, update, and regulate external memory during or around agent execution (Hu et al., 2025b; Zhang et al., 2025b; Hu et al., 2025a). MemGPT makes memory reading and writing explicit as callable operations, allowing models to move information between limited context and external storage (Packer et al., 2023). Mem0 provides a long-term memory management interface that supports adding, updating, retrieving, and deleting memory entries (Chhikara et al., 2025). Memory-R1 formulates external memory management as a learnable operation selection problem, enabling models to choose actions such as ADD, UPDATE, DELETE, and NOOP according to task feedback (Yan et al., 2025). MemSkill further encapsulates memory generation and maintenance processes into reusable and evolving memory skills, and uses a controller to select appropriate skills for memory construction and revision (Zhang et al., 2026).

These works show that long-term memory capability depends not only on what is stored in external memory, but also on whether the model can decide when and how to operate on memory (Tan et al., 2025; Hu et al., 2025a; Zhang et al., 2025b; Kang et al., 2025; Zhang et al., 2025c; Yu et al., 2025; Yuan et al., 2025; Wu et al., 2025b). GeMS-SC shares this motivation, but the key distinction lies in the unit of control. Memory-R1 and MemSkill mainly organize control around memory entries, operation labels, or reusable memory-skill templates. GeMS-SC instead selects graph-native memory behaviors whose inputs and outputs are defined over typed graph objects, such as evidence subgraphs, update paths, support chains, conflict relations, and compact graph views. This control granularity matters in multi-step execution because the returned graph-structured results can condition later reasoning, memory updates, tool-argument construction, and task/environment actions.

3 Method

3.1 State-Conditioned Graph-Memory Invocation Guidance

We study runtime invocation guidance for graph-structured external memory in multi-step agent execution. At each decision step, the base agent ob-

serves the current task context and generates the next action, which may be a task tool call, a memory tool call, or a final answer. GeMS-SC does not modify the base model or replace its action generation process. Instead, it derives step-level guidance from Graph-Native Memory Skills according to the current execution state and injects the resulting guidance into the current context.

Let the execution state at step t be

$$x_t = (q_t, o_{t-1}, h_{t-L:t-1}), \tag{1}$$

where q_t is the current subtask, o_{t-1} is the previous observation returned by tools or the environment, and $h_{t-L:t-1}$ is the recent execution trajectory within a window of length L. GeMS-SC maintains a candidate skill set

$$\mathcal{S} = \{S_0, S_1, \ldots, S_K\}, \tag{2}$$

where S_0 denotes CONTINUE REGULAR EXECUTION, i.e., no additional memory guidance is injected. The remaining candidates are Graph-Native Memory Skills that provide compositional guidance for graph-structured memory operations.

GeMS-SC learns a lightweight state-skill scoring model

$$\pi_\psi(S_k \mid x_t, \mathcal{S}), \tag{3}$$

and identifies the skill with the highest score:

$$S_t^* = \arg\max_{S_k \in \mathcal{S}} \pi_\psi(S_k \mid x_t, \mathcal{S}). \tag{4}$$

The guidance associated with the highest-scoring skill is then injected as

$$\tilde{c}_t = \begin{cases} c_t, & S_t^* = S_0, \\ \left[\mathrm{text}(S_t^*); c_t\right], & S_t^* \neq S_0. \end{cases} \tag{5}$$

The base model continues action generation under $\tilde{c}_t$. Thus, GeMS-SC controls when and how graph-memory guidance is provided, while concrete memory tool calls, task tool calls, and final actions remain generated by the base model.

3.2 CASE: Cross-Attentive State-Skill Interaction

To model state-skill interactions for step-level guidance, we introduce CASE, a Cross-Attentive State-Skill Encoder. Given the execution state x_t and the candidate skill set $\mathcal{S}$, CASE outputs a distribution over candidate skills:

$$\pi_\psi(S_k \mid x_t, \mathcal{S}). \tag{6}$$

The execution state contains three complementary signals: the current subtask q_t, the previous observation o_{t-1}, and the recent trajectory $h_{t-L:t-1}$. These signals respectively describe the current intention, the latest external feedback, and the local execution context. Candidate Graph-Native Memory Skills describe possible memory invocation strategies for the current decision step.

Figure 2 illustrates the architecture of CASE. CASE first encodes execution-state fragments and candidate skill descriptions into fragment-level semantic representations. Type embeddings distinguish the current subtask, previous observation, recent trajectory fragments, and candidate skills, while slot embeddings preserve trajectory positions and candidate-skill identities. A lightweight Transformer encoder then jointly contextualizes all fragments, producing state representations H_t^{state} and skill representations H_t^{skill}. Details of fragment encoding, projection layers, and embedding settings are provided in the appendix.

The central design of CASE is Skill-to-State Cross-Attention. Instead of matching each candidate skill only against the task text, CASE lets each skill explicitly attend to the current execution state. Specifically, skill representations are used as queries, while state representations are used as keys and values:

$$\begin{aligned} Z_t^{\text{skill}} &= \mathrm{CrossAttn}(Q_t, K_t, V_t), \\ Q_t = H_t^{\text{skill}}, &\quad K_t = V_t = H_t^{\text{state}}. \end{aligned} \tag{7}$$

The resulting Z_t^{skill} represents candidate skills after reading the current execution state. For each candidate skill S_k, a shared scoring head computes its compatibility score:

$$r_{t,k} = \mathrm{MLP}_{\text{score}}\left(Z_{t,k}^{\text{skill}}\right). \tag{8}$$

The final selection distribution is obtained by normalizing scores over all candidate skills:

$$\pi_\psi(S_k \mid x_t, \mathcal{S}) = \frac{\exp(r_{t,k})}{\sum_{S_j \in \mathcal{S}} \exp(r_{t,j})}. \tag{9}$$

This design allows the same Graph-Native Memory Skill to receive different scores at different decision steps, depending on the current subtask, the previous observation, and the recent trajectory. CASE therefore provides step-level state-conditioned guidance by modeling interactions between candidate skills and the current execution state, rather than relying on a fixed rule, a static prompt, or a task-level router.

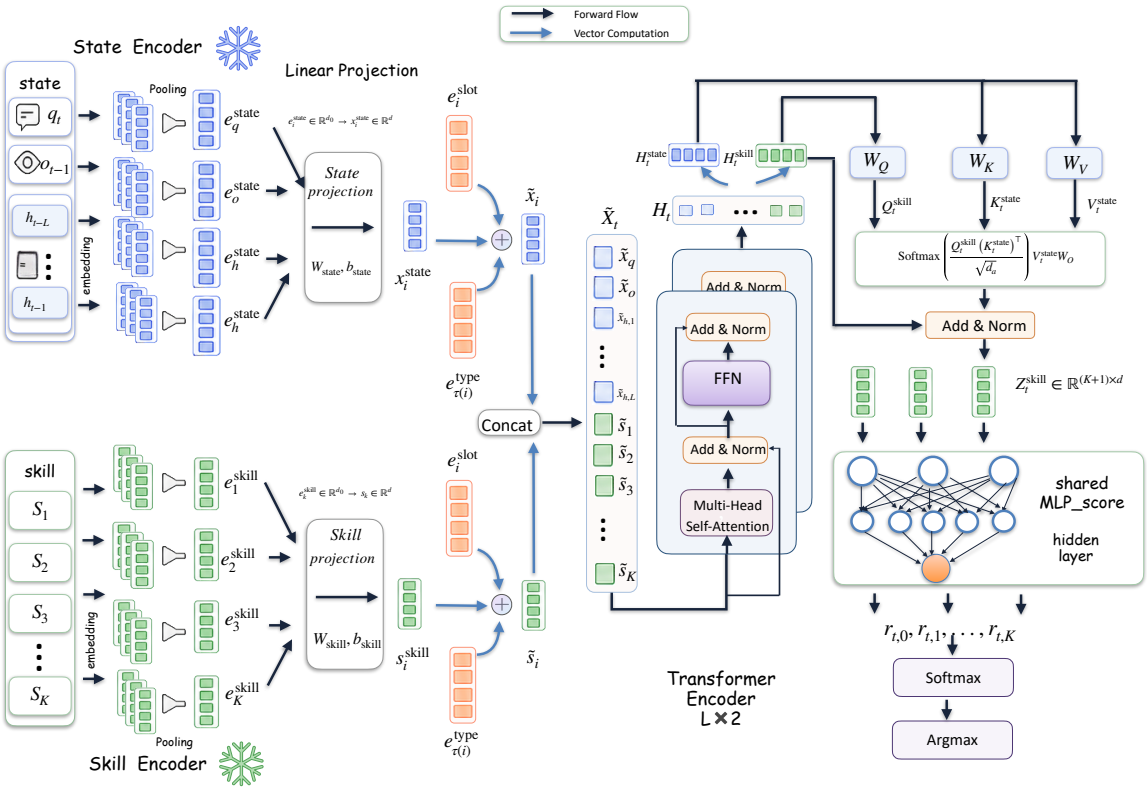

Figure 2: Architecture of CASE. CASE encodes the current execution state and candidate Graph-Native Memory Skills as fragment-level representations, performs joint state-skill contextualization, and uses Skill-to-State Cross-Attention to produce state-conditioned skill representations for step-level guidance modeling.

3.3 Graph-Native Memory Skills

Graph-Native Memory Skills define the candidate guidance space used by CASE. At each decision step, CASE scores the null skill S_0 and all non-null skills in the candidate set $\mathcal{S}$, and activates the guidance associated with the highest-scoring skill. The activated guidance serves as a local conditioning signal for the next action generation: it guides how the base model should invoke graph-native memory tools and how the returned memory results should be used in subsequent reasoning, answer generation, memory operations, or task/environment actions.

We represent the k-th skill as

$$S_k = (n_k, m_k, p_k),$$

where n_k is the skill name, m_k is the state-matching description encoded by CASE, and p_k is the execution prompt injected into the base model context when its guidance is activated. The state-matching description specifies when the skill is applicable, while the execution prompt provides the concise guidance injected after selection.

A Graph-Native Memory Skill is not an executable tool or a fixed tool chain. It specifies an operation protocol for the current decision step, including the memory goal, preferred graph-native tool scope, argument-construction cues, output handoff, and stopping condition.

This design separates guidance modeling from concrete tool execution. CASE only estimates which memory invocation guidance is appropriate for the current execution state, while graph-native memory tools carry out the actual memory state transition over the external graph memory. The null skill indicates that no additional memory guidance is needed at the current step. Non-null skills cover recurring memory demands in long-horizon execution, such as grounded evidence retrieval, active-state tracking, episodic state writing, conflict-aware maintenance, context selection and compression, pattern abstraction, and answer-and-store consolidation. The complete skill fields, candidate skill set, and a representative guidance prompt are provided in Appendix D.

3.4 Two-Stage Learning of CASE

CASE is trained to match execution states with appropriate Graph-Native Memory Skills, while the base model parameters remain fixed.

Stage 1: State-Skill Alignment Warm-up. This stage uses weak supervision to initialize the correspondence between execution states and candidate skills. For each decision step, we construct a weak skill label $\hat{S}_t$ from the memory tool calls observed in a short subsequent window, and train CASE with a cross-entropy objective:

$$\mathcal{L}_{\mathrm{BC}} = -E_{(X_t,\hat{S}_t)\in\mathcal{D}_{\mathrm{bc}}} \log \pi_\psi(\hat{S}_t \mid X_t, \mathcal{S}). \quad (10)$$

Here, X_t denotes the CASE input at the current decision step, $\mathcal{S}$ denotes the candidate skill set, and $\pi_\psi(\cdot \mid X_t, \mathcal{S})$ denotes the skill selection distribution produced by CASE. This stage provides CASE with an initial state-skill matching ability.

Stage 2: Trajectory-Feedback Skill Optimization. This stage further adjusts the state-skill matching preference of CASE using feedback from complete execution trajectories. Given the i-th execution trajectory Γ_i, we compute a trajectory-level composite feedback:

$$\begin{aligned} R_i = \lambda_1 R_i^{\mathrm{task}} + \lambda_2 R_i^{\mathrm{align}} \\ + \lambda_3 R_i^{\mathrm{memory}} + \lambda_4 R_i^{\mathrm{eff}}. \end{aligned} \quad (11)$$

Here, $\lambda_1, \ldots, \lambda_4$ are normalized weights. R_i^{task} measures task completion quality, R_i^{align} measures the consistency between activated guidance and the actual memory tool calls, R_i^{memory} measures the quality of memory operations, and R_i^{eff} measures tool-call overhead and context overhead. The detailed computation of feedback components, weight settings, and training hyperparameters are provided in the appendix.

