MACE: MEMORY-AGENT CO-EVOLUTION WITH ADAPTIVE MEMORY GRAPHS FOR MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM based multiagent systems (MAS) accumulate collaboration experience through task execution, including task conditions, agent actions, intermediate artifacts, repairs, and outcomes. Reusing this experience can improve later collaboration, yet existing memory mechanisms often treat experience as isolated records or static graph context. They therefore miss two requirements of multiagent reuse. The first is functional structure, because a useful experience depends on how conditions, actions, and outcomes support one another and how this support extends across experiences. The second is adaptive use, because the same memory unit may need different selection, format, placement, and strength for different agents and execution states. We propose MACE, a memory agent co evolution framework that jointly organizes experience and learns how agents should use it. MACE introduces MemGoG, which stores experience as functional graph units with typed links across units. It then uses a Need Aware Memory Composer to build a compact working graph, an Adaptive Agent Memory Coupler to render role specific memory patches, and a Feedback Co Evolver to update unit utility, graph links, and agent side use policies after execution. Across eight benchmarks, MACE achieves an average score of 81.11%, improving over the strongest controlled baseline by 2.14%.

1 INTRODUCTION

LLM-based multi-agent systems (MAS) complete complex tasks through interaction and collaboration among agents with different roles (Wu et al., 2023, Hong et al., 2024). As executions accumulate, agents produce a large amount of collaboration experience, including task conditions, execution processes, and final outcomes. These historical experiences encode effective collaboration patterns under different scenarios (Qian et al., 2024, Zhao et al., 2024). Reusing this experience to improve future execution is therefore a central problem for long-running multi-agent systems.

Existing methods usually preserve historical task trajectories through memory mechanisms and retrieve task matching experience for new tasks (Zhang et al., 2026a, Jiang et al., 2026, Tan et al., 2026, Wang et al., 2026). However, multi-agent experience reuse is not merely a matter of finding similar tasks. It requires recovering the structural information in historical experience that can support current decisions. To analyze how current memory mechanisms affect experience reuse, we conduct a systematic empirical study in Sec. 4, examining the reuse process from three perspectives including experience representation, memory composition, and feedback-based updates.

As shown in Fig. 1, we analyze memory reuse from the perspectives of experience structure, memory composition, and execution feedback. The results show that effective multi-agent experience reuse must jointly consider three factors including internal experience structure and cross-experience relations, need-aware memory composition for agents, and continuous optimization driven by execution feedback. Specifically, Fig. 1(a)-(b) show that preserving dependencies among task conditions, actions, and outcomes, and explicitly connecting support relations across experiences, improves the retention of useful information during retrieval. Fig. 1(c) further shows that the effect of memory composition depends on how agents use the retrieved experience. Different roles, stages, and task demands require different memory organizations. Fig. 1(d) shows that jointly updating memory selection and use strategies with execution feedback can continuously improve subsequent task perfor-

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.

mance. These findings indicate that future memory mechanisms should combine structured experience representation, need-aware dynamic composition, and feedback-driven continuous optimization.

Based on these findings, we propose MACE (Memory Agent Co Evolution), a unified framework for structurally representing and dynamically organizing multi-agent collaboration experience while enabling memory and agents to co-evolve. MACE first introduces MemGoG (Memory Graph-of-Graphs), which organizes historical experience as functional memory units. MemGoG preserves internal dependencies among task conditions, actions, and outcomes, and connects reusable units with typed support relations. On this basis, MACE designs a Need-aware Memory Composer that selects task matching memory units according to the current task requirements, agent roles, and execution states, then composes them into a budget-constrained working graph. Furthermore, through the Adaptive Agent-Memory Coