GeMS-SC: Learning a Cross-Attentive State-Skill Policy for Runtime Graph-Memory Control in Long-Horizon Agents

Anonymous authors
Paper under double-blind review

Abstract

Long-horizon large language model (LLM) agents with graph-structured memory must decide at every execution step whether memory is needed, which graph-native behavior to invoke, and how returned structures should condition later actions. Existing approaches primarily optimize memory representation, retrieval, or generic entry-level operations, leaving this runtime control problem underspecified. We formulate graph-memory invocation as a state-conditioned skill-selection policy and introduce GeMS-SC. Its controller, the Cross-Attentive State-Skill Encoder (CASE), jointly contextualizes execution-state fragments, including the current subtask, previous observation, and recent trajectory, with candidate Graph-Native Memory Skills. A lightweight interaction encoder followed by Skill-to-State Cross-Attention constructs state-conditioned skill representations; a shared scoring head then selects one skill or CONTINUE REGULAR EXECUTION. The selected skill injects an operation protocol specifying the memory objective, graph-tool scope, argument cues, result handoff, and stopping condition, while the frozen base agent retains control over concrete tool calls and task actions. CASE is learned in two stages: weak state-skill alignment from future memory-tool behavior, followed by advantage-weighted trajectory optimization using task success, skill-behavior alignment, memory-operation quality, and execution efficiency. Controlled comparisons with state-aware LLM skill routing and direct graph-tool prompting isolate the effects of learned selection and intermediate skill guidance. Across Qwen3-8B-Instruct and Llama-3.2-3B-Instruct on LoCoMo, LongMemEval, ALFWorld, and MemoryAgentBench, GeMS-SC is competitive with the strongest structured-memory baseline on LoCoMo and yields consistent gains on cross-session and unseen interaction metrics, with its largest improvement on dynamic memory maintenance.

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). Memory participates directly in this loop by shaping 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

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

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., 2025c). Both lines treat long-term memory as an external state whose representation, access, update, and reuse affect agent execution.

However, these advances still leave the runtime invocation of graph-structured memory under-characterized in multi-step agent execution. With memory organized as nodes, edges, and relational paths, each decision must specify both the structural objects to operate on and the handoff of 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 context. These graph-native behaviors involve different traversal patterns, relation-update procedures, and result formats, and therefore affect downstream reasoning, memory updates, tool-argument construction, and task/environment actions. Because graph-memory calls occur inside the execution loop, an inappropriate invocation may 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 formulate runtime graph-memory invocation as a state-conditioned skill-selection policy and introduce GeMS-SC. Its controller, the Cross-Attentive State-Skill Encoder (CASE), jointly contextualizes execution-state fragments and candidate Graph-Native Memory Skill descriptions, then uses Skill-to-State Cross-Attention and a shared scoring head to parameterize the selection policy. The selected skill injects a concise operation protocol for the next decision step, while the base model remains responsible for concrete memory-tool calls, arguments, and task actions. GeMS-SC assumes an existing graph-memory substrate and addresses runtime control over that substrate. Comparisons among graph memory, structured text, event logs, and hierarchical stores fall outside the scope of this study.

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 learned cross-attentive state-skill policy for step-level memory guidance selection. CASE jointly contextualizes execution-state fragments and skill descriptions, applies Skill-to-State Cross-Attention and a shared scoring head, and is trained through weak state-skill alignment followed by advantage-weighted trajectory optimization.
- We evaluate GeMS-SC on LoCoMo, LongMemEval, ALFWorld, and MemoryAgentBench using two open-weight backbones and controlled state-aware prompting baselines. Results show competitive language-memory performance and the clearest gains on dynamic memory maintenance.

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;

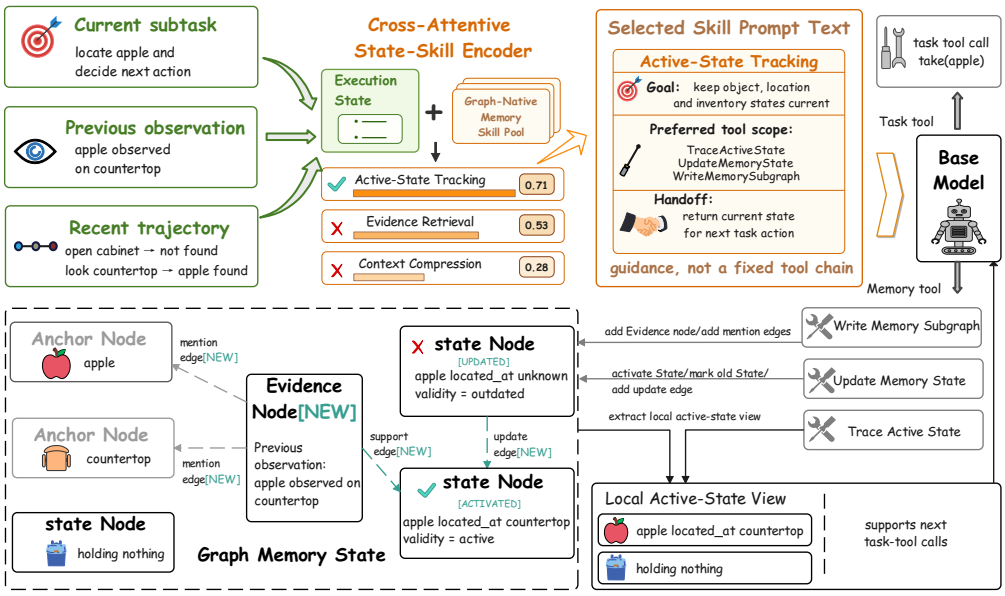

Figure 1: Overview of GeMS-SC. Given the execution state, CASE models state-skill interactions and injects the resulting guidance. The base agent still generates concrete actions. In the illustrated ALFWorld step, ACTIVE-STATE TRACKING guides memory tool calls that add an evidence node, create typed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

Ong et al., 2025). These methods provide useful external memory capabilities, but dependencies, state changes, and evidence chains are less directly addressable unless textual entries and summaries are augmented with an explicit relational schema (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 improve 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., 2025c; 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).

Effective long-term memory requires both suitable stored content and a policy for when and how to operate on it (Tan et al., 2025; Hu et al., 2025a; Zhang et al., 2025c; Kang et al., 2025; Zhang et al., 2025b; Yu et al., 2025; Yuan et al., 2025; Wu et al., 2025b). GeMS-SC differs from prior controllers in its unit of control. Memory-R1 and MemSkill organize decisions around memory entries, operation labels, or reusable memory-skill templates. GeMS-SC selects behaviors defined over typed graph objects, including evidence subgraphs, update paths, support chains, conflict relations, and compact graph views. This granularity lets returned structures 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 observes 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 leaves the base model's action-generation process unchanged. 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 selection policy

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

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

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

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 parameterize the state-skill policy 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 the policy 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

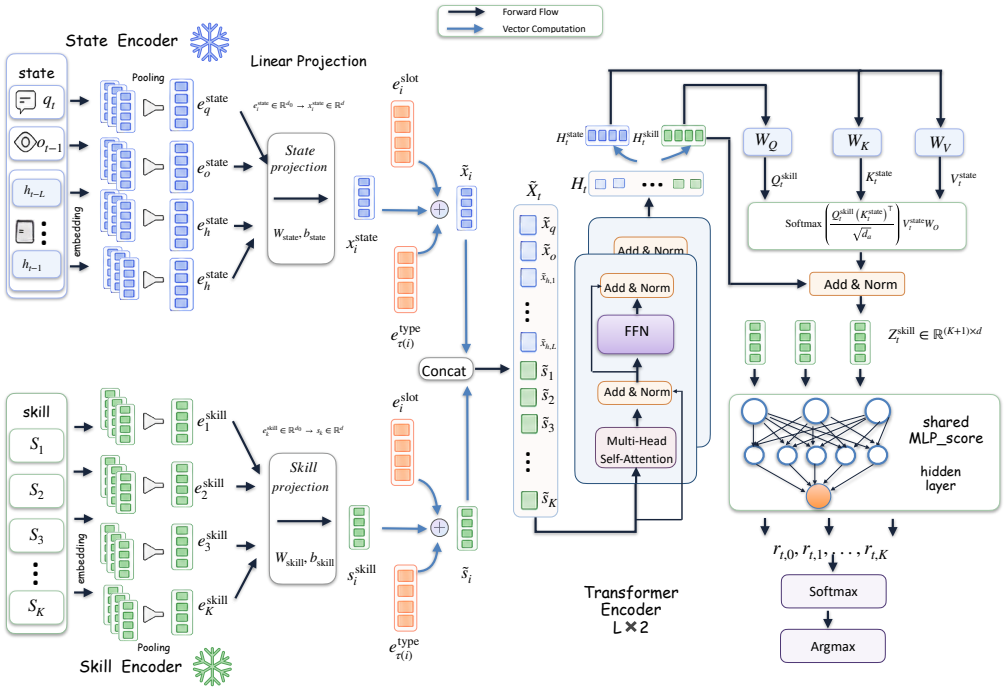

Figure 2: Architecture of CASE. CASE encodes the current execution state and candidate Graph-Native Memory Skills as fragment-level representations, performs joint state-skill contextualization, and uses Skill-to-State Cross-Attention to produce state-conditioned skill representations for step-level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

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 Appendix B.2.

The central operation in CASE is Skill-to-State Cross-Attention, through which each candidate skill attends explicitly to the current execution state. Skill representations serve as queries, while state representations serve as keys and values:

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

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:

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

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

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

Conditioning every score on the current subtask, previous observation, and recent trajectory allows the same skill to receive different scores within a task. CASE can therefore vary its guidance at the decision-step level.

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 specifies an operation protocol for the current decision step: the memory goal, preferred graph-native tool scope, argument-construction cues, output handoff, and stopping condition. Concrete tool execution and call ordering remain with the base model.

CASE selects the guidance appropriate for the current execution state, and graph-native memory tools carry out the corresponding 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. Table 1 makes the complete selection space explicit; Appendix D provides the full field format and a representative execution prompt.

Table 1: Candidate Graph-Native Memory Skills. CASE selects one skill at each decision step; it selects again after the next observation, permitting sequential but not simultaneous skill use.

Skill	Applicable state and memory objective
Continue Regular Execution	Existing context or task tools suffice; inject no memory guidance.
Evidence-Grounded Retrieval	Historical evidence or support paths are needed; recover a grounded evidence subgraph.
Active-State Tracking	The latest valid entity, preference, task, or environment state is needed; trace update relations.
Episodic State Writing	A new fact, observation, tool result, or intermediate state should be connected to graph memory.
Conflict-Aware Maintenance	New evidence supersedes or conflicts with stored memory; update validity while preserving provenance.
Context Selection and Compression	A local graph view is too large; preserve key evidence chains and constraints in a compact view.
Pattern Abstraction	Recurrent supported behavior or experience should be promoted into a reusable insight.
Answer-and-Store Consolidation	The task is complete or nearly complete; store the result, key evidence, and completion state.

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, a fixed labeling procedure examines a short future window containing memory-tool calls, returned observations, and their downstream use, then assigns a weak skill label $\hat{S}_t$. Low-confidence or behavior-inconsistent labels are discarded. CASE is trained on the retained labels with a cross-entropy objective:

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

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 optimizes the CASE selection policy using feedback from complete execution trajectories. Given the i-th execution trajectory Γ_i, we compute a trajectory-level composite feedback:

$$R_i = \lambda_1 R_i^{\text{task}} + \lambda_2 R_i^{\text{align}} + \lambda_3 R_i^{\text{memory}} + \lambda_4 R_i^{\text{eff}}. \tag{11}$$

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 Appendix F.3.

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

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

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. We define the trajectory-level average log-probability of the sampled skills as

$$\bar{\ell}_i(\psi) = \frac{1}{T_i + 1} \sum_{t=0}^{T_i} \log \pi_\psi \left(S_{i,t} \mid X_{i,t}, \mathcal{S} \right). \tag{13}$$

$$\mathcal{L}_{\text{CASE}} = -\mathbb{E}_{\Gamma_i \in \mathcal{B}} \left[A_i \, \bar{\ell}_i(\psi) \right] - \lambda_{\text{ent}} \mathcal{H}_{\text{ent}} + \lambda_{\text{reg}} \|\psi\|_2^2. \tag{14}$$

Here, $\mathcal{H}_{\text{ent}}$ 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.

Table 2: Main results with Qwen3-8B-Instruct, reported as means over three independent runs. Appendix A.1 reports standard deviations. Bold and underline denote the best and second-best means. Single, Multi, Temporal, Open, and Overall denote the LoCoMo categories and weighted overall score; AR, TTL, LRU, and SF denote the MemoryAgentBench categories.

Method	Long-Term Language Memory						Dynamic Interaction and Memory Maintenance							
	LoCoMo L-J↑					LongMem Eval	ALFWorld SR↑			MemoryAgentBench↑				
	Single	Multi	Temp.	Open	Overall		Seen	Unseen	Avg.	AR	TTL	LRU	SF	Overall
ReAct	38.4	30.6	27.9	35.0	34.6	33.9	31.8	25.2	28.5	30.6	26.0	23.8	9.9	22.6
MemoryBank	41.3	32.4	29.7	37.6	37.0	36.9	34.0	28.3	31.2	32.5	26.4	22.0	12.2	23.3
A-MEM	43.0	34.9	31.4	39.7	38.9	39.1	35.9	29.4	32.7	33.8	28.3	23.5	13.6	24.8
Mem0	45.6	36.7	33.9	41.0	41.2	42.2	38.2	32.6	35.4	31.9	24.7	20.6	11.3	22.1
Mem0-Graph	47.2	38.0	35.6	42.6	42.8	44.2	40.0	34.2	37.1	37.3	29.7	25.6	14.8	26.9
H-MEM	48.8	39.6	37.0	43.6	44.3	44.9	41.8	35.6	38.7	39.1	31.1	29.2	16.9	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	49.3	**43.9**	41.4	<u>45.9</u>	46.5	50.1	42.9	36.9	39.9	41.3	34.7	<u>34.9</u>	20.9	33.0
MemSkill	49.2	40.1	37.6	43.8	44.8	<u>52.1</u>	**46.5**	<u>39.4</u>	<u>43.0</u>	<u>43.5</u>	**37.4**	34.6	<u>21.9</u>	<u>34.4</u>
GeMS-SC	<u>50.1</u>	<u>43.5</u>	<u>41.7</u>	**46.8**	<u>46.9</u>	**52.8**	<u>45.6</u>	**41.1**	**43.4**	**49.8**	<u>36.8</u>	**36.1**	**25.6**	**37.1**

Common evaluation setting. For the main results, we use Qwen3-8B-Instruct (Qwen Team, 2025) as the base model for all methods. Within each backbone, we standardize the test split, task input, maximum execution steps, decoding, and evaluation scripts, while preserving each baseline's defining memory representation, trainable components, and optimization procedure. GeMS-SC trains only CASE. The LoCoMo-trained CASE is used for LoCoMo and LongMemEval, and the ALFWorld-trained CASE for ALFWorld and MemoryAgentBench; target test instances are never used for training. Appendices A.2, A.5, and H report the second-open-weight-backbone results, DeepSeek-V3.2 (DeepSeek-AI, 2025) results, and method-specific adaptations.