Within each mini-batch $\mathcal{B}$, the trajectory-level feedback is normalized into a trajectory-level advantage:

$$A_i = \frac{R_i - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \epsilon}, \quad (12)$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ denote the mean and standard deviation of trajectory feedback within the current mini-batch, respectively. All decision steps in the same trajectory share this trajectory-level advantage.

CASE is updated with an advantage-weighted objective. Let $\bar{\ell}_i(\psi) = \frac{1}{T_i+1}\sum_{t=0}^{T_i} \log \pi_\psi(S_{i,t} \mid X_{i,t}, \mathcal{S})$ denote the trajectory-level average log-probability of the sampled skills.

$$\begin{aligned} \mathcal{L}_{\mathrm{CASE}} = & -E_{\Gamma_i\in\mathcal{B}}\big[A_i\, \bar{\ell}_i(\psi)\big] \\ & - \lambda_{\mathrm{ent}}\mathcal{H}_{\mathrm{ent}} + \lambda_{\mathrm{reg}}\|\psi\|_2^2. \end{aligned} \quad (13)$$

Here, $\mathcal{H}_{\mathrm{ent}}$ denotes the average entropy of the skill selection distribution within the mini-batch, and λ_{ent} and λ_{reg} control entropy regularization and parameter regularization, respectively. This objective adjusts the selection probabilities of candidate skills under different execution states according to trajectory feedback, aligning CASE's state-skill matching preference with task completion, consistency between activated guidance and memory tool calls, memory-operation quality, and execution overhead.

4 Experiments

4.1 Experimental Setup

Benchmarks and metrics. We evaluate GeMS-SC on two long-term language memory benchmarks and two dynamic interaction benchmarks. LoCoMo (Maharana et al., 2024) evaluates evidence recovery and multi-hop information integration over long dialogue histories. We report L-J scores for Single-hop, Multi-hop, Temporal, and Open-domain questions, as well as the weighted Overall score. LongMemEval (Wu et al., 2025a) focuses on cross-session long-term memory, including user fact retention, information updates, and conflict handling; we report its overall L-J score. For dynamic interaction, ALFWorld (Shridhar et al., 2021) requires agents to track environment states during multi-step action execution, and we report Seen, Unseen, and average Success Rate. MemoryAgentBench (Hu et al., 2026) evaluates incremental multi-turn agent memory, and we report Accurate Retrieval (AR), Test-Time Learning (TTL), Long-Range Understanding (LRU), Selective Forgetting (SF), and Overall.

Baselines. We compare GeMS-SC with representative long-term memory and memory management baselines: ReAct (Yao et al., 2023), MemoryBank (Zhong et al., 2024), A-MEM (Xu et al., 2025), Mem0 and Mem0-Graph (Chhikara et al., 2025), H-MEM (Sun et al., 2026), StructMem (Xu et al., 2026), Memory-R1 (Yan et al., 2025), and MemSkill (Zhang et al., 2026). These methods cover agents without explicit long-term memory, textual memory management, graph-enhanced memory, hierarchical memory, structured memory representation, learnable memory operation selection, and skill-based memory management.

Common evaluation setting. For the main results, we use Qwen3-8B-Instruct (Qwen Team,

Table 1: Main results on long-term memory and dynamic interaction benchmarks with Qwen3-8B-Instruct as the base model.

Method	Long-Term Language Memory						Dynamic Interaction and Memory Maintenance							
	LoCoMo L-J↑					LongMemEval L-J↑	ALFWorld SR↑			MemoryAgentBench↑				
	Single	Multi	Temporal	Open	Overall		Seen	Unseen	Avg.	AR	TTL	LRU	SF	Overall
ReAct	38.6	30.4	27.8	35.2	34.6	34.1	31.6	25.4	28.5	30.8	25.9	23.6	9.7	22.5
MemoryBank	41.1	32.6	29.5	37.8	36.9	36.7	34.2	28.1	31.2	32.4	26.6	21.8	12.4	23.3
A-MEM	43.2	34.8	31.6	39.5	39.0	39.0	35.8	29.6	32.7	34.0	28.2	23.7	13.5	24.9
Mem0	45.8	36.5	33.7	41.2	41.3	42.4	38.4	32.5	35.5	31.8	24.5	20.4	11.2	22.0
Mem0-Graph	47.1	38.2	35.4	42.8	42.8	44.1	40.1	34.0	37.1	37.5	29.6	25.8	14.7	26.9
H-MEM	48.9	39.5	37.1	43.4	44.4	45.0	41.6	35.8	38.7	39.2	31.0	29.4	16.8	29.1
StructMem	**50.6**	42.1	**43.0**	45.2	**47.1**	47.3	43.4	37.5	40.5	42.0	32.5	33.8	19.4	31.9
Memory-R1	49.2	**44.0**	41.2	46.0	46.4	50.2	42.8	36.7	39.8	41.2	34.8	35.0	20.8	33.0
MemSkill	49.2	40.1	37.6	43.8	44.8	52.1	**46.5**	39.4	43.0	43.5	**37.4**	34.6	21.9	34.4
GeMS-SC	50.1	43.5	41.7	**46.8**	46.9	**52.8**	45.6	**41.1**	**43.4**	**49.8**	36.8	**36.1**	**25.6**	**37.1**

Note: Bold indicates the best result, and underline indicates the second-best result. Tied best results are all bolded. Single, Multi, Temporal, Open, and Overall denote the four LoCoMo categories and the weighted overall score. ALFWorld reports Seen, Unseen, and average success rates. MemoryAgentBench reports Accurate Retrieval (AR), Test-Time Learning (TTL), Long-Range Understanding (LRU), Selective Forgetting (SF), and Overall.

2025) as the base model for all methods. We keep the test split, task input, maximum reasoning steps, decoding setting, and evaluation protocol consistent whenever the original method permits. GeMS-SC only trains the CASE module and does not update the base model parameters. Full results under DeepSeek-V3.2 (DeepSeek-AI, 2025), baseline adaptation details, hyperparameters, and the L-J evaluation protocol are provided in the appendix.

4.2 Main Results

Table 1 reports the main results under the unified Qwen3-8B-Instruct setting. Overall, GeMS-SC obtains the best numerical results on LongMemEval, ALFWorld Avg. SR, and MemoryAgentBench Overall, while remaining second-best and close to StructMem on LoCoMo Overall.

On LoCoMo, StructMem slightly outperforms GeMS-SC on Overall and Temporal questions, suggesting that explicit event-structured memory remains effective for temporal reasoning in static long-context memory QA. In contrast, GeMS-SC obtains the best numerical results on Open-domain questions and LongMemEval. Open-domain questions often require locating open-ended evidence scattered across long histories, while LongMemEval emphasizes cross-session state maintenance, information updates, and conflict handling. These results suggest that state-conditioned graph-memory invocation helps open-ended historical evidence localization and cross-session memory maintenance.

Table 2: Ablation results on long-term memory and dynamic interaction tasks with Qwen3-8B-Instruct as the base model.

Variant	LoCoMo Overall	LongMem Eval	ALFWorld Avg.	MAB Overall
Full Model	**46.9**	**52.8**	**43.4**	**37.1**
Graph-CRUD Interface	35.3	39.6	31.8	25.2
w/o Skill Guidance	37.2	41.2	35.6	28.2
w/o CASE Selection	38.2	40.6	36.3	29.1
w/o CASE Learning	40.7	44.3	39.5	32.4
w/o Skill-to-State Cross-Attention	39.9	43.2	38.5	31.6

On ALFWorld and MemoryAgentBench, GeMS-SC obtains the best numerical Avg. SR and Overall scores, respectively, and performs strongly on Unseen SR, AR, LRU, and SF. This pattern is consistent with the role of runtime graph-memory invocation in dynamic multi-step execution, where agents must maintain states and incorporate new observations. In ALFWorld, this appears as memory-guided environment actions, while in MemoryAgentBench it appears as incremental writing, retrieval, and maintenance that support later query answering. MemSkill remains strong on Seen SR and TTL, suggesting that MemSkill's skill-based memory construction and GeMS-SC's runtime guidance over graph-native memory tools can be complementary in some settings.

Method	Tok.	Ratio	LLM	Mem.	L-J
ReAct	24.8k	100.0	2.6	0.0	34.6
A-MEM	3.6k	14.5	2.8	1.8	39.0
Mem0	2.1k	8.5	2.9	1.6	41.3
Mem0-Graph	4.8k	19.4	3.1	2.1	42.8
Memory-R1	3.9k	15.7	3.4	2.5	46.4
MemSkill	5.4k	21.8	3.7	3.0	44.8
GeMS-SC	3.2k	12.9	3.0	2.2	46.9

Table 3: Efficiency and context overhead on LoCoMo with Qwen3-8B-Instruct. Tok. denotes average online context tokens injected during inference; Ratio is normalized by ReAct; LLM and Mem. denote average LLM and memory tool calls per instance; L-J is the LoCoMo Overall score.

4.3 Ablation Studies

Table 2 reports the ablation results on both long-term language memory and dynamic interaction tasks. Across all four summary metrics, the full model consistently performs best, indicating that the benefits of GeMS-SC are not limited to static long-context memory QA, but also extend to interactive settings where the agent must update and reuse memory during multi-step execution.

Replacing graph-native memory tools with the Graph-CRUD Interface leads to the largest performance drop across tasks. This variant preserves the same graph memory state, but replaces graph-native operations such as evidence-subgraph retrieval, active-state tracing, support-path recovery, conflict resolution, and memory state maintenance with generic operations that directly edit individual nodes and edges. The drop is especially visible on ALFWorld and MemoryAgentBench, suggesting that dynamic interaction requires memory operations that expose state changes, support paths, and maintenance behavior, rather than only direct edits to nodes and edges.

The variants related to skill guidance and CASE further show that both the guidance space and state-conditioned selection are important. Removing runtime skill guidance substantially degrades performance, even though the base model can still access the same graph-native memory tools. Replacing CASE with static skill guidance also underperforms the full model, suggesting that a fixed memory prompt cannot adapt invocation behavior to the current subtask, previous observation, and recent trajectory.

Finally, removing CASE learning or Skill-to-State Cross-Attention weakens performance on both language memory and dynamic interaction tasks. These results support the view that CASE benefits from trajectory-based learning signals and explicit state-skill interaction. Overall, the ablation results indicate that GeMS-SC depends on the combination of graph-native memory tools, Graph-Native Memory Skills, and state-conditioned guidance modeling, rather than merely exposing additional memory tools or injecting fixed skill instructions. Complete category-level ablation results and detailed variant descriptions are provided in Appendix A.2.

4.4 Efficiency and Context Overhead

We analyze efficiency on LoCoMo with Qwen3-8B-Instruct to examine whether GeMS-SC's gains come from longer contexts or more frequent memory invocation. Table 3 reports average online context tokens and average LLM and memory tool calls per instance.

GeMS-SC uses 3.2k context tokens on average, corresponding to 12.9% of the ReAct context footprint, and performs 2.2 memory tool calls per instance. It remains more context-efficient than Mem0-Graph and MemSkill while achieving a higher LoCoMo Overall score, suggesting that its gains are not mainly explained by longer contexts or repeated memory invocation.

5 Conclusion

We presented GeMS-SC, a runtime graph-memory invocation framework for long-horizon LLM agents with graph-structured external memory. GeMS-SC treats graph-memory invocation during multi-step execution as a step-level control problem. Through CASE, the framework models interactions between the current execution state and Graph-Native Memory Skills, and derives concise guidance for coordinating graph-native memory tool calls with downstream task execution.

Experiments on LoCoMo, LongMemEval, ALFWorld, and MemoryAgentBench show the effectiveness of GeMS-SC across long-term language memory and dynamic interaction settings. Ablations further separate the effects of graph-native tools, skill guidance, and state-conditioned selection. These results suggest that long-term memory for LLM agents should be studied not only as representation or entry-level management, but also as runtime graph-memory invocation during agent execution.

Limitations

This paper focuses on runtime guidance for graph-based external memory, but several limitations remain. First, the current memory skill set is manually designed and may need automatic discovery or adaptation in more open-ended environments. Second, CASE is intended as a lightweight state-skill interaction module, and its effectiveness may depend on the diversity and quality of training trajectories. Third, the evaluation focuses on existing long-horizon memory and interactive benchmarks, which may not fully capture privacy, safety, and long-term deployment concerns in real applications.

Ethical considerations

This work studies memory tool invocation in long-horizon language model agents. Because long-term memory systems may store user preferences, interaction histories, environment observations, or task-specific states, they raise privacy, consent, retention, and misuse concerns (Wang et al., 2025). In practical deployments, memory writing and retrieval should be governed by explicit user consent, data minimization, access control, retention limits, and mechanisms for user inspection and deletion. The proposed graph-native memory tools are intended to improve controllability and traceability of memory tool invocation, but they could also make it easier to persist or retrieve sensitive information if deployed without safeguards. Experiments should therefore use datasets in accordance with their licenses and intended use, avoid storing personally identifying information beyond what is necessary for the benchmark, and report any human annotation, user data, or AI-assisted procedures required by the Responsible NLP Research Checklist.

References

Alexey Anokhin, Nikita Semenov, Konstantin Sorokin, Dmytro Evseev, Natalia Vassilieva, and Evgeny Burnaev. 2024. AriGraph: Learning knowledge graph world models with episodic memory for LLM agents. *arXiv preprint arXiv:2407.04363*.

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

DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. *arXiv preprint arXiv:2512.02556*.

Bernal Jiménez Gutiérrez, Seonmin Yoon, Sohoon Suh, and Yu-Xiong Xiong. 2025. From RAG to memory: Non-parametric continual learning for large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 20415–20436. PMLR.

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

Yuanzhe Hu, Yu Wang, and Julian McAuley. 2026. Evaluating memory in LLM agents via incremental multi-turn interactions. In *International Conference on Learning Representations*. ICLR 2026 Poster.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, and 28 others. 2025b. Memory in the age of AI agents. *arXiv preprint arXiv:2512.13564*.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. 2025. Memory OS of AI Agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25961–25970, Suzhou, China. Association for Computational Linguistics.

Xiang Lei, Qin Li, and Min Zhang. 2025. D-SMART: Enhancing LLM dialogue consistency via dynamic structured memory and reasoning tree. *arXiv preprint arXiv:2510.13363*.

Wen Li, Yinghui Li, Yao Sun, Shuchang Si, Yulin Chen, Qunxuan Huang, Tianxing Zhang, Xintong Liu, Xiaoming Si, Yuhao Zhang, and Zhenjiang Zhang. 2025. CAM: A large language model-based continuous agentic memory framework. *arXiv preprint arXiv:2510.05520*.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13851–13870, Bangkok, Thailand. Association for Computational Linguistics.

Kai Tzu-iunn Ong, Namyoung Kim, Minju Gwak, Hyungjoo Chae, Taeyoon Kwon, Yohan Jo, Seungwon Hwang, Dongha Lee, and Jinyoung Yeo. 2025. Towards lifelong dialogue agents via timeline-based memory management. 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), pages 8631–8661, Albuquerque, New Mexico. Association for Computational Linguistics.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*.

Qwen Team. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*.

Preston Rasmussen, Gregory Karypis, Kartik Talamadupula, and Subbarao Kambhampati. 2025. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*.

Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. 2025. MemInsight: Autonomous memory augmentation for LLM agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33136–33152, Suzhou, China. Association for Computational Linguistics.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*.

Haoran Sun, Shaoning Zeng, and Bob Zhang. 2026. H-MEM: Hierarchical Memory for High-Efficiency Long-Term Reasoning in LLM Agents. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 341–350, Rabat, Morocco. Association for Computational Linguistics.

Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long T. Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, Anand Rajan Iyer, Tianlong Chen, Huan Liu, Chen-Yu Lee, and Tomas Pfister. 2025. 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)*, pages 8416–8439, Vienna, Austria. Association for Computational Linguistics.

Bo Wang, Weiyi He, Shenglai Zeng, Zhen Xiang, Yue Xing, Jiliang Tang, and Pengfei He. 2025. Unveiling privacy risks in LLM agent memory. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 25241–25260, Vienna, Austria. Association for Computational Linguistics.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. 2025a. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*.

Xilai Wu, Yizhe Chen, Chuanyang Pan, Yizhe Kang, Panfeng Hu, Yulin Wang, Xinfeng Ma, Yulin Peng, Zhichen Liu, Mingjie Li, Tong Lu, Tianxing Li, Tianyang Zhang, Hanwei Xu, Jiaqi Li, Qiang Li, Zhe Wang, Yuchen Shao, Ruihua Song, and 2 others. 2025b. ReSum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*.

Yiming Wu, Yanfang Yang, Xingfeng Li, and Yangqiu Song. 2025c. SGMem: Sentence graph memory for long-term conversational agents. *arXiv preprint arXiv:2509.21212*.

Buqiang Xu, Yijun Chen, Jizhan Fang, Ruobin Zhong, Yunzhi Yao, Yuqi Zhu, Lun Du, and Shumin Deng. 2026. StructMem: Structured Memory for Long-Horizon Behavior in LLMs. *arXiv preprint arXiv:2604.21748*.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*. NeurIPS 2025 Poster.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Hinrich Schütze, Volker Tresp, and Yunpu Ma. 2025. Memory-R1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. *arXiv preprint arXiv:2508.19828*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Zhe Yu, Zheng Cheng, Jiaxin Shi, Runlin Liu, Junyi Li, Zheng Chen, Jiasheng Jiang, Dong Lin, Xiyao Huang, Fangcheng Xiao, Di Hu, Kai Zheng, Changyang He, Jiaqi Han, Quanshi Yao, Yu Qiao, Yelong Shen, Xin Zhang, and Jianwei Liu. 2025. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent. *arXiv preprint arXiv:2507.02259*.

Yifu Yuan, Di Wang, Meghanath Thakkar, Zhengjie Miao, Ye Ri, Xiaohan Xu, Yuyang Wang, Tong Luo, and Srinivasan Iyer. 2025. MemSearcher: Training LLMs to reason, search and manage memory via end-to-end reinforcement learning. *arXiv preprint arXiv:2511.02805*.

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. 2025a. G-Memory: Tracing hierarchical memory for multi-agent systems. *arXiv preprint arXiv:2506.07398*.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. 2026. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong

Wen. 2025b. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*.

Zhuosheng Zhang, Yun Jia, Xijia Chen, Xiaobao Long, Xiaohan Wang, Yuanhao Qiu, Wee Sun Lee, See-Kiong Ng, and Minlie Huang. 2025c. MemGen: Weaving generative latent memory for self-evolving agents. *arXiv preprint arXiv:2509.24704*.

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

A Supplementary Experimental Results

A.1 Complete Results under DeepSeek-V3.2

This section reports complete results under DeepSeek-V3.2. Memory-R1 is omitted in this setting because its original pipeline requires policy optimization over trainable model components.

Method	LoCoMo					LongMem Eval
	Single	Multi	Temp.	Open	Overall	
ReAct	62.4	51.8	47.5	58.6	57.1	50.6
MemoryBank	66.5	55.2	50.8	62.4	60.9	53.8
A-MEM	70.8	59.6	54.3	65.9	65.0	57.4
Mem0	76.1	66.5	60.2	70.8	70.7	62.8
Mem0-Graph	78.4	68.7	63.1	72.0	73.0	64.2
H-MEM	81.0	72.1	65.8	**73.6**	75.7	65.1
StructMem	84.2	78.1	**80.4**	72.1	**81.5**	67.7
MemSkill	80.2	70.5	65.1	72.9	74.8	66.5
GeMS-SC	**85.0**	**78.9**	77.2	73.4	**81.5**	**69.1**

Table 4: Complete long-term language memory results under DeepSeek-V3.2. Bold indicates the best result, and underline indicates the second-best result. Tied best results are all bolded.

Table 4 shows the complete long-term language memory results under DeepSeek-V3.2. Compared with the Qwen3-8B-Instruct setting in the main text, all methods obtain higher absolute scores, suggesting that the stronger base model already provides better long-context evidence integration ability. On top of this stronger base model, GeMS-SC achieves the best results on Single-hop, Multi-hop, and LongMemEval, ties with StructMem on LoCoMo Overall, and remains second-best on Open-domain. StructMem performs best on Temporal questions, indicating that explicit event structures and temporal relation organization remain useful for temporal reasoning questions. Overall, GeMS-SC remains competitive under the stronger base model, with clearer advantages in multi-hop information integration and cross-session memory maintenance.

Table 5 shows the complete dynamic interaction and memory maintenance results under DeepSeek-V3.2. On ALFWorld, GeMS-SC achieves the best Unseen SR and Avg. SR, while MemSkill obtains the best Seen SR. This suggests that GeMS-SC is more effective in unseen environments where the agent needs to track evolving states and reuse historical observations, whereas MemSkill remains strong in seen task distributions. On MemoryAgentBench, GeMS-SC achieves the best AR, LRU, SF, and Overall scores, while MemSkill performs best on TTL. These results suggest that graph-native retrieval, state maintenance, and conflict handling are well aligned with the requirements of dynamic memory-agent tasks, while explicit skill learning or experience evolution remains beneficial for test-time learning. Overall, the DeepSeek-V3.2 results are consistent with the main Qwen3-8B-Instruct results: GeMS-SC remains strong on cross-session memory maintenance and dynamic interaction, while StructMem and MemSkill retain advantages on temporal reasoning and test-time learning, respectively.

Method	ALFWorld			MemoryAgentBench				
	Seen	Unseen	Avg.	AR	TTL	LRU	SF	Overall
ReAct	52.4	46.8	49.6	48.5	40.2	38.6	20.8	37.0
MemoryBank	55.2	49.2	52.2	50.4	41.0	37.2	22.8	37.9
A-MEM	56.6	50.8	53.7	52.5	43.1	39.4	24.2	39.8
Mem0	58.4	52.1	55.3	50.9	39.5	36.8	22.5	37.4
Mem0-Graph	60.3	54.6	57.5	57.2	44.9	42.0	27.1	42.8
H-MEM	61.5	55.1	58.3	58.0	45.7	43.8	28.0	43.9
StructMem	63.2	56.4	59.8	60.6	46.1	45.9	29.4	45.5
MemSkill	**65.8**	58.7	62.3	62.4	**50.8**	45.0	31.2	47.4
GeMS-SC	65.1	**60.9**	**63.0**	**67.9**	49.7	**47.8**	**35.5**	**50.2**

Table 5: Complete dynamic interaction and memory maintenance results under DeepSeek-V3.2. Bold indicates the best result, and underline indicates the second-best result. Tied best results are all bolded.

A.2 Complete Ablation Results

This section provides detailed ablation results. We report category-level results on LoCoMo and complete dynamic-interaction results on ALFWorld and MemoryAgentBench. All variants use Qwen3-8B-Instruct as the base model and follow the same evaluation protocol as the full model.

Variant	Single	Multi	Temp.	Open	Overall	LongMem Eval
Full Model	**50.1**	**43.5**	**41.7**	**46.8**	**46.9**	**52.8**
Graph-CRUD Interface	39.4	30.8	27.2	39.1	35.3	39.6
w/o Skill Guidance	41.8	32.6	29.5	36.7	37.2	41.2
w/o CASE Selection	42.6	34.1	30.4	38.2	38.2	40.6
w/o CASE Learning	45.3	35.8	33.1	40.7	40.7	44.3
[l]w/o Skill-to-State Cross-Attention	43.7	37.2	31.6	42.1	39.9	43.2

Table 6: Complete ablation results on long-term language memory tasks.

Variant definitions. The Graph-CRUD Interface keeps the same graph memory state but replaces graph-native memory tools with generic operations that directly edit individual nodes and edges. The w/o Skill Guidance variant keeps the graph-native memory tools but removes the injected Graph-Native Memory Skill guidance. The w/o CASE Selection variant replaces CASE with static

Variant	ALFWorld SR			MemoryAgentBench				
	Seen	Unseen	Avg.	AR	TTL	LRU	SF	Overall
Full Model	**45.6**	**41.1**	**43.4**	**49.8**	**36.8**	**36.1**	**25.6**	**37.1**
Graph-CRUD Interface	34.0	29.6	31.8	36.2	27.4	24.9	12.5	25.2
w/o Skill Guidance	37.6	33.5	35.6	39.8	29.6	27.1	16.2	28.2
w/o CASE Selection	38.4	34.1	36.3	41.0	30.2	28.4	16.8	29.1
w/o CASE Learning	41.5	37.4	39.5	44.5	33.0	31.8	20.4	32.4
[l]w/o Skill-to-State Cross-Attention	40.2	36.8	38.5	43.1	32.5	30.8	19.8	31.6

Table 7: Complete ablation results on dynamic interaction and memory maintenance tasks.

skill guidance. The w/o CASE Learning variant keeps the CASE architecture but removes the two-stage learning procedure. The w/o Skill-to-State Cross-Attention variant removes the explicit cross-attention layer between candidate skills and execution-state fragments.