CASE training. The LoCoMo warm-up uses 128 trajectories (584 decision steps; 552 retained weak labels), followed by 247 feedback-optimization trajectories (1,268 steps). The corresponding ALFWorld stages use 320 trajectories each (2,864/3,096 steps; 2,617 retained labels in warm-up). We use AdamW, batch size 16, three warm-up epochs, five feedback epochs, learning rate 10^{-4}, and entropy/L2 weights $0.01/10^{-4}$. Complete construction and supervision details are in Appendices B and F.

4.2 Main Results

Table 2 reports three-run means under Qwen3-8B-Instruct. Relative to the strongest non-GeMS-SC method, GeMS-SC is 0.2 points lower on LoCoMo Overall and numerically higher by 0.7, 0.4, and 2.7 points on LongMemEval, ALFWorld Avg., and MemoryAgentBench Overall. The first two positive margins are small; the clearest difference is on MemoryAgentBench. StructMem remains strongest on LoCoMo Temporal, while MemSkill remains strongest on ALFWorld Seen and MemoryAgentBench TTL.

The second open-weight backbone, Llama-3.2-3B-Instruct (Meta AI, 2024), reproduces this pattern (Appendix A.2): the corresponding differences are $+0.3$, $+0.6$, $+0.5$, and $+2.9$. Under both backbones, GeMS-SC improves ALFWorld Unseen by 1.7 points and leads on MemoryAgentBench AR, LRU, and SF. We therefore characterize GeMS-SC as competitive on static language memory, modestly stronger on cross-session and unseen interaction metrics, and most clearly improved on dynamic memory maintenance.

4.3 Ablation Studies

Table 3 separates learned state-conditioned selection from prompting over the same substrate. SA-SR receives the same state and skill descriptions as CASE, selects one skill (or the null skill), and uses the same base model for action generation. SA-DTP removes CASE and the skill layer, exposing the state and graph-tool descriptions directly to the base model. Graph memory, tools, base

Table 3: Controlled variants with Qwen3-8B-Instruct. State-Aware Skill Router (SA-SR) and State-Aware Direct Tool Prompting (SA-DTP) are three-run prompting baselines; other rows are component ablations. Skill-State CA denotes Skill-to-State Cross-Attention.

Variant	LoCoMo Overall	LongMem Eval	ALFWorld Avg.	MAB Overall
CASE (Full Model)	**46.9**	**52.8**	**43.4**	**37.1**
SA-SR	43.8	49.2	39.0	33.2
SA-DTP	42.1	47.5	37.4	31.6
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-State CA	39.9	43.2	38.5	31.6

model, execution budget, decoding, and evaluation are otherwise fixed. CASE exceeds SA-SR by 3.1–4.4 points across the four aggregates, while SA-SR exceeds SA-DTP by 1.6–1.7 points. The same ordering holds with Llama-3.2-3B-Instruct (Appendix A.3). CASE also performs selection through a local forward pass, whereas SA-SR requires an additional LLM request.

Graph-CRUD preserves the graph state but replaces relation-aware tools with generic node/edge edits, so its drop isolates the proposed tool interface within the same graph substrate. Comparisons among alternative memory representations remain outside this ablation. Removing skill guidance, CASE learning, or Skill-to-State Cross-Attention also degrades all aggregates. Together, the controlled comparisons support separate roles for the graph-native interface, intermediate skill guidance, and learned state-conditioned selection. Complete category-level results and variant definitions are in Appendix A.6.

4.4 Efficiency and Context Overhead

On LoCoMo, GeMS-SC uses 3.2k context tokens (12.9% of ReAct) and 2.2 memory calls per instance. It uses less context than Mem0-Graph and MemSkill and fewer memory calls than Memory-R1 and MemSkill, so its gains are not explained by longer contexts or repeated invocation alone. Appendix A.4 reports the complete comparison.

5 Conclusion

GeMS-SC treats graph-memory invocation as step-level control over an existing graph substrate. CASE parameterizes a cross-attentive state-skill policy and learns it through weak alignment followed by trajectory feedback, while leaving concrete tools and actions to the base model. Across two open-weight backbones, GeMS-SC is competitive on static language memory and shows its largest relative improvement on dynamic memory maintenance. Controlled prompting baselines and component ablations isolate the contributions of the graph-native interface, intermediate skill guidance, and learned state-conditioned selection.

Limitations

The skill inventory is manually designed, and one-skill-per-step selection cannot cover simultaneous or unseen domain-specific demands without sequential selection or skill expansion. GeMS-SC assumes a graph substrate; Graph-CRUD provides no comparison with structured text, logs, or hierarchical stores. Weak labels and trajectory feedback capture observable behavior and utility without assigning causal credit to individual invocations. We also do not systematically inject graph corruption, and the benchmarks do not fully cover privacy, safety, or long-term deployment.

AI USE STATEMENT

Generative AI tools were used for language polishing, LaTeX formatting, code editing and debugging, and submission-readiness review, including checks of manuscript and statistical-code consistency. They were not used to generate the reported experimental measurements. All AI-assisted changes and outputs were reviewed and verified by the authors, who take responsibility for the accuracy and integrity of this work.

ETHICS STATEMENT

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.

In our experiments, we use existing research benchmarks and model outputs under their original access conditions, and do not collect new personal data from users. We do not store personally identifying information beyond what is already required by the benchmark protocols. The author-side verification of L-J decisions is used only to assess evaluation reliability; no external annotators, crowdworkers, or human subjects are recruited.

REPRODUCIBILITY STATEMENT

The appendices document the implementation and evaluation information available in the manuscript. Appendix B specifies CASE input construction and model parameters; Appendices C, D, and E define the graph-memory schema, skills, tool interfaces, and algorithms; Appendix F reports training data, weak-label construction, hyperparameters, and feedback computation; and Appendices A.1, A.2, A.3, A.8, G, and H provide complete repeated-run results, cross-backbone and prompting controls, selection and failure analyses, evaluation details, and baseline adaptations.

REFERENCES

Petr Anokhin, Nikita Semenov, Artyom Sorokin, Dmitry Evseev, Andrey Kravchenko, Mikhail Burtsev, and Evgeny Burnaev. AriGraph: Learning knowledge graph world models with episodic memory for LLM agents. *arXiv preprint arXiv:2407.04363*, 2024. URL https://arxiv.org/abs/2407.04363.

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

DeepSeek-AI. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. *arXiv preprint arXiv:2512.02556*, 2025. URL https://arxiv.org/abs/2512.02556.

Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. From RAG to memory: Non-parametric continual learning for large language models. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 21497–21515. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/gutierrez25a.html.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. 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)*, pp. 32779–32798, Vienna, Austria, July 2025a. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1575. URL https://aclanthology.org/2025.acl-long.1575/.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=DT7JyQC3MR. 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, Zhenrong Cheng, Xuanbo Fan, Jiaxin Guo, Xinlei Yu, Zhenhong Zhou, Zewen Hu, Jiahao Huo, Junhao Wang, Yuwei Niu, Yu Wang, Zhenfei Yin, Xiaobin Hu, Yue Liao, Qiankun Li, Kun Wang, Wangchunshu Zhou, Yixin Liu, Dawei Cheng, Qi Zhang, Tao Gui, Shirui Pan, Yan Zhang, Philip Torr, Zhicheng Dou, Ji-Rong Wen, Xuanjing Huang, Yu-Gang Jiang, and Shuicheng Yan. Memory in the age of AI agents. *arXiv preprint arXiv:2512.13564*, 2025b. URL https://arxiv.org/abs/2512.13564.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory OS of AI Agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25961–25970, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1318. URL https://aclanthology.org/2025.emnlp-main.1318/.

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

Rui Li, Zeyu Zhang, Xiaohe Bo, Zihang Tian, Xu Chen, Quanyu Dai, Zhenhua Dong, and Ruiming Tang. CAM: A constructivist view of agentic memory for LLM-based reading comprehension. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2510.05520. NeurIPS 2025.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 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)*, pp. 13851–13870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

Meta AI. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. Meta AI Blog, 2024. URL https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/. Accessed 2026-09-16.

Kai Tzu-iunn Ong, Namyoung Kim, Minju Gwak, Hyungjoo Chae, Taeyoon Kwon, Yohan Jo, Seung-won Hwang, Dongha Lee, and Jinyoung Yeo. 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)*, pp. 8631–8661, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.435. URL https://aclanthology.org/2025.naacl-long.435/.

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

Qwen Team. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025. URL https://arxiv.org/abs/2501.13956.

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

pp. 33136–33152, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1683. URL https://aclanthology.org/2025.emnlp-main.1683/.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2010.03768.

Haoran Sun, Shaoning Zeng, and Bob Zhang. 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)*, pp. 341–350, Rabat, Morocco, March 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.15. URL https://aclanthology.org/2026.eacl-long.15/.

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. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8416–8439, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.413. URL https://aclanthology.org/2025.acl-long.413/.

Bo Wang, Weiyi He, Shenglai Zeng, Zhen Xiang, Yue Xing, Jiliang Tang, and Pengfei He. Unveiling privacy risks in LLM agent memory. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25241–25260, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1227. URL https://aclanthology.org/2025.acl-long.1227/.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025a. URL https://arxiv.org/abs/2410.10813.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. ReSum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025b. URL https://arxiv.org/abs/2509.13313.

Yaxiong Wu, Yongyue Zhang, Sheng Liang, and Yong Liu. SGMem: Sentence graph memory for long-term conversational agents. *arXiv preprint arXiv:2509.21212*, 2025c. URL https://arxiv.org/abs/2509.21212.

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

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=FiM0M8gcct. NeurIPS 2025 Poster.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent. *arXiv preprint arXiv:2507.02259*, 2025. URL https://arxiv.org/abs/2507.02259.

Qianhao Yuan, Jie Lou, Zichao Li, Jiawei Chen, Yaojie Lu, Hongyu Lin, Le Sun, Debing Zhang, and Xianpei Han. MemSearcher: Training LLMs to reason, search and manage memory via end-to-end reinforcement learning. *arXiv preprint arXiv:2511.02805*, 2025. URL https://arxiv.org/abs/2511.02805.

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

Guibin Zhang, Muxin Fu, and Shuicheng Yan. MemGen: Weaving generative latent memory for self-evolving agents. *arXiv preprint arXiv:2509.24704*, 2025b. URL https://arxiv.org/abs/2509.24704.

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

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 2025c. doi: 10.1145/3748302. URL https://doi.org/10.1145/3748302.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://arxiv.org/abs/2305.10250.

A Supplementary Experimental Results

A.1 Complete Three-Run Results under Qwen3-8B-Instruct

Tables 4–6 report mean ± standard deviation over three independent runs for every baseline and primary metric. All runs use the same splits, execution limits, decoding, and evaluation scripts.

Table 4: Three-run long-term language-memory results under Qwen3-8B-Instruct.

Method	Single	Multi	Temp.	Open	Overall	LongMemEval
ReAct	38.4±1.2	30.6±0.6	27.9±0.9	35.0±0.7	34.6±1.0	33.9±0.5
MemoryBank	41.3±0.7	32.4±1.0	29.7±0.5	37.6±0.8	37.0±0.6	36.9±0.9
A-MEM	43.0±0.5	34.9±0.8	31.4±1.1	39.7±0.6	38.9±0.9	39.1±0.7
Mem0	45.6±0.9	36.7±0.5	33.9±0.7	41.0±1.0	41.2±0.6	42.2±0.8
Mem0-Graph	47.2±0.6	38.0±1.1	35.6±0.8	42.6±0.5	42.8±0.9	44.2±0.7
H-MEM	48.8±1.0	39.6±0.7	37.0±0.9	43.6±1.1	44.3±0.5	44.9±0.8
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
Memory-R1	49.3±0.8	43.9±1.2	41.4±0.6	45.9±0.9	46.5±0.7	50.1±0.5
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
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

Table 5: Three-run ALFWorld results under Qwen3-8B-Instruct.

Method	Seen	Unseen	Avg.
ReAct	31.8±0.8	25.2±1.1	28.5±0.6
MemoryBank	34.0±1.1	28.3±0.4	31.2±0.7
A-MEM	35.9±0.4	29.4±1.0	32.7±0.8
Mem0	38.2±1.2	32.6±0.6	35.4±0.9
Mem0-Graph	40.0±0.8	34.2±1.2	37.1±0.5
H-MEM	41.8±0.6	35.6±1.0	38.7±1.2
StructMem	43.4±0.9	37.5±0.9	40.5±0.6
Memory-R1	42.9±1.0	36.9±0.8	39.9±1.1
MemSkill	46.5±1.2	39.4±1.3	43.0±0.9
GeMS-SC	45.6±0.7	41.1±0.4	43.4±1.0

Table 6: Three-run MemoryAgentBench results under Qwen3-8B-Instruct.

Method	AR	TTL	LRU	SF	Overall
ReAct	30.6±0.9	26.0±0.4	23.8±0.7	9.9±1.3	22.6±0.8
MemoryBank	32.5±0.5	26.4±0.8	22.0±1.2	12.2±0.6	23.3±0.9
A-MEM	33.8±1.2	28.3±0.6	23.5±0.5	13.6±0.9	24.8±0.7
Mem0	31.9±0.7	24.7±1.1	20.6±0.8	11.3±0.5	22.1±1.0
Mem0-Graph	37.3±1.0	29.7±0.7	25.6±0.9	14.8±0.6	26.9±0.8
H-MEM	39.1±0.7	31.1±0.9	29.2±0.5	16.9±0.8	29.1±1.1
StructMem	42.0±1.1	32.5±0.8	33.8±0.4	19.4±0.7	31.9±1.0
Memory-R1	41.3±0.6	34.7±0.9	34.9±1.2	20.9±0.5	33.0±0.7
MemSkill	43.5±0.7	37.4±1.0	34.6±0.6	21.9±0.8	34.4±1.1
GeMS-SC	49.8±0.9	36.8±0.5	36.1±0.8	25.6±1.1	37.1±0.6

The standard-deviation ranges overlap for GeMS-SC and StructMem on LoCoMo Overall and for GeMS-SC and MemSkill on LongMemEval and ALFWorld Avg. These descriptive intervals are not hypothesis tests; we therefore treat the corresponding differences as close or small-margin results. The MemoryAgentBench Overall margin is larger and is consistent across runs.