Observations. Tables 6 and 7 show consistent component-level trends across language memory and dynamic interaction tasks. The Graph-CRUD Interface causes the largest degradation, suggesting that direct edits to nodes and edges are less effective than graph-native tools for relation tracing, state maintenance, and conflict handling. Removing skill guidance or replacing CASE with static guidance also weakens performance, indicating that exposing memory tools alone is insufficient without step-level invocation guidance. The drops from removing CASE learning and Skill-to-State Cross-Attention further show that learned state-skill interaction contributes to selecting appropriate memory guidance under the current execution state.

A.3 Training Source Transfer Analysis

To examine whether the state-skill matching preferences learned by CASE can transfer across task types, we compare different CASE training sources on dynamic interaction tasks. Specifically, we directly transfer a LoCoMo-trained CASE to ALFWorld and MemoryAgentBench, and compare it with the main setting where CASE is trained on ALFWorld. This analysis is not used as the main experimental setting, but serves as a diagnostic study of how the training source affects dynamic interaction performance. Except for the CASE training source, the two GeMS-SC variants use the same base model, graph-native memory tools, maximum reasoning steps, decoding settings, and evaluation protocols.

Table 8 shows that the LoCoMo-trained CASE remains effective when transferred to dynamic interaction tasks. On MemoryAgentBench, it still outperforms MemSkill on Overall and remains stronger on AR, LRU, and SF. This suggests that part of the state-skill matching behavior learned by CASE is transferable across task types, especially for memory demands related to evidence recovery, state maintenance, and conflict handling.

Source	Seen	Unseen	Avg.	AR	TTL	LRU	SF	Overall
MemSkill	**46.5**	39.4	<u>43.0</u>	43.5	**37.4**	34.6	21.9	34.4
GeMS-SC w/ LoCoMo CASE	44.2	<u>39.9</u>	42.1	<u>48.6</u>	36.2	<u>35.4</u>	<u>24.1</u>	<u>36.1</u>
GeMS-SC w/ ALFWorld CASE	<u>45.6</u>	**41.1**	**43.4**	**49.8**	<u>36.8</u>	**36.1**	**25.6**	**37.1**

Table 8: Performance of different CASE training sources on dynamic interaction tasks with Qwen3-8B-Instruct as the base model. Bold indicates the best result, and underline indicates the second-best result. Tied best results are all bolded.

However, the ALFWorld-trained CASE further improves ALFWorld Unseen SR, Avg. SR, and MemoryAgentBench Overall. This indicates that dynamic interaction tasks introduce task-specific signals, such as action-observation histories, environment state changes, and new observation writing, which affect the state-skill matching preferences of CASE. In other words, CASE exhibits a degree of cross-task transferability, but its effectiveness still benefits from training sources that match the interaction form of the target task. This result supports the main experimental setting, where CASE is trained separately for the long-term language memory task group and the dynamic interaction task group.

A.4 Repeated Runs and Stability Analysis

To examine the stability of the main results, we conduct three independent runs for GeMS-SC and the closest strong baselines under the Qwen3-8B-Instruct setting. All runs use the same data splits, maximum reasoning steps, evaluation scripts, and L-J evaluation protocol. Table 9 reports the mean

Method	Single-hop	Multi-hop	Temporal	Open-domain	LoCoMo Overall	LongMemEval	ALFWorld Unseen
StructMem	50.6 ± 0.2	42.1 ± 0.7	43.0 ± 0.5	45.2 ± 0.4	47.1 ± 0.3	47.3 ± 1.0	37.5 ± 0.9
MemSkill	49.2 ± 0.5	40.1 ± 0.4	37.6 ± 1.1	43.8 ± 0.8	44.8 ± 0.6	52.1 ± 0.7	39.4 ± 1.3
GeMS-SC	50.1 ± 0.3	43.5 ± 0.9	41.7 ± 0.6	46.8 ± 0.5	46.9 ± 0.8	52.8 ± 1.2	41.1 ± 0.4

Table 9: Repeated-run results under Qwen3-8B-Instruct. We report mean and standard deviation over three independent runs.

and standard deviation.

The repeated runs show that GeMS-SC obtains high mean scores on Open-domain, LongMemEval, and ALFWorld Unseen, and remains competitive on Multi-hop questions. This suggests that GeMS-SC provides stable performance in open-ended evidence localization, cross-session memory maintenance, and state tracking in unseen environments.

At the same time, GeMS-SC and StructMem are very close on LoCoMo Overall, with 46.9 ± 0.8 and 47.1 ± 0.3, respectively. Their mean $\pm$ standard deviation ranges overlap, so we do not interpret the LoCoMo difference as a statistically significant gap. Instead, we view GeMS-SC as competitive with the strongest structured memory baseline on LoCoMo. StructMem remains stronger on Temporal questions, suggesting that explicit event structures are still effective for temporal reasoning.

On LongMemEval, GeMS-SC obtains 52.8 ± 1.2, while MemSkill obtains 52.1 ± 0.7. Since the margin is small, we interpret this result as a numerically higher but small-margin result of GeMS-SC on cross-session memory maintenance, without making an additional claim of statistical significance.

Overall, the repeated-run results support the stable competitiveness of GeMS-SC across different long-term memory settings. The main conclusion does not rely on a small margin on a single metric, but on its consistent performance across long-term language memory, cross-session memory, and dynamic interaction tasks.

A.5 State-Conditioned Skill Selection Analysis

To examine whether CASE learns state-conditioned memory guidance rather than relying on fixed prompt injection, we analyze the skill selection distribution across benchmarks. Figure 3 reports the selection frequency of non-null Graph-Native Memory Skills. Continue Regular Execution is omitted because it is the null skill that injects no additional memory guidance.

In LoCoMo and LongMemEval, Evidence-Grounded Retrieval is activated most frequently, indicating that long dialogue QA and cross-session memory tasks rely heavily on recovering relevant evidence from history. LongMemEval also shows a higher frequency of Conflict-Aware Maintenance than LoCoMo, which is consistent with its emphasis on information updates, temporal changes, and conflict handling.

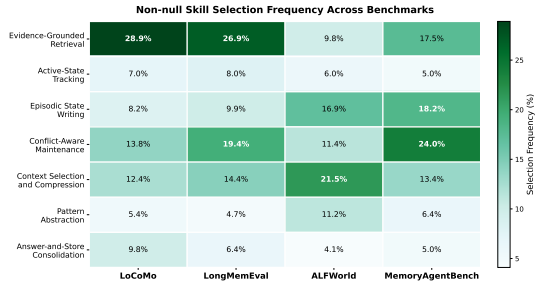

Figure 3: Selection frequency of non-null Graph-Native Memory Skills across benchmarks. Continue Regular Execution is omitted because it is the null skill that injects no additional memory guidance.

Active-State Tracking is activated less frequently across all benchmarks. This suggests that explicit state tracing is used as specialized intermediate guidance for verifying the currently valid state, rather than as a dominant standalone memory behavior.

In ALFWorld, Context Selection and Compression is activated most frequently, and Episodic State Writing is activated more often than in language memory tasks. This aligns with embodied multi-step interaction, where the agent needs to track action-observation histories, select local context, and record new environmental observations. In MemoryAgentBench, Conflict-Aware Maintenance is activated most frequently, while Episodic State Writing and Evidence-Grounded Retrieval also remain common, reflecting its focus on incremental information writing, evidence recovery, and conflict maintenance.

These results suggest that CASE does not apply the same memory prompt across all tasks. Instead, it adjusts guidance activation according to task type and execution state, complementing the ablation results in the main text on state-conditioned guidance modeling.

B CASE Construction Details

B.1 CASE Input Construction

Following the execution state and candidate skill set defined in the main text, CASE represents each decision step as a sequence of fragment-level units. Table 10 lists the fragments used for CASE encoding.

Frag.	N	Content	Type
Current subtask q_t	1	Current question, local goal, or action-decision requirement	query
Previous observation o_{t-1}	1	Observation returned by the previous task tool, memory tool, or environment	observation
Recent trajectory $h_{t-L:t-1}$	L	Recent execution history, including actions, observations, and intermediate state summaries	history
Candidate skills $S_0, \ldots, S_K$	$K+1$	Candidate Graph-Native Memory Skills, including the null skill	skill

Table 10: Input fragments used by CASE at each decision step.

For candidate skills, CASE only encodes the selection-side text:

$$\text{text}_{\text{select}}(S_k) = [\text{skill name}; \\ \text{state-matching description}].$$

The execution prompt of S_k is not used for CASE encoding or scoring. It is injected into the base model context only after the corresponding guidance is activated. This separation prevents CASE from encoding lengthy tool use instructions, while still allowing the activated guidance to provide concrete support for memory tool invocation to the base model.

B.2 Fragment Pooling and Embedding Construction

CASE operates on fragment-level representations rather than word-level tokens. Each input fragment can be the current subtask, the previous observation, a recent trajectory record, or the selection-side text of a candidate skill. This design reduces the sequence length processed by CASE and makes state-skill interaction modeling focus on execution-level semantic units.

Given a state fragment u_i^{state}, a frozen text encoder first produces its token-level hidden states:

$$H_i^{\text{state}} = E_{\text{text}}(u_i^{\text{state}}).$$

A pooling function then converts the token-level hidden states into a fragment-level vector:

$$e_i^{\text{state}} = \text{Pool}(H_i^{\text{state}}), \qquad e_i^{\text{state}} \in R^{d_0}.$$

For a candidate skill S_k, CASE encodes only its selection-side text:

$$H_k^{\text{skill}} = E_{\text{text}}(\text{text}_{\text{select}}(S_k)),$$

$$e_k^{\text{skill}} = \text{Pool}(H_k^{\text{skill}}), \qquad e_k^{\text{skill}} \in R^{d_0}.$$

The frozen encoder parameters are not updated during CASE training.

State fragments and skill fragments have different semantic roles. State fragments describe the current execution context, while skill fragments describe candidate memory invocation strategies. Therefore, CASE uses separate projection layers to map them into a shared hidden space:

$$x_i^{\text{state}} = W_{\text{state}} e_i^{\text{state}} + b_{\text{state}}, \qquad x_i^{\text{state}} \in R^d,$$

$$x_k^{\text{skill}} = W_{\text{skill}} e_k^{\text{skill}} + b_{\text{skill}}, \qquad x_k^{\text{skill}} \in R^d.$$

To distinguish the functional role and position of each fragment, CASE adds type embeddings and slot embeddings to the projected representations. For a state fragment, the final input representation is

$$\widetilde{x}_i^{\text{state}} = x_i^{\text{state}} + e_{\tau(i)}^{\text{type}} + e_{p(i)}^{\text{slot}},$$

where $\tau(i)$ denotes the fragment type and $p(i)$ denotes the slot index of the fragment. For a candidate skill, the final representation is

$$\widetilde{x}_k^{\text{skill}} = x_k^{\text{skill}} + e_{\text{skill}}^{\text{type}} + e_{p(k)}^{\text{slot}}.$$

The type embedding indicates the role of a fragment in the current decision step. We use four type categories: *query*, *observation*, *history*, and *skill*. They correspond to the current subtask, the previous observation, recent trajectory fragments, and candidate skills, respectively.

The slot embedding preserves positional and candidate-identity information. For recent trajectory fragments, slot embeddings encode their relative temporal positions within the window $h_{t-L:t-1}$, allowing CASE to distinguish earlier trajectory records from more recent ones. For candidate skills, slot embeddings encode the skill index in the candidate set, so that each skill keeps a stable candidate identity under the shared scoring architecture. The current subtask and the previous observation use fixed slots.

Frag.	Slot	Purpose
Current subtask q_t	Fixed query slot	Marks the current execution intention
Previous observation o_{t-1}	Fixed observation slot	Marks the latest external feedback
Recent trajectory $h_{t-L:t-1}$	Relative temporal history slots	Distinguishes earlier and more recent trajectory records
Candidate skills $S_0, \ldots, S_K$	Skill-index slots	Distinguishes candidate skill identities

Table 11: Slot embedding settings for CASE input fragments.

After adding type and slot embeddings, CASE forms the complete fragment-level input sequence:

$$\widetilde{X}_t = [\widetilde{x}_q, \widetilde{x}_o, \widetilde{x}_{h,1}, \ldots, \widetilde{x}_{h,L}, \widetilde{x}_0^{\text{skill}}, \widetilde{x}_1^{\text{skill}}, \ldots, \widetilde{x}_K^{\text{skill}}].$$

The resulting sequence is used as the input to the CASE interaction encoder.

B.3 CASE Construction Parameters

Table 12 summarizes the construction parameters used in all experiments.

Param.	Description	Value
L	Window size of recent execution trajectory	6
$K+1$	Number of candidate skills, including the null skill S_0	8
N_{state}	Number of state fragments	$2+L=8$
N_{skill}	Number of skill fragments	$K+1=8$
N_{total}	Total number of fragment-level input units	$N_{\text{state}} + N_{\text{skill}} = 16$
E_{text}	Frozen text encoder for state and skill fragments	`BGE-M3`
d	CASE hidden dimension after projection	128
Transformer layers	Number of layers in the state-skill interaction encoder	2
Attention heads	Number of attention heads in the interaction encoder and Skill-to-State Cross-Attention	4
Dropout	Dropout rate in the encoder and attention layers	0.1
Type embeddings	Fragment role categories	query, observation, history, skill
History slots	Slot embeddings for recent trajectory fragments	6
Skill slots	Slot embeddings for candidate skills	8

Table 12: CASE construction parameters. These parameters specify the model structure and input construction, rather than training optimization settings.

C Graph Memory Schema

This appendix specifies the graph memory schema used by graph-native memory tools. At decision step t, the external graph memory is represented as

$$M_t = (V_t, E_t, A_t),$$

where V_t denotes the set of memory nodes, E_t denotes the set of typed edges, and A_t denotes the attributes associated with nodes and edges. Graph-native memory tools operate on this structured memory state and return step-relevant structured results, such as evidence subgraphs, active states, support paths, conflict relations, or compact memory views, as memory observations for subsequent agent execution.

C.1 Node Types

Table 13 summarizes the node types used in the graph memory. These node types provide the basic memory objects over which evidence tracing, state tracking, conflict maintenance, and pattern abstraction are performed.

Anchor nodes normalize different surface forms of the same memory object and serve as entry points for graph traversal. Evidence nodes preserve traceable information sources. State nodes represent current or historical states under temporal or validity conditions. Insight nodes store reusable patterns supported by evidence nodes, state nodes, or local memory subgraphs. Outcome nodes record task results and their supporting evidence.

C.2 Edge Types

Table 14 lists the edge types used to connect memory objects. These edges make evidence sources, state changes, conflicts, and abstractions explicit in the memory graph.

Mention edges connect raw evidence to normalized memory anchors. Support edges keep the provenance of states, insights, and outcomes. Temporal / Update edges represent how events or states evolve over time and how outdated states are replaced by newer ones. Conflict edges mark inconsistent or invalidated memory structures. Abstraction edges connect evidence nodes, state nodes, or local memory subgraphs to reusable insights.

C.3 Attributes

Nodes and edges are associated with attributes that support localization, tracing, filtering, and maintenance. Table 15 gives the shared attribute fields.

Node type	Stored object	Main attributes
Anchor node	Normalized memory anchors, such as entities, events, user preferences, environment objects, or task objects.	canonical name, aliases, embedding
Evidence node	Raw or near-raw evidence units, such as dialogue turns, tool outputs, environment observations, user facts, or historical events.	content, timestamp, source, confidence
State node	Time- or validity-conditioned states of users, tasks, environments, or objects.	target, state type, value, validity, timestamp
Insight node	Reusable insights abstracted from repeated interactions, such as stable preferences, recurring patterns, failure modes, or reusable strategies.	pattern, condition, support count, source links
Outcome node	Task-level results, such as final answers, execution outcomes, completion states, or key evidence sets.	result, task id, supporting evidence, completion status

Table 13: Node types in the graph memory schema.

Edge type	Relation represented	Typical form
Mention edge	A reference relation between an evidence node and a normalized anchor.	evidence → anchor
Support edge	An evidential support relation from evidence to a state, insight, or outcome.	evidence → state / insight / outcome
Temporal / Update edge	A temporal, update, replacement, or supersession relation between events or states.	old state → new state
Conflict edge	A contradiction, invalidation, or resolution relation between evidence or state nodes.	state A ↔ state B
Abstraction edge	An abstraction relation from evidence nodes, state nodes, or local memory subgraphs to an insight node.	evidence / state → insight

Table 14: Edge types in the graph memory schema.

Attribute	Description
id	Unique identifier of a node or edge.
type	Node type or edge type.
content	Textual content, structured value, or relation description.
timestamp	Timestamp, session position, or execution position.
source	Information source, such as dialogue, tool output, environment observation, or task result.
confidence	Confidence score of the memory object or relation.
salience	Current importance, access weight, or retention priority.
validity	Validity status, such as active, outdated, conflicting, or archived.
embedding	Vector representation used for node localization and similarity matching.

Table 15: Shared attributes for nodes and edges in the graph memory.

Different node types additionally use type-specific attributes. Anchor nodes use *canonical name* and *aliases* to normalize different mentions of the same object. State nodes use *target*, *state type*, *value*, and *validity* to represent current or historical states. Insight nodes use *condition*, *support count*, and *source links* to record their applicability and supporting structures. Outcome nodes use *task id*, *result*, *supporting evidence*, and *completion status* to preserve task results and their evidence.

Under this schema, graph-native memory tools can return structured observations from M_t without exposing the full graph to the base model. The returned observation may contain a local evidence subgraph, an active-state path, a conflict relation, an insight node, or a compact memory view, depending on the current request and output budget.

D Graph-Native Memory Skill Specifications

D.1 Skill Field Format

The main text defines each Graph-Native Memory Skill as $S_k = (n_k, m_k, p_k)$, where n_k and m_k form the selection-side text encoded by CASE, and p_k is the execution-side prompt injected into the base model context after selection. This appendix provides the complete field specification omitted from the main text.

In implementation, the execution prompt p_k is decomposed as

$$p_k = [\underbrace{g_k}_{\text{goal}}, \underbrace{A_k}_{\text{tool APIs}}, \underbrace{\rho_k}_{\text{arguments}}, \underbrace{\eta_k}_{\text{handoff}}, \underbrace{\omega_k}_{\text{stop}}].$$

Here, g_k denotes the memory goal of the skill, A_k denotes the graph-native memory tool APIs that

should be considered, ρ_k describes how tool arguments should be constructed from the current subtask, previous observation, and recent trajectory, η_k specifies how memory tool outputs should be handed off to subsequent answer generation, memory operations, or task/environment actions, and ω_k specifies when to stop memory invocation and return to regular execution.

Table 16 specifies which fields are encoded by CASE and which fields are injected into the base model context after selection.

Field	Description	Sel.	Ctx.
`skill name`	The name of the skill, used to identify the type of candidate memory guidance.	Yes	Opt.
`state-matching description`	A short description of the execution states in which the skill is applicable.	Yes	No
`execution prompt`	The concise guidance prompt serialized into the context when the guidance is activated.	No	Yes
`execution prompt.goal`	The memory invocation goal of the activated guidance.	No	Yes
`execution prompt.tool APIs`	The graph-native memory tool APIs that should be preferentially considered under the activated guidance.	No	Yes
`execution prompt arguments`	Guidance for constructing tool arguments from the current subtask, previous observation, and recent trajectory.	No	Yes
`execution prompt handoff`	Guidance on how memory tool outputs should support subsequent answer generation, memory operations, or task/environment actions.	No	Yes
`execution prompt stop`	The stopping condition for memory invocation before returning to regular execution.	No	Yes

Table 16: Field format of Graph-Native Memory Skills.

The tool APIs listed in a skill define a preferred tool scope rather than a fixed tool chain. The activated guidance indicates which graph-native memory tools should be considered, but it does not determine the concrete tool names, arguments, or call order. These decisions are still made by the base model during subsequent action generation. For the null skill *Continue Regular Execution*, no execution prompt is injected.

D.2 Candidate Graph-Native Memory Skills

D.3 Representative Skill: Evidence-Grounded Retrieval

This subsection gives a representative Graph-Native Memory Skill, *Evidence-Grounded Retrieval*.

Skill name. *Evidence-Grounded Retrieval*.

State-matching description. This skill is applicable when the current execution step needs to recover historical evidence before answering, acting, or continuing with task/environment actions when available. It is used when the current question depends on dialogue history, cross-session facts, tool outputs, environment observations, multi-hop clues, or support relations. It is also applicable when the agent needs to verify the evidence source of an entity, event, preference, constraint, or task condition.

Execution prompt. The execution-side prompt of this skill consists of the following fields.

Goal. Recover structured evidence from the graph memory that is relevant to the current execution step. The returned result should include relevant nodes, edge types, support paths, temporal or update relations, and evidence sources, rather than isolated text snippets.

Tool APIs. This skill preferentially considers the following graph-native memory tools, but the listed tools do not form a fixed tool chain.

`RetrieveEvidenceSubgraph` recovers historical evidence, multi-hop clues, local evidence subgraphs, or support paths. Its arguments are constructed from the current query, entity or event anchors, temporal constraints, relation scope, execution constraints, and output budget. It returns evidence subgraphs, anchor nodes, edge types, support paths, and evidence sources.

`TraceActiveState` identifies the latest valid state of an entity, preference, task condition, or environment object. Its arguments are constructed from the target object, state type, temporal constraints, update-relation scope, and output budget. It returns the current valid state, state update path, invalidated states, and supporting evidence.

`RetrieveRelevantInsights` retrieves reusable experiences, stable preferences, failure patterns, or tool use strategies. Its arguments are constructed from the task goal, insight type, support-tracing

Skill name	Applicable state	Main goal
Continue Regular Execution	The current step mainly relies on the existing context, answer generation, or task tools/actions when available.	Inject no additional memory guidance and continue regular execution.
Evidence-Grounded Retrieval	The current step depends on historical evidence, support paths, multi-hop clues, or evidence sources.	Recover structured evidence subgraphs and hand off grounded evidence to later reasoning, answer generation, memory operations, or task/environment actions.
Active-State Tracking	The current step needs the latest valid state of an entity, event, preference, task condition, or environment object.	Trace temporal, update, or replacement relations to identify the current valid state.
Episodic State Writing	The current step produces new user facts, environment observations, task states, tool outputs, or intermediate conclusions.	Write new information into graph memory and connect it to relevant nodes, edges, and support relations.
Conflict-Aware Maintenance	The current observation conflicts with, supersedes, or invalidates existing memory.	Correct conflicting memory structures while preserving provenance and updating the current valid state.
Context Selection and Compression	Retrieved evidence, recent trajectory, or local memory view is too large for direct use.	Compress redundant local subgraphs while preserving key evidence chains, state nodes, and task constraints.
Pattern Abstraction	Repeated preferences, failure modes, strategies, or tool use experiences appear across tasks or trajectories.	Promote supported recurring structures into reusable insight nodes.
Answer-and-Store Consolidation	The task is finished or close to completion.	Store the final answer, execution result, key evidence, and task state for future use.

Table 17: Candidate Graph-Native Memory Skills used by CASE.

requirement, execution constraints, and output budget. It returns relevant insight nodes, applicability conditions, support chains, and source evidence.

Arguments. The agent should extract the query target, entity or event anchors, temporal constraints, state types, and evidence requirements from the current subtask. It should also use the previous observation and recent trajectory to identify retrieval failures, unresolved constraints, states that need verification, and context-budget pressure. If the current step mainly depends on historical evidence, the agent should prioritize constructing arguments for `RetrieveEvidenceSubgraph`. If the step concerns the latest valid state, it should prioritize `TraceActiveState`. If the step depends on high-level experience or reusable patterns, it should prioritize `RetrieveRelevantInsights`.

Handoff. The returned evidence subgraph, support path, active state, or relevant insight should be used as grounded context for subsequent reasoning. If the agent continues with task/environment actions or external task tools, it should use entities, constraints, current states, or historical tool outputs from the returned memory results to construct action or tool arguments. If the agent generates an answer, the answer should be consistent with the recovered evidence sources, state paths, and support relations.

Stop. The agent should stop memory invocation and return to regular execution when the evidence subgraph covers the key entities, temporal constraints, and support paths required by the current question, or when `TraceActiveState` has identified the latest valid state. If the same anchor, constraint, or state object has already been retrieved repeatedly, the agent should avoid further calls to the same type of memory tool.

E Graph-Native Memory Tools

E.1 Tool Invocation Interface

This section specifies the request and observation format of graph-native memory tools.

Let $\mathcal{A}_m$ denote the graph-native memory tool set. For a graph-native memory tool $a_m \in \mathcal{A}_m$, its invocation is defined as

$$a_m : (\mathcal{M}_t, r_t) \rightarrow (\mathcal{M}_{t+1}, o_t),$$

where $\mathcal{M}_t$ denotes the external graph memory state before the current decision step, r_t denotes the tool request generated at step t, and o_t denotes the observation returned to the base model. The updated memory state $\mathcal{M}_{t+1}$ may be identical to $\mathcal{M}_t$ for read-only tools, or may contain graph-state changes produced by writing, abstraction, or maintenance tools.