A.2 Complete Results under Llama-3.2-3B-Instruct

Tables 7–9 repeat the complete evaluation using Llama-3.2-3B-Instruct as a second open-weight backbone.

Table 7: Three-run long-term language-memory results under Llama-3.2-3B-Instruct.

Method	Single	Multi	Temp.	Open	Overall	LongMemEval
ReAct	31.2±1.0	22.8±0.7	20.5±1.1	27.4±0.6	27.2±0.9	25.3±0.8
MemoryBank	33.9±0.8	24.7±1.1	22.1±0.6	29.9±0.9	29.5±0.7	27.8±1.0
A-MEM	35.7±0.6	26.8±0.9	23.3±1.2	31.4±0.7	31.2±1.0	29.6±0.8
Mem0	38.1±1.1	28.6±0.7	25.4±0.9	33.0±0.8	33.4±0.6	32.0±0.9
Mem0-Graph	39.4±0.7	30.3±1.0	27.2±0.8	34.5±0.6	34.9±0.9	33.7±0.7
H-MEM	41.2±0.9	31.5±0.8	28.9±1.0	35.1±1.1	36.5±0.6	34.5±0.8
StructMem	42.8±0.3	34.0±0.6	34.7±0.8	36.9±0.5	39.1±0.4	36.8±0.9
Memory-R1	42.0±0.8	35.8±1.1	32.9±0.7	37.7±0.9	38.7±0.8	39.7±0.6
MemSkill	41.5±0.6	31.9±0.5	29.3±0.9	35.5±0.7	36.8±0.8	41.4±0.6
GeMS-SC	42.9±0.4	35.9±0.8	33.8±0.7	38.4±0.6	39.4±0.9	42.0±1.1

Table 8: Three-run ALFWorld results under Llama-3.2-3B-Instruct.

Method	Seen	Unseen	Avg.
ReAct	23.1±1.2	17.6±0.5	20.4±0.7
MemoryBank	25.4±0.5	20.3±0.8	22.9±1.1
A-MEM	26.9±0.5	21.7±1.1	24.3±0.9
Mem0	29.3±1.0	24.2±0.7	26.8±1.1
Mem0-Graph	31.0±0.8	25.6±1.1	28.3±0.5
H-MEM	32.5±0.7	27.4±0.9	30.0±1.2
StructMem	34.2±1.0	29.1±0.7	31.7±0.8
Memory-R1	33.7±0.9	28.3±1.0	31.0±1.1
MemSkill	37.3±1.1	30.9±1.2	34.1±1.0
GeMS-SC	36.5±0.8	32.6±0.5	34.6±0.9

Table 9: Three-run MemoryAgentBench results under Llama-3.2-3B-Instruct.

Method	AR	TTL	LRU	SF	Overall
ReAct	22.4±1.0	18.3±0.6	16.1±0.9	2.1±1.2	14.7±0.8
MemoryBank	24.1±0.6	19.0±0.9	14.2±1.0	4.5±0.7	15.5±0.5
A-MEM	25.3±1.2	20.5±0.7	15.9±0.6	5.4±1.0	16.8±0.8
Mem0	23.1±0.8	17.8±1.2	12.7±0.9	3.3±0.6	14.2±1.0
Mem0-Graph	28.9±1.0	22.3±0.7	18.2±0.8	6.9±0.6	19.1±0.9
H-MEM	30.8±0.8	23.7±1.0	21.8±0.6	8.8±0.9	21.3±1.1
StructMem	33.5±1.1	25.2±0.9	26.1±0.5	11.5±0.8	24.1±1.0
Memory-R1	32.6±0.7	27.4±0.8	27.3±1.2	12.9±0.5	25.1±0.7
MemSkill	34.8±0.9	30.1±0.7	26.8±0.8	13.8±0.6	26.4±1.0
GeMS-SC	41.9±1.0	29.4±0.6	28.2±0.7	17.5±1.0	29.3±0.8

Across Qwen3-8B-Instruct and Llama-3.2-3B-Instruct, the differences from the strongest alternative are respectively $-0.2/+0.3$ on LoCoMo Overall, $+0.7/+0.6$ on LongMemEval, $+0.4/+0.5$ on ALFWorld Avg., and $+2.7/+2.9$ on MemoryAgentBench Overall. The subtask pattern is also stable: GeMS-SC gains 1.7 points on ALFWorld Unseen under both backbones and leads on MemoryAgentBench AR, LRU, and SF, while StructMem leads LoCoMo Temporal and MemSkill leads ALFWorld Seen and MemoryAgentBench TTL.

A.3 State-Aware Prompting Baselines

Table 10: State-aware prompting baselines versus CASE. Results are mean ± standard deviation over three runs.

Backbone	Method	LoCoMo Overall	LongMemEval	ALFWorld Avg.	MAB Overall
Qwen3-8B-Instruct	State-Aware Skill Router	43.8±0.7	49.2±1.0	39.0±0.9	33.2±0.6
	State-Aware Direct Tool Prompting	42.1±0.9	47.5±1.1	37.4±1.0	31.6±0.8
	CASE	46.9±0.8	52.8±1.2	43.4±1.0	37.1±0.6
Llama-3.2-3B-Instruct	State-Aware Skill Router	35.5±0.8	37.8±1.1	29.8±0.8	24.6±0.7
	State-Aware Direct Tool Prompting	33.2±1.0	35.1±0.9	27.3±1.1	22.0±0.9
	CASE	39.4±0.9	42.0±1.1	34.6±0.9	29.3±0.8

The Skill Router uses the same execution state and skill descriptions as CASE, while Direct Tool Prompting removes the skill layer and exposes the same graph-tool descriptions to the base model. CASE improves over the Skill Router by 3.1–4.4 points under Qwen3-8B-Instruct and 3.9–4.8 points under Llama-3.2-3B-Instruct. The two performance gaps separately quantify the contributions of intermediate skill guidance and learned selection over prompt-based routing.

A.4 Efficiency and Context Overhead

Table 11: Efficiency and context overhead on LoCoMo with Qwen3-8B-Instruct. Tok. is average online context tokens; Ratio is normalized by ReAct; LLM and Mem. are average LLM and memory-tool calls.

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	38.9
Mem0	2.1k	8.5	2.9	1.6	41.2
Mem0-Graph	4.8k	19.4	3.1	2.1	42.8
Memory-R1	3.9k	15.7	3.4	2.5	46.5
MemSkill	5.4k	21.8	3.7	3.0	44.8
GeMS-SC	3.2k	12.9	3.0	2.2	46.9

GeMS-SC's context footprint is lower than Mem0-Graph and MemSkill, and its memory-call count is lower than Memory-R1 and MemSkill. This analysis is limited to the evaluated LoCoMo protocol.

A.5 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.

Table 12: 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.

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 12 gives the complete long-term language-memory results under DeepSeek-V3.2. Every method scores higher than in the Qwen3-8B-Instruct setting, consistent with stronger long-context evidence integration from the base model. GeMS-SC leads Single-hop, Multi-hop, and LongMemEval, ties StructMem on LoCoMo Overall, and ranks second on Open-domain. StructMem retains the highest Temporal score. GeMS-SC therefore remains competitive with a stronger backbone, with its clearest advantages in multi-hop integration and cross-session maintenance.

Table 13: 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.

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 13 reports dynamic interaction and memory-maintenance results under DeepSeek-V3.2. GeMS-SC leads ALFWorld Unseen and Avg., whereas MemSkill leads Seen. On MemoryAgentBench, GeMS-SC leads AR, LRU, SF, and Overall; MemSkill leads TTL. This metric-level division matches the Qwen3-8B-Instruct results: GeMS-SC is strongest on cross-session maintenance and unseen interaction, while StructMem and MemSkill retain their advantages on temporal reasoning and test-time learning, respectively.

A.6 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.

Table 14: Complete ablation results on long-term language memory tasks.

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
w/o Skill-to-State Cross-Attention	43.7	37.2	31.6	42.1	39.9	43.2

Table 15: Complete ablation results on dynamic interaction and memory maintenance tasks.

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
w/o Skill-to-State Cross-Attention	40.2	36.8	38.5	43.1	32.5	30.8	19.8	31.6

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 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 14 and 15 show consistent component-level trends across language memory and dynamic interaction tasks. Because Graph-CRUD retains the same graph state, its degradation measures the value of the relation-aware tool interface within this substrate. The experiment does not compare alternative memory representations. Removing skill guidance or replacing CASE with static guidance also weakens performance, and the drops from removing CASE learning and Skill-to-State Cross-Attention quantify the contribution of learned state-skill interaction.

A.7 Training Source Transfer Analysis

We diagnose cross-task transfer by applying a LoCoMo-trained CASE directly to ALFWorld and MemoryAgentBench and comparing it with the main ALFWorld-trained setting. The two variants

differ only in the CASE training source; they share the base model, graph-native memory tools, maximum reasoning steps, decoding settings, and evaluation protocols.

Table 16: 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.

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 16 shows that the LoCoMo-trained CASE remains effective on dynamic interaction tasks. On MemoryAgentBench, it exceeds MemSkill on Overall, AR, LRU, and SF. The transferred gains provide evidence that state-skill matching learned for evidence recovery, state maintenance, and conflict handling generalizes partly across task types.

Training CASE on ALFWorld yields additional gains on ALFWorld Unseen, ALFWorld Avg., and MemoryAgentBench Overall. Action–observation histories, environment changes, and new-observation writing therefore contribute task-specific selection signals. CASE transfers partly across task groups but performs best when its training source matches the target interaction form, which motivates the separate language-memory and dynamic-interaction training used in the main experiments.

A.8 Selection Quality, Sequential Invocation, and Failure Analysis

Weak-label–selection agreement on held-out validation steps has per-skill diagonal values from 0.76 to 0.89. Because weak labels are not human gold labels, we additionally annotate acceptable skill sets for 1,159 decision steps. Table 17 reports both ambiguity (more than one acceptable skill) and selection error (the chosen skill is outside the acceptable set).

Table 17: CASE selection quality against manually annotated acceptable skill sets.

Task type	Steps	Multi-applicable	Selection error
LoCoMo Single-hop	216	18.5%	3.2%
LoCoMo Multi-hop	243	29.2%	5.8%
LoCoMo Temporal	228	33.8%	6.4%
LoCoMo Open-domain	207	25.1%	4.5%
ALFWorld task families	265	31.4%	5.1%
Overall	1,159	27.9%	4.9%

Temporal and multi-hop states are both more ambiguous and more error-prone. The 4.9% aggregate is a decision-step error rate and should not be confused with the share of selector errors among failed trajectories below.

Although CASE activates only one skill per decision step, it selects again after each memory-tool or environment observation. To quantify sequential use, we omit the null skill, collapse consecutive repetitions, and measure trajectories containing at least two distinct non-null skills (Table 18).

Table 18: Sequential non-null skill invocation. AST: Active-State Tracking; CSC: Context Selection and Compression; ASC: Answer-and-Store Consolidation; EGR: Evidence-Grounded Retrieval.

Task group	$\geq$2 skills	Avg. transitions	Most frequent transition
LoCoMo	38.7%	2.41	AST $\rightarrow$ CSC
ALFWorld	46.2%	2.87	ASC $\rightarrow$ EGR
MemoryAgentBench	41.5%	2.63	EGR $\rightarrow$ AST

We further assign one primary cause to a stratified random sample of 259 failed trajectories (Table 19). Selector error is not the dominant source: graph-state and downstream reasoning/action errors together account for 57.9% of failures.

Table 19: Primary failure attribution over 259 failed trajectories.

Primary failure source	Count	Share
Selector error	47	18.1%
Tool-executor error	62	23.9%
Graph-state error	73	28.2%
Downstream reasoning/action error	77	29.7%

Selector errors select outside the annotated acceptable set; tool-executor errors use incorrect tools, arguments, or call order despite reasonable guidance; graph-state errors include stale or missing relations, duplicates, incorrect edges, and conflict-handling failures; downstream errors occur after otherwise reasonable selection and tool execution. Representative cases include stale active states, incorrect node merges, evidence loss during compression, and incorrect downstream actions. These observations diagnose the evaluated protocol. Robustness to deliberately corrupted graphs remains untested.

Illustrative CASE walkthrough. In the ALFWorld example in Figure 1, a new observation invalidates a previous object-location state. The query, latest observation, and recent trajectory therefore raise the score of ACTIVE-STATE TRACKING. CASE injects that guidance; the base model then chooses the concrete memory tools and arguments, receives the updated active-state view, and constructs the subsequent `take(apple)` action. Stage-2 feedback rewards the trajectory-level outcome and guidance–behavior consistency without assigning causal credit to that individual selection.

A.9 STATE-CONDITIONED SKILL SELECTION ANALYSIS

We analyze skill-selection frequencies across benchmarks to test whether CASE varies its guidance with the execution state. Figure 3 reports the non-null Graph-Native Memory Skills; CONTINUE REGULAR EXECUTION is omitted because it injects no guidance.

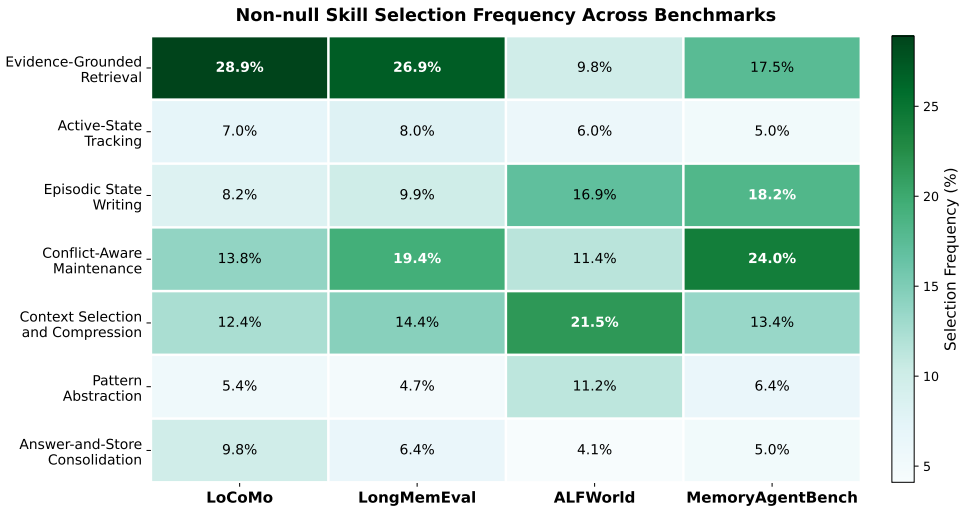

Figure 3: Selection frequency of non-null Graph-Native Memory Skills across benchmarks. Continue Regular Execution is omitted because it is the null skill that injects no additional memory guidance.