Each tool request contains the fields listed in Table 18. The returned observation o_t is serialized

into the subsequent execution trajectory and can be used by the base model for further reasoning, task-side decisions, memory tool calls, or final answer generation. Its fields are summarized in Table 19.

If a tool request violates its preconditions, the tool returns a failure observation and keeps the external graph memory state unchanged. The failure observation is still added to the execution history, so that the base model and CASE can condition subsequent decisions on the failed memory invocation. This interface design keeps memory operations explicit and traceable while preserving the base model's responsibility for action generation.

E.2 Graph-Native Memory Tool Set

We organize graph-native memory tools into four categories: writing, retrieval, abstraction, and maintenance. Each tool corresponds to a complete memory operation over the graph memory. Its internal execution may involve node localization, edge-type filtering, support-chain tracing, local subgraph selection, or graph-state updating, but these fine-grained procedures are not exposed as separate tools to the base model.

Tables 20–23 summarize the graph-native memory tool set by category. Writing tools connect new information to the graph memory and establish necessary typed edges and support relations. Retrieval tools return structured memory views required by the current decision step, such as evidence subgraphs, active states, or insight nodes. Abstraction tools reduce redundant structures and promote reusable patterns. Maintenance tools handle conflicts, salience changes, and inactive memory structures during long-horizon execution.

Together, these tools expose graph-object-level operations to the base model through a unified invocation interface.

E.3 Representative Tool Algorithms

This subsection gives representative algorithms for three graph-native memory tools. These algorithms describe the internal behavior of the tool executor over the graph memory. They are not intermediate reasoning steps generated by the base model. During agent execution, the base model only generates the tool name and request; graph traversal, node scoring, support tracing, state-path tracking, and conflict resolution are handled by the tool executor.

[!htbp] [1] Memory graph $\mathcal{M}_t$, query q_t, relation scope $\mathcal{R}$, maximum depth h, budget B Evidence subgraph $\widehat{\mathcal{G}}_t$ and observation o_t Select candidate nodes $\mathcal{V}_c$ satisfying the operation scope and execution constraints. Compute anchor score

$$s_{\text{anchor}}(v \mid q_t) = \frac{\text{sim}(q_t, v) + \text{supp}(v) + \text{time}(q_t, v)}{3} - \text{conflict}(v).$$

Select anchors $\mathcal{A}_t = \text{TopK}_{v \in \mathcal{V}_c} s_{\text{anchor}}(v \mid q_t)$. $\mathcal{A}_t = \emptyset$ $\mathcal{M}_t$, failure observation. Expand $\mathcal{G}_c = \text{Expand}(\mathcal{M}_t, \mathcal{A}_t, \mathcal{R}, h)$. Remove outdated, invalid, and unresolved conflict nodes from $\mathcal{G}_c$. Trace support chains for query-level or insight-level nodes. Select $\widehat{\mathcal{G}}_t$ under budget B by maximizing relevance and support while penalizing redundancy. $\widehat{\mathcal{G}}_t = \emptyset$ $\mathcal{M}_t$, failure observation. Construct o_t with status, evidence subgraph, anchors, and provenance. $\mathcal{M}_t$, o_t.

[!htbp] [1] Memory graph $\mathcal{M}_t$, target object x, state type c, temporal constraint $\mathcal{T}$ Active state v^*, update path $\mathcal{P}^*$, and observation o_t Locate candidate states $\mathcal{V}_{x,c} = \{v \in \mathcal{M}_t \mid \text{obj}(v) = x, \text{state}(v) = c\}$. $\mathcal{V}_{x,c} = \emptyset$ $\mathcal{M}_t$, failure observation. Define update relations $\mathcal{R}_u = \{\texttt{temporal}, \texttt{update}, \texttt{replace}\}$. Construct a state-transition subgraph from $\mathcal{V}_{x,c}$ using relations in $\mathcal{R}_u$. Remove states that do not satisfy the temporal constraint $\mathcal{T}$. Mark v as superseded if there exists a valid later state u connected by a relation in $\mathcal{R}_u$. Obtain $\mathcal{V}_{\text{active}}$, the states not superseded by valid later states. $\mathcal{V}_{\text{active}} = \emptyset$ $\mathcal{M}_t$, failure observation. Score each active-state candidate:

$$s_{\text{state}}(v) = \frac{\text{recency}(v) + \text{supp}(v) + \text{source_quality}(v)}{3}.$$

Select $v^* = \arg\max_{v \in \mathcal{V}_{\text{active}}} s_{\text{state}}(v)$. Trace update path $\mathcal{P}^* = \text{Path}_{\mathcal{R}_u}(v_{\text{init}} \to \cdots \to v^*)$, where v_{init} is the earliest reachable state node satisfying the target object and state type. Construct o_t with status, active state, update path, superseded states, and support evidence. $\mathcal{M}_t$, o_t.

[!htbp] [1] Memory graph $\mathcal{M}_t$, candidate node set $\mathcal{C}$, maintenance constraints Ω, salience decay λ Updated memory graph $\mathcal{M}_{t+1}$ and observation o_t Initialize conflict pairs $\mathcal{P}_{\text{conflict}} = \emptyset$. Add (u, v) to $\mathcal{P}_{\text{conflict}}$ when the two nodes have the same target and state type but contradict each other. $\mathcal{P}_{\text{conflict}} = \emptyset$ $\mathcal{M}_t$, success observation with no graph-state change. $(u, v) \in \mathcal{P}_{\text{conflict}}$ Compute keep score

$$s_{\text{keep}}(z) = \frac{\text{recency}(z) + \text{supp}(z) + \text{source_quality}(z)}{3}, \quad z \in \{u, v\}.$$

Select $v_{\text{keep}} = \arg\max_{z \in \{u,v\}} s_{\text{keep}}(z)$ and denote the other node as v_{old}. Add replacement edge $(v_{\text{old}}, \texttt{replace}, v_{\text{keep}})$. Preserve provenance links and support chains of v_{old}. Decay salience $\text{salience}_{t+1}(v_{\text{old}}) = \lambda \cdot \text{salience}_t(v_{\text{old}})$. Update affected support edges. Construct o_t with status, kept nodes, superseded nodes, added edges, and provenance. $\mathcal{M}_{t+1}$, o_t.

Other graph-native memory tools follow the same invocation interface, but differ in their target graph objects and memory state transitions. $\text{source_quality}(\cdot)$ denotes a scalar reliability score derived from the information source.

@p0.29X@

Field	Description
tool_name	The graph-native memory tool API to be invoked at the current decision step.
operation target	The structured graph object directly operated on by the tool, such as a node, edge, local subgraph, support chain, update path, or conflict subgraph.
operation scope	The node types, edge types, relation scopes, or local neighborhoods that the tool is allowed to access or modify.
execution constraints	Conditions that the tool execution should satisfy, such as temporal constraints, node types, support strength, state validity, or conflict markers.
output budget	The output-size constraint, such as the maximum number of nodes, edges, evidence chains, or context tokens returned by the tool.

Table 18: Fields of a graph-native memory tool request.

@p0.29X@

Field	Description
status	The execution status of the tool call, indicating whether the invocation succeeds or fails.
structured result	The structured graph result returned by the tool, such as an evidence subgraph, active state, compressed memory view, conflict-resolution result, or insight node.
graph_state change	The graph-state change produced by the tool, including node insertion, state update, subgraph merging, archival, or relation reweighting. This field is empty for read-only tools.
provenance	The source nodes or support paths that justify the returned result.
failure message	The failure reason returned when the request does not satisfy the tool preconditions or no valid graph object can be found.

Table 19: Fields of a graph-native memory tool observation.

Tool API	Main graph objects	Function
WriteMemory Subgraph	Typed nodes, typed edges, support relations	Writes new facts, environment observations, task-tool outputs when available, or task states into the graph memory, and establishes necessary typed edges and support relations.
UpdateMemory State	State nodes, update edges, support relations	Updates existing task states, user states, or environment states according to new observations or new evidence, so that the memory graph reflects currently valid information.
StoreTask Outcome	Task outcome nodes, evidence links, result edges	Stores the final answer, execution result, key evidence sources, and task completion state after a task is finished, providing traceable experience for future tasks.

Table 20: Graph-native memory tool set: writing tools.

Tool API	Main graph objects	Function
RetrieveEvidence Subgraph	Local evidence subgraphs	Recovers the structured evidence subgraph required by the current question or action decision, returning relevant nodes, edge types, support paths, and evidence sources.
TraceActive State	Temporal paths, update paths, state nodes	Traces the currently valid state of an entity, event, preference, or task condition, and filters out outdated or superseded historical states.
RetrieveRelevant Insights	Insight nodes, support chains	Retrieves reusable insights, stable preferences, failure patterns, or reusable strategies related to the current task, while preserving their underlying support sources.

Table 21: Graph-native memory tool set: retrieval tools.

Tool API	Main graph objects	Function
CompressMemory Subgraph	Dense local subgraphs	Compresses an oversized local subgraph into a compact state or structured memory view that can be injected into context, while preserving key edges and evidence sources.
MergeEquivalent Subgraphs	Equivalent nodes, redundant subgraphs	Merges duplicate, equivalent, or highly redundant memory nodes and local subgraphs, reducing repeated structures in long-term memory.
PromoteSupported Pattern	Stable patterns, support subgraphs	Promotes recurring and sufficiently supported structural patterns into insight nodes for reuse in subsequent tasks.

Table 22: Graph-native memory tool set: abstraction tools.

Tool API	Main graph objects	Function
ResolveMemory Conflict	Conflict subgraphs, replacement edges, conflict edges	Resolves conflicting, outdated, mutually exclusive, or superseded memory structures, while preserving the provenance of old information.
RefreshMemory Salience	Typed edges, support relations, node weights	Refreshes the salience of nodes and edges according to new evidence, temporal decay, usage frequency, repeated support, or conflict-resolution results.
ArchiveInactive Subgraph	Low-value nodes, inactive subgraphs	Moves low-support, rarely used, or long-inactive memory structures out of the active graph while preserving traceable provenance.

Table 23: Graph-native memory tool set: maintenance tools.

F CASE Learning Details

F.1 Training Data and Hyperparameters

Table 24 gives the training data scale used for the two-stage learning procedure defined in the main text. Table 25 reports the optimization hyperparameters.

All training data are drawn from the corresponding training splits. Validation and test examples are not used for parameter updates.

The same optimizer, learning rate, batch size, regularization settings, and decoding temperature are used across task groups. The decoding temperature is set to zero for trajectory generation and evaluation to reduce randomness in trajectory collection and downstream assessment.

F.2 Weak Skill Label Construction

The state-skill alignment warm-up stage uses automatically constructed step-level weak skill labels. For each decision step t in a training trajectory, we collect the current CASE input $X_{i,t}$ and a short future window of execution behavior, including subsequent tool calls, environment feedback, and returned observations:

$$\mathcal{W}_{i,t}^{\Delta} = \{(a_{i,j}, o_{i,j})\}_{j=t}^{\min(t+\Delta, T_i)},$$

where $a_{i,j}$ denotes the action at step j, $o_{i,j}$ denotes the corresponding tool or environment observation, and Δ is the future-window size. The weak label is generated by a fixed weak-labeling procedure:

$$\widehat{S}_{i,t} = g_{\text{weak}}\left(X_{i,t}, \mathcal{W}_{i,t}^{\Delta}, \mathcal{S}\right),$$

where $\mathcal{S}$ is the candidate Graph-Native Memory Skill set.

In implementation, g_{weak} is instantiated with a fixed LLM-assisted labeling prompt. The prompt takes the current subtask, the previous observation, the recent trajectory, the future-window behavior, and the candidate skill descriptions as input, and outputs a weak skill label, a confidence level, and a short rationale. Figure 4 shows the prompt template used for weak label construction. All warm-up samples use the same prompt, candidate skill set, and output schema.

The weak label is not determined by the mere occurrence of a specific memory tool. Instead, the labeling procedure considers the tool types in the future window, the returned observations, whether the returned memory results are used by subsequent reasoning, answer generation, or task/environment actions when available, and the main memory demand reflected by the window. We keep only high- and medium-confidence samples for the warm-up stage. Low-confidence samples and samples whose rationales are inconsistent with the observed execution behavior are discarded. These weak labels are used only as coarse initialization signals for state-skill matching, rather than gold annotations or inference-time decision rules.

Figure 5 provides representative weak-label construction examples for retrieval-oriented and conflict-maintenance states.