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.

Active-State Tracking is activated less frequently across all benchmarks, marking explicit state tracing as specialized intermediate guidance for validating the current state.

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.

The benchmark-specific distributions show that guidance activation varies with task type and execution state. This evidence complements the state-conditioned guidance ablations in the main text.

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 20 lists the fragments used for CASE encoding.

Table 20: Input fragments used by CASE at each decision step.

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

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

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

CASE encoding and scoring use only the selection-side text of S_k. After activation, the execution prompt enters the base-model context and supplies the concrete memory-tool guidance. This separation keeps lengthy tool-use instructions out of the selector input.

B.2 FRAGMENT POOLING AND EMBEDDING CONSTRUCTION

CASE operates on fragment-level representations. Each fragment is the current subtask, previous observation, a recent trajectory record, or the selection-side text of a candidate skill. This representation shortens the CASE input sequence and focuses state-skill interaction on execution-level semantic units.

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

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

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

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

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

$$H_k^{\text{skill}} = E_{\text{text}}(\text{text}_{\text{select}}(S_k)),$$
$$e_k^{\text{skill}} = \text{Pool}(H_k^{\text{skill}}), \qquad e_k^{\text{skill}} \in \mathbb{R}^{d_0}.$$

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:

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

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.

Table 21: Slot embedding settings for CASE input fragments.

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

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 22 summarizes the construction parameters used in all experiments.

Table 22: CASE model-structure and input-construction parameters. Training hyperparameters are reported in Appendix F.1.

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

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 23 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.

Table 23: Node types in the graph memory schema.

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

C.2 Edge Types

Table 24 lists the edge types used to connect memory objects. These edges make evidence sources, state changes, conflicts, and abstractions explicit in the memory graph.

Table 24: Edge 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 $\rightarrow$ anchor
Support edge	An evidential support relation from evidence to a state, insight, or outcome.	evidence $\rightarrow$ state / insight / outcome
Temporal / Update edge	A temporal, update, replacement, or supersession relation between events or states.	old state $\rightarrow$ new state
Conflict edge	A contradiction, invalidation, or resolution relation between evidence or state nodes.	state A $\leftrightarrow$ state B
Abstraction edge	An abstraction relation from evidence nodes, state nodes, or local memory subgraphs to an insight node.	evidence / state $\rightarrow$ insight

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 25 gives the shared attribute fields.

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.

Table 25: Shared attributes for nodes and edges in the graph memory.

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.

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

The tool APIs listed in a skill define its preferred scope. Activated guidance identifies relevant graph-native memory tools, while the base model determines the concrete tool names, arguments, and call order during subsequent action generation. The null skill *Continue Regular Execution* injects no execution prompt.

Table 26: Field format of Graph-Native Memory Skills.

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

D.2 Candidate Graph-Native Memory Skills

Table 1 in Section 3.3 lists the null skill and seven non-null skills together with their applicable states and objectives. The remaining subsections specify the execution-side fields and a complete representative prompt.

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 relevant to the current execution step, including nodes, edge types, support paths, temporal or update relations, and evidence sources.

Tool APIs. This skill preferentially considers the following graph-native memory tools. Their order and combination depend on the current execution state.

`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 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

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

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 27. 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 28.

Table 27: Fields of a graph-native memory tool request.

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.

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.

Table 28: Fields of a graph-native memory tool observation.

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.

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 29–32 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.

Table 29: Graph-native memory tool set: writing tools.

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.

The tool set exposes graph-object-level operations to the base model through a unified invocation interface.

Table 30: Graph-native memory tool set: retrieval 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 31: Graph-native memory tool set: abstraction 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 32: Graph-native memory tool set: maintenance 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.

E.3 Representative Tool Algorithms

The following three algorithms specify internal graph-memory tool execution. During agent execution, the base model generates only the tool name and request; the executor handles graph traversal, node scoring, support tracing, state-path tracking, and conflict resolution.

Algorithm 1 `RetrieveEvidenceSubgraph`

Require: Memory graph $\mathcal{M}_t$, query q_t, relation scope $\mathcal{R}$, maximum depth h, budget B
Ensure: 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 $\mathcal{A}_t = \text{TopK}_{v \in \mathcal{V}_c} s_{\text{anchor}}(v \mid q_t)$.
if $\mathcal{A}_t = \emptyset$ **then**
    return $\mathcal{M}_t$, failure observation.
end if
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.
if $\widehat{\mathcal{G}}_t = \emptyset$ **then**
    return $\mathcal{M}_t$, failure observation.
end if
Construct o_t with status, evidence subgraph, anchors, and provenance.
return $\mathcal{M}_t$, o_t.

Algorithm 2 `TraceActiveState`

Require: Memory graph $\mathcal{M}_t$, target object x, state type c, temporal constraint $\mathcal{T}$
Ensure: Active state v^*, update path $\mathcal{P}^*$, and observation o_t

Locate candidate states $\mathcal{V}_{x,c} = \{v \in \mathcal{M}_t \mid \text{obj}(v) = x, \text{state}(v) = c\}$.
if $\mathcal{V}_{x,c} = \emptyset$ **then**
    return $\mathcal{M}_t$, failure observation.
end if
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.
if $\mathcal{V}_{\text{active}} = \emptyset$ **then**
    return $\mathcal{M}_t$, failure observation.
end if
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.
return $\mathcal{M}_t$, o_t.

Algorithm 3 `ResolveMemoryConflict`

Require: Memory graph $\mathcal{M}_t$, candidate node set $\mathcal{C}$, maintenance constraints Ω, salience decay λ
Ensure: 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.
if $\mathcal{P}_{\text{conflict}} = \emptyset$ **then**
    return $\mathcal{M}_t$, success observation with no graph-state change.
end if
for all $(u, v) \in \mathcal{P}_{\text{conflict}}$ **do**
    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.
end for
Construct o_t with status, kept nodes, superseded nodes, added edges, and provenance.
return $\mathcal{M}_{t+1}$, 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.

F CASE Learning Details

F.1 Training Data and Hyperparameters

Table 33 gives the training data scale used for the two-stage learning procedure defined in the main text. Table 34 reports the optimization hyperparameters.

Table 33: Training data scale for the two-stage learning of CASE. Weak labels are used only in the state-skill alignment warm-up stage.

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

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

Table 34: Training hyperparameters for CASE.

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

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 labeling procedure combines the tool types in the future window, returned observations, downstream use of memory results, and the dominant memory demand reflected by the window. We retain high- and medium-confidence samples for warm-up and discard low-confidence samples or rationales inconsistent with the observed behavior. The resulting labels provide coarse initialization signals for state-skill matching and are absent at inference time.

Table 35: 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.

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

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.

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

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.

Table 35 shows that more than 90% of decision steps in both task groups receive retained weak labels, providing broad initialization coverage. Retention measures confidence-filtered coverage. Selection quality is evaluated independently against acceptable skill sets in Appendix A.8. The discarded samples mainly correspond to windows without a clear memory demand, windows with mixed behaviors but no dominant skill, or rationales unsupported by observed behavior. ALFWorld has a slightly higher discarded rate because its trajectories contain denser tool and environment feedback.