Table 26 shows that more than 90% of decision steps in both task groups receive retained weak labels, providing a stable initialization signal for CASE warm-up. The discarded samples mainly correspond to windows without a clear memory demand, windows with mixed behaviors but no dominant skill, or rationales that are not supported by the observed execution behavior. ALFWorld has a slightly higher discarded rate, because embodied interaction trajectories contain denser tool calls and environment feedback, making short future windows more likely to mix multiple behaviors.

The weak labels are not used during inference. At test time, CASE only observes the current subtask, the previous observation, the recent trajectory, and the candidate skill set, and derives current-step guidance according to the learned state-skill matching distribution. The second-stage trajectory-feedback optimization further adjusts the state-skill matching preference according to task completion, consistency between activated guidance and memory tool calls, memory state quality, and execution overhead.

F.3 Trajectory-Level Feedback Components

This subsection details the trajectory-level feedback used in the second-stage training of CASE. The composite feedback follows the form defined in the main text. All feedback components are normalized to $[0, 1]$ before linear combination. By default, we use

$$(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = (0.40, 0.25, 0.25, 0.10).$$

Task completion quality. R_i^{task} measures whether the trajectory completes the target task. For long-term language memory tasks, this component is computed from the reference answer or semantic correctness judgment of the training query. For dynamic interaction tasks, it

Task group	Learning stage	Data source	Queries / tasks	Trajectories	Decision steps	Weak labels
LoCoMo	State-skill alignment warm-up	1 training conversation	128	128	584	552
LoCoMo	Trajectory-feedback skill optimization	2 training conversations	247	247	1268	—
ALFWorld	State-skill alignment warm-up	Expert trajectories from the training split	320	320	2864	2617
ALFWorld	Trajectory-feedback skill optimization	Training-task rollouts	320	320	3096	—

Table 24: Training data scale for the two-stage learning of CASE. Weak labels are used only in the state-skill alignment warm-up stage.

Prompt for Weak Skill Label Construction

```
[SYSTEM]
You are a weak-labeling assistant for CASE warm-up training.
Given the current execution state and a short future window of behavior, assign
 the most appropriate weak skill label for the current decision step.
[INPUT]
Current Subtask: The local goal or question at the current step.
Previous Observation: The latest task-tool, memory-tool, or environment feedback.
Recent Trajectory: The recent execution history before the current step.
Future-Window Behavior: A short window of subsequent actions and observations.
Candidate Skills:
Evidence-Grounded Retrieval; Active-State Tracking; Episodic State Writing;
Conflict-Aware Maintenance; Context Selection and Compression; Pattern Abstraction;
Answer-and-Store Consolidation; Continue Regular Execution.
[RULES]
Evidence-Grounded Retrieval: recovering historical evidence or support paths
Active-State Tracking: tracing the latest valid state of an entity, event,
preference, task condition, or environment object
Episodic State Writing: recording new observations or task states
Conflict-Aware Maintenance: resolving outdated or conflicting memory states
Context Selection and Compression: selecting or compressing retrieved memory context
Pattern Abstraction: abstracting repeated evidence or behavior into reusable insight
nodes
Answer-and-Store Consolidation: storing final answers, task outcomes, key evidence,
and resulting task states
Continue Regular Execution: no clear memory demand
[OUTPUT]
Return only a valid JSON object:
{
  "weak_label": "one of the candidate skill names"
  "confidence": "high | medium | low",
  "rationale": "brief explanation"
}
```

Figure 4: Prompt template used for weak skill label construction. The prompt takes the current execution state, a short future behavior window, and candidate skill descriptions as input, and returns a weak skill label, confidence level, and brief rationale.

Weak-Label Construction Examples

Sample 1

Current Subtask:
Answer a long-dialogue question that requires historical evidence from previous turns.

Previous Observation:
The current context is insufficient to support a grounded answer.

Recent History:

Step -3: The agent identifies the target person and event from the question.

Step -2: The agent checks the local context but does not find enough evidence.

Step -1: The agent decides that memory retrieval is needed before answering.

CASE Input:
Current subtask || Previous observation || Recent trajectory || Candidate skill descriptions

Future-window Behavior:
RetrieveEvidenceSubgraph -> TraceActiveState

Weak Label: Evidence-Grounded Retrieval

Confidence: High

Rationale:
The following steps mainly recover historical evidence and verify

the currently valid state before answer generation.

Sample 2

Current Subtask:

Answer a question involving an updated or temporally changed memory state.

Previous Observation:
The retrieved memory may be outdated or superseded by newer evidence.

Recent History:

Step -3: The agent retrieves an earlier state from memory.

Step -2: The agent observes that a later update may override the earlier state.

Step -1: The agent needs to resolve the conflict before continuing.

CASE Input:
Current subtask || Previous observation || Recent trajectory || Candidate skill descriptions

Future-window Behavior:
TraceActiveState -> ResolveMemoryConflict -> UpdateMemoryState

Weak Label: Conflict-Aware Maintenance

Confidence: High

Rationale:
The following steps focus on checking update relations and

correcting an outdated or conflicting memory state.

Figure 5: Representative examples of weak skill label construction. The examples illustrate how future-window memory behavior is mapped to coarse initialization labels for CASE warm-up.

Parameter	Description	Value
Optimizer	Optimizer for CASE parameters	AdamW
Batch size	Mini-batch size for CASE training	16
Warm-up epochs	Number of epochs for state-skill alignment warm-up	3
Feedback optimization epochs	Number of epochs for trajectory-feedback skill optimization	5
Learning rate	AdamW learning rate	1×10^{-4}
λ_{ent}	Entropy regularization weight	0.01
λ_{reg}	Parameter regularization weight	1×10^{-4}
Decoding temperature	Decoding temperature for trajectory generation and evaluation	0

Table 25: Training hyperparameters for CASE.

Task group	Steps	Retained	Discarded	Ret.	Disc.
LoCoMo	584	552	32	94.5%	5.5%
ALFWorld	2864	2617	247	91.4%	8.6%

Table 26: Weak-label retention statistics for the state-skill alignment warm-up stage. Retained labels include high- and medium-confidence samples. Discarded samples include low-confidence cases and samples whose rationales are inconsistent with the observed execution behavior.

is computed from the environment feedback or task success signal. This component evaluates the final task outcome of the whole trajectory, rather than the correctness of each intermediate guidance activation.

Skill-behavior alignment. R_i^{align} measures whether the activated guidance is consistent with the subsequent memory tool behavior. For decision steps where such consistency can be evaluated, we check whether the tool scope associated with the activated guidance covers the dominant memory tool behavior in a short future window:

$$R_i^{\text{align}} = \frac{1}{|\mathcal{T}_i^{\text{align}}|} \sum_{t \in \mathcal{T}_i^{\text{align}}} I\left[\text{match}\left(S_{i,t}, a_{i,t:t+\Delta}\right)\right],$$

where $\mathcal{T}_i^{\text{align}}$ denotes the set of evaluable decision steps, $S_{i,t}$ is the skill whose guidance is activated at step t, and $a_{i,t:t+\Delta}$ denotes the actions in the short future window. If $\mathcal{T}_i^{\text{align}} = \emptyset$, we set $R_i^{\text{align}} = 0.5$ as a neutral value. For *Continue Regular Execution*, the activation is treated as aligned when the future window contains no additional memory tool calls and the trajectory continues to make task progress.

Memory-operation quality. R_i^{memory} measures whether memory tool calls improve the graph memory state through effective memory operations. Effective writing, evidence retrieval, conflict resolution, support-relation construction, and archival of inactive structures are counted as positive memory operations.

For retrieval tools, we use a deterministic trace-linked usage proxy rather than an additional semantic judge. For a retrieval call at step t, let

$$\mathcal{K}_{i,t} = \text{Key}(o_{i,t}^{\text{mem}})$$

denote the set of structured keys returned in the memory observation. These keys include graph node ids, edge ids, evidence ids, support-path ids, canonical anchors, and state tuples such as $(\text{target}, \text{state type}, \text{value})$. We then extract the set of keys explicitly referenced after step t:

$$\mathcal{D}_{i,t}^{+} = \left(\bigcup_{j=t+1}^{T_i} \text{Key}(a_{i,j}, \Delta M_{i,j}) \right) \cup \text{Key}(\hat{y}_i),$$

where $\Delta M_{i,j}$ denotes the graph-state change produced by a later memory tool call, and $\hat{y}_i$ denotes the final answer when available. The function $\text{Key}(\cdot)$ extracts graph-object ids, canonical anchors, state tuples, or normalized argument values from later model-generated actions, memory-tool requests, graph-state changes, and final answers. Raw environment observations are not counted by themselves unless their content is written into a later memory update or used in a model-generated action or request.

A retrieval call is counted as effective when it succeeds and at least one returned key is explicitly reused later:

$$u_{i,t}^{\text{ret}} = I\big[\text{status}(o_{i,t}^{\text{mem}}) = \text{success} \wedge \mathcal{K}_{i,t} \cap \mathcal{D}_{i,t}^{+} \neq \emptyset\big].$$

Accordingly,

$$N_i^{\text{retrieve}} = \sum_{t: a_{i,t} \in \mathcal{A}_{\text{ret}}} u_{i,t}^{\text{ret}},$$

where $\mathcal{A}_{\text{ret}}$ denotes the retrieval-type memory tools.

For writing and maintenance tools, a call is counted as effective only when it satisfies the tool preconditions, returns a successful status, and produces a valid graph-state change without breaking existing provenance or support relations. We then compute

$$R_i^{\text{memory}} = \text{clip}\left(\frac{N_i^{\text{write}} + N_i^{\text{retrieve}} + N_i^{\text{resolve}} + N_i^{\text{support}} + N_i^{\text{archive}}}{N_i^{\text{mem}} + \epsilon}, 0, 1 \right),$$

where N_i^{mem} is the total number of memory tool calls in trajectory i. This proxy may under-count implicit use of retrieved memory, but it provides a deterministic and reproducible feedback signal from execution logs without introducing another LLM judge into CASE training.

Execution efficiency. R_i^{eff} penalizes excessive tool calls, long injected contexts, and repeated loops:

$$R_i^{\text{eff}} = 1 - \min(1, C_i).$$

The normalized cost C_i is computed as

$$C_i = \alpha_{\text{call}} \frac{N_i^{\text{call}}}{N_{\max}^{\text{call}}} + \alpha_{\text{tok}} \frac{T_i^{\text{tok}}}{T_{\max}^{\text{tok}}} + \alpha_{\text{loop}} \frac{N_i^{\text{loop}}}{N_{\max}^{\text{loop}}},$$

where N_i^{call} is the number of tool calls, T_i^{tok} is the context-token overhead, and N_i^{loop} is the number of repeated-loop patterns in trajectory i. We use

$$(\alpha_{\text{call}}, \alpha_{\text{tok}}, \alpha_{\text{loop}}) = (0.4, 0.4, 0.2).$$

Table 27 summarizes the default component weights.

Component	Description	Weight
R_i^{task}	Final task completion quality, measured by semantic correctness for language memory tasks or task success signals for dynamic interaction tasks.	0.40
R_i^{align}	Consistency between activated guidance and subsequent memory tool behavior.	0.25
R_i^{memory}	Quality of memory operations, including effective writing, retrieval, conflict resolution, support construction, and archival.	0.25
R_i^{eff}	Execution efficiency, penalizing excessive tool calls, context overhead, and repeated loops.	0.10

Table 27: Trajectory-level feedback components used for CASE training.

F.4 Feedback Weight Sensitivity

To examine the effect of the trajectory-level feedback weights in the composite feedback defined in the main text, we compare four representative weight settings. The compared settings are summarized in Table 28. The default setting is used in the main experiments.

Setting	λ_1 Task	λ_2 Alignment	λ_3 Memory	λ_4 Efficiency
Default	0.40	0.25	0.25	0.10
Equal	0.25	0.25	0.25	0.25
Task-heavy	0.55	0.15	0.20	0.10
Memory-heavy	0.30	0.25	0.35	0.10

Table 28: Feedback weight settings for the sensitivity analysis.

Figure 6 shows the moving-average trajectory feedback under different weight settings. The

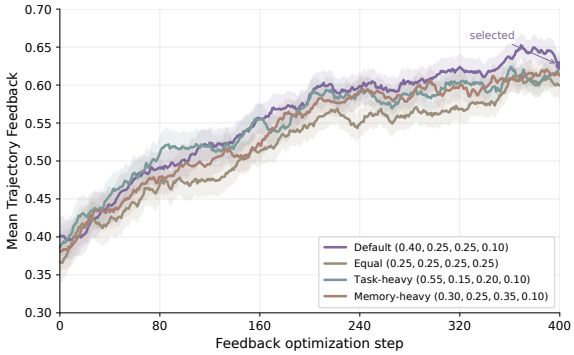

Figure 6: Feedback weight sensitivity. Curves show moving-average trajectory feedback under different weight settings; shaded regions indicate mini-batch-level variation.

Equal setting assigns the same weight to all components, but it gives relatively high importance to the efficiency term, which may discourage useful memory tool calls during early exploration. The Task-heavy setting emphasizes final task completion quality, but reduces the influence of skill-behavior alignment and memory-operation quality. As a result, it provides weaker direct supervision for whether the activated guidance matches the realized memory tool behavior. The Memory-heavy setting increases the weight of memory-operation quality, which encourages effective memory operations, but may overemphasize memory state changes when the current step mainly requires regular task progress.

The Default setting balances task completion, skill-behavior alignment, memory-operation quality, and execution efficiency. It assigns the largest weight to task completion quality, while still keeping substantial weights for alignment and memory-operation quality. This design encourages CASE to select skills that improve the final task outcome, remain consistent with subsequent memory tool behavior, and produce useful graph-memory states. The efficiency component is kept smaller, serving mainly as a regularizing penalty against excessive tool calls and context expansion.

Overall, the sensitivity analysis shows that the default setting yields a more stable feedback trend in the later stage of training. We therefore use $(0.40, 0.25, 0.25, 0.10)$ as the feedback weight setting in the main experiments.

G LLM-as-a-Judge Evaluation Protocol

We use an LLM-as-a-Judge protocol to evaluate semantic correctness on long-term language mem-

ory tasks. For each instance, the judge model receives the question, the reference answer, and the model prediction, and returns a binary semantic-correctness label. A score of 1 indicates a semantically correct answer, while a score of 0 indicates an incorrect answer. We use GPT-4o as the judge model with temperature set to 0, and keep the judge prompt and decoding settings fixed across all evaluated methods.

Evaluation setting. Each judging instance contains only the task question, the reference answer, and the model answer. Method names, model identities, and other metadata are hidden from the judge to reduce potential bias. The same judging prompt is used for all methods.

Human verification. To assess the reliability of the binary L-J decisions, we manually verify a stratified subset of judged examples from LoCoMo and LongMemEval. The subset covers different question types and predictions from multiple methods. Table 29 reports the agreement between L-J decisions and human verification.

Dataset	Samples	Agreement	κ	Disagr.
LoCoMo	100	96.0%	0.91	4
LongMemEval	100	97.0%	0.93	3
Total	200	96.5%	0.92	7

Table 29: Agreement between L-J decisions and human verification. κ denotes Cohen's kappa.

The verification results show that the binary L-J decisions are highly consistent with human judgment on the checked samples. Disagreements mainly occur in borderline cases, such as partially correct answers, complex temporal updates, or semantically close but incomplete paraphrases. We do not observe a systematic disagreement pattern favoring a specific method. Therefore, we use binary L-J as the primary automatic metric, while avoiding strong significance claims when the margin between methods is small.

Judge prompt. The exact prompt used for L-J evaluation is shown in Figure 7.

H Baseline Adaptation and Reproducibility Details

This section describes how each baseline is adapted to the unified evaluation protocol. Our goal is to keep the evaluation environment comparable while preserving the original memory construction, retrieval, controller learning, policy optimization, or skill evolution mechanisms of each baseline. All methods are evaluated on the same test splits, task inputs, maximum execution steps, decoding settings, and evaluation scripts. For baselines that require training or configuration selection, only training or validation data are used. During evaluation, no baseline component is updated with feedback from the target test set.

We do not claim that all baselines use identical training resources. Different baselines make different assumptions about memory construction, trainable modules, retrieval policies, and controller optimization. Therefore, we describe the reproduction settings of each baseline separately, including its memory representation, required tools or modules, trainable components, training data source, and fixed test-time configuration when applicable.

ReAct. ReAct is used as the base agent without adding an explicit long-term memory management module, structured memory state, trainable memory controller, or memory skill module. It follows the same Think–Act–Observe execution format as the other methods and uses the unified base model, task input, maximum execution steps, decoding setting, and evaluation script. ReAct can access the original task history, environment observations, standard database, or benchmark-provided retrieval interface when such information is part of the task input. It does not perform explicit memory writing, memory updating, memory compression, memory deletion, or memory guidance activation. No task-specific training is performed.

MemoryBank. MemoryBank is reproduced as a user-level long-term memory baseline. It maintains user facts, historical events, and user-profile information, and supports memory writing, retrieval, and updating across interactions. Following the original design, we preserve its dynamic memory strength mechanism, where memories can be reinforced or weakened according to time interval and importance. For each evaluation instance, we first construct the user-level long-term memory from the available historical interactions and then retrieve relevant memories for the current query or decision step. MemoryBank does not train a task-specific neural controller in our evaluation. Its retrieval and update settings are selected on training or validation data and fixed during testing.

L-J Evaluation Prompt

System:

```
You are an impartial evaluator for long-term memory question answering.
Your task is to judge whether the model answer is semantically correct
with respect to the reference answer for the given question.

You will be given three inputs:

Question:
{question}

Reference Answer:
{reference_answer}

Model Answer:
{model_answer}

Please follow these rules:

Score 1 if the model answer correctly answers the question and is
semantically consistent with the reference answer.

Score 1 if the model answer uses different wording or a shorter
expression, as long as the meaning is correct.

Score 1 if the model answer includes extra information that does not
contradict the reference answer.

Score 0 if the model answer is incorrect, contradictory, unsupported,
incomplete, or only partially answers the question.

Score 0 if the model answer gives the wrong person, object, event, time,
location, preference, or state.

Score 0 if the model answer relies on an outdated memory state when the
reference answer reflects a newer or corrected state.

For temporal or updated facts, judge according to the latest valid state
implied by the reference answer.

Do not use external knowledge. Judge only based on the question and the
reference answer.

Do not reward fluency, length, style, or reasoning process. Judge only
semantic correctness.

Return only a valid JSON object:


{
  "score": 0 or 1,
  "reason": "a brief explanation of the judgment"
}

Do not output any text outside the JSON object.
```

Figure 7: Prompt used for L-J evaluation.

A-MEM. A-MEM is reproduced as an agentic memory organization baseline. Each newly added memory is converted into a structured memory note that contains the original content, contextual description, keywords, tags, and other attributes used by the memory system. When a new memory note is inserted, A-MEM computes semantic similarity and shared-attribute signals to establish links with existing notes, forming a dynamically evolving memory network. Retrieval is performed through semantic matching and association expansion over this memory network. We keep the original memory-note construction, keyword/tag generation, dynamic linking, and association-expanded retrieval mechanisms. Retrieval scale, similarity thresholds, and expansion depth are selected on training or validation data and fixed during testing.

Mem0. Mem0 is reproduced with its memory extraction, update, consolidation, and retrieval pipeline. During memory construction, recent dialogue or interaction history is processed to extract salient candidate memories. The update stage compares each candidate memory with semantically similar existing memories and selects an operation such as ADD, UPDATE, DELETE, or NOOP. Retrieval returns memories that are relevant to the current query or execution state and injects them into the model context. In our reproduction, memory extraction uses the recent history window specified by the original protocol, and memory updating compares candidate facts against retrieved similar memories. The extraction window, retrieval size, and update rules are selected on training or validation data and fixed during testing. Mem0 does not train an additional task-specific policy in our evaluation.

Mem0-Graph. Mem0-Graph enables the graph-enhanced memory mode on top of the Mem0 pipeline. In addition to textual memory extraction and update, it extracts entities and relation triplets from interactions, merges equivalent nodes, detects conflicts, marks invalid or superseded relations, and supports graph-enhanced retrieval. We preserve the two retrieval modes used by the original graph-enhanced memory design. Entity-centric retrieval first locates query-relevant entities and expands neighboring relations, while semantic triplet retrieval returns relation triplets according to semantic similarity with the query. Graph construction and conflict handling are performed according to the original protocol. The graph schema, retrieval scope, and conflict-resolution rules are selected on training or validation data and fixed during testing.

H-MEM. H-MEM is reproduced as a hierarchical memory baseline. We follow the original four-layer memory structure, consisting of the Domain Layer, Category Layer, Memory Trace Layer, and Episode Layer. Each layer stores memories at a different abstraction level and maintains positional indices that point to semantically related lower-level memories. During retrieval, the query is encoded and routed from higher layers to lower layers through the index structure, rather than exhaustively comparing the query with every memory item. The selected top-k memories and user-profile information are then provided to the base model, together with confidence weights when available. H-MEM requires memory construction and hierarchical indexing, but does not train a task-specific memory controller in our setting. Layer construction, routing depth, and top-k retrieval settings are selected on training or validation data and fixed during testing.

StructMem. StructMem is reproduced as a structure-enriched hierarchical memory baseline. We follow its two main stages: event-level binding and cross-event consolidation. In event-level binding, historical interactions are transformed into structured event units through dual-perspective extraction, producing factual entries and relational entries anchored to timestamps. In cross-event consolidation, semantically related events are periodically retrieved and integrated to construct cross-event structures. This design is intended to preserve temporal anchors and event relations for long-horizon reasoning. In our reproduction, we keep the original extraction prompts, temporal anchoring, semantic retrieval, consolidation frequency, and event-merging rules whenever applicable. These construction parameters are selected on training or validation data and fixed during testing. StructMem does not update its construction rules using target test feedback.

Memory-R1. Memory-R1 is reproduced as a training-based memory management baseline. It contains two specialized trainable agents: a Memory Manager and an Answer Agent. The Memory Manager learns structured memory operations, including ADD, UPDATE, DELETE, and NOOP, to maintain an external memory bank. The Answer

Agent retrieves candidate memories and reasons over them to generate the final answer. Following the original protocol, the trainable agents are optimized with outcome-driven reinforcement learning, including PPO or GRPO when required by the original implementation. For Answer Agent training and evaluation, the method retrieves the top-30 relevant memories for each participant, resulting in 60 candidate memories in two-participant dialogue settings. Training and configuration selection use only training or validation data. At test time, the trained Memory Manager and Answer Agent are fixed, and no target test feedback is used for further updates.

MemSkill. MemSkill is reproduced as a skill-based memory learning and evolution baseline. The method consists of a controller, an executor, and a designer. The controller computes state and skill representations with a shared encoder and selects a small set of relevant memory skills. The executor generates structured memory updates conditioned on the current text span, retrieved memories, and selected skills. The designer periodically reviews difficult training cases and revises or expands the skill library. The initial skill library contains Insert, Update, Delete, and Skip.

We use Contriever as the default memory retriever and keep the retrieval and action-selection settings fixed during evaluation: `mem-top-k` $= 20$, `mem-top-k-eval` $= 20$, `action-top-k` $= 3$, `ppo-epochs` $= 2$, `new-action-bias-steps` $= 25$, and `stage-reward-fraction` $= 0.25$. The designer is enabled with `designer-reflection-cycles` $= 3$, `designer-max-changes` $= 3$, `designer-failure-window-epochs` $= 100$, and `designer-failure-pool-size` $= 2000$.

For LoCoMo, MemSkill uses the full-session training mode with `outer-epochs` $= 10$, `inner-epochs` $= 100$, batch size $= 4$, encode batch size $= 64$, and F1 as the reward metric. For ALFWorld, MemSkill uses the fixed-length training mode with `outer-epochs` $= 10$, `inner-epochs` $= 100$, batch size $= 16$, encode batch size $= 8$, and task success as the reward signal. The ALFWorld trajectory construction settings are `pair-a-min` $= 40$, `pair-a-max` $= 60$, `pair-b-size` $= 5$, and `same-type-prob` $= 1.0$. After training, the learned controller, executor behavior, and evolved skill library are fixed during testing.

DeepSeek-V3.2 setting. As noted in Appendix A.1, Memory-R1 is not reported under DeepSeek-V3.2 because its original pipeline requires trainable policy-optimization components for the Memory Manager and Answer Agent, whereas DeepSeek-V3.2 is accessed as an API-based model in our experiments. Reporting Memory-R1 under this setting would require changing its original training protocol. The remaining baselines are evaluated under the same task input, maximum execution step, decoding, and evaluation settings.

Overall, the controlled factors in our comparison are the test split, task input, decoding setting, maximum execution steps, and evaluation script. For baselines that require memory construction, controller learning, policy optimization, or skill evolution, we preserve the corresponding original procedure whenever applicable and report the method-specific settings above. This setup avoids altering the intended mechanism of each baseline while making clear which baseline components are trained or constructed before evaluation.