At test time, CASE observes only the current subtask, previous observation, recent trajectory, and candidate skill set, then derives guidance from the learned state-skill distribution. The second-stage trajectory-feedback optimization adjusts this distribution according to task completion, guidance–tool consistency, 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 trajectory-level task completion. For long-term language memory tasks, it is computed from the reference answer or semantic-correctness judgment of the training query. For dynamic interaction tasks, it uses environment feedback or the task-success signal. A single endpoint score is shared by all guidance activations in the trajectory.

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:

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

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 and introduce no 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:

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

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

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

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

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

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 36: Trajectory-level feedback components used for CASE training.

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

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 37. The default setting is used in the main experiments.

Table 37: Feedback weight settings for the sensitivity analysis.

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

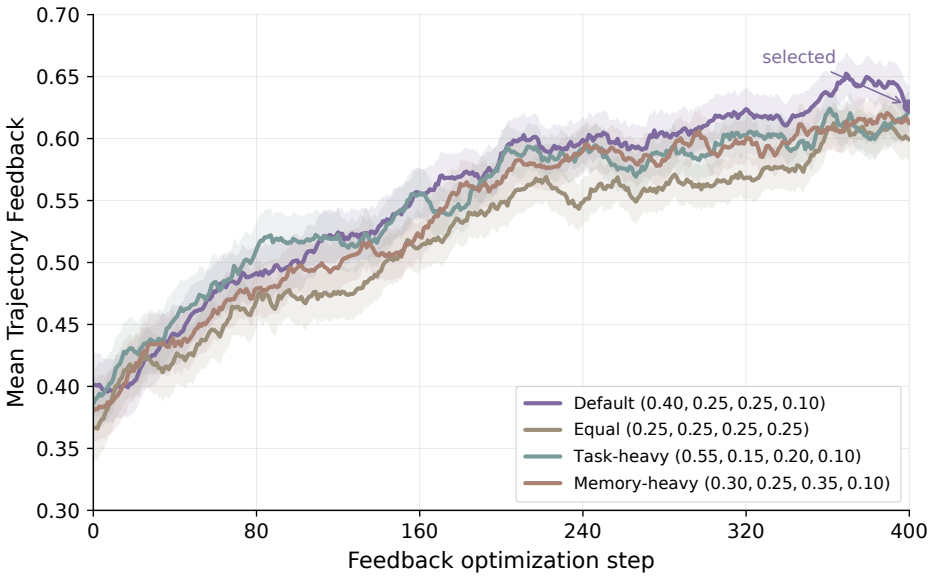

Figure 6: Feedback weight sensitivity. Curves show moving-average trajectory feedback under different weight settings; shaded regions indicate mini-batch-level variation.

Figure 6 shows the moving-average trajectory feedback under different weight settings. The 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 assigns the largest weight to task completion and retains substantial weights for skill-behavior alignment and memory-operation quality. This combination rewards skills associated with successful outcomes, consistent downstream tool behavior, and useful graph-memory states. The smaller efficiency term regularizes excessive tool calls and context expansion.

The default setting yields the most stable late-training feedback trend in Figure 6; the main experiments therefore use $(0.40, 0.25, 0.25, 0.10)$.

G LLM-AS-A-JUDGE EVALUATION PROTOCOL

We use an LLM-as-a-Judge protocol to evaluate semantic correctness on long-term language memory 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 via the fixed API snapshot `gpt-4o-2024-08-06`, with temperature set to 0. The same judge snapshot, prompt, and decoding settings are used 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 38 reports the agreement between L-J decisions and human verification.

Table 38: Agreement between L-J decisions and human verification. κ denotes Cohen's kappa.

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

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.

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.

H Baseline Adaptation and Reproducibility Details

This section describes how each baseline is adapted to the unified evaluation protocol, and provides reproducibility details for the experiments. 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.

Training resources differ across baselines because their defining mechanisms require different memory construction, trainable modules, retrieval policies, and controller optimization. We therefore

report each method's memory representation, required tools or modules, trainable components, training data source, and fixed test-time configuration separately.

Artifacts and intended use. We use existing benchmarks, model checkpoints or APIs, baseline implementations, and evaluation scripts only for academic research evaluation. The main evaluation artifacts include LoCoMo, LongMemEval, ALFWorld, and MemoryAgentBench, which respectively cover long-dialogue memory QA, cross-session long-term memory, multi-step interaction, and incremental agent memory. The evaluated base models are Qwen3-8B-Instruct, Llama-3.2-3B-Instruct, and DeepSeek-V3.2. We do not redistribute benchmark data, model weights, or third-party code. All artifacts are used under their original access conditions and intended research use.

Computational infrastructure and implementation. The main experiments use Qwen3-8B-Instruct, with Llama-3.2-3B-Instruct as a second open-weight validation backbone. Base-model parameters are frozen for GeMS-SC; only CASE is trained. Experiments were conducted on an NVIDIA RTX A6000 GPU. We did not perform large-scale hyperparameter search; hyperparameters were selected on training or validation data and then fixed during testing. For API-based models such as DeepSeek-V3.2, we report results under the same task inputs, maximum execution steps, decoding settings, and evaluation scripts whenever the corresponding baseline can be evaluated without changing its original training protocol.

Use of AI assistants. The AI use disclosure in the main paper applies to the preparation of this appendix. All AI-assisted changes and outputs were reviewed and verified by the authors.

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.

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 routed from higher to lower layers through these indices, avoiding exhaustive comparison 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 and has no task-specific trainable 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 follows event-level binding and cross-event consolidation. Dual-perspective extraction converts interactions into timestamped factual and relational entries; periodic retrieval then integrates related events into cross-event structures. We retain the original extraction prompts, temporal anchoring, semantic retrieval, consolidation frequency, and merging rules whenever applicable. Construction settings are selected on training or validation data, fixed during testing, and never updated from target-test feedback.

Memory-R1. Memory-R1 uses trainable Memory Manager and Answer Agent components. The manager learns ADD, UPDATE, DELETE, and NOOP operations; the answer agent retrieves and reasons over candidate memories. We preserve the original outcome-driven optimization and retrieve the top 30 memories per participant (60 in two-participant dialogues). Training and configuration selection use only training or validation data; both agents are fixed at test time without target-test feedback.

MemSkill. MemSkill uses a shared-encoder controller to select skills, an executor to generate structured memory updates, and a designer to revise the skill library, initialized with Insert, Update, Delete, and Skip. Contriever is the retriever. Memory retrieval top-k, evaluation top-k, and action top-k are $(20, 20, 3)$; PPO epochs, new-action bias steps, and stage-reward fraction are $(2, 25, 0.25)$. Designer reflection cycles, maximum changes, failure-window epochs, and failure-pool size are $(3, 3, 100, 2000)$.

LoCoMo uses full-session training with 10 outer and 100 inner epochs, batch size 4, encoding batch size 64, and F1 reward. ALFWorld uses fixed-length training with the same epoch counts, batch size 16, encoding batch size 8, and task-success reward; its trajectory settings $(\text{pair-a-min}, \text{pair-a-max}, \text{pair-b-size}, \text{same-type-prob})$ are $(40, 60, 5, 1.0)$. The learned controller, executor behavior, and skill library are fixed at test time.

DeepSeek-V3.2 setting. Memory-R1 is omitted under DeepSeek-V3.2 because its original pipeline requires trainable policy-optimization components, whereas our DeepSeek-V3.2 access is API based; including it would change the original training protocol. The remaining baselines share the task input, execution limit, decoding, and evaluation settings.

Across comparisons, we control the test split, task input, decoding, maximum execution steps, and evaluation script. Method-specific construction or training follows the corresponding procedure described above and is completed before evaluation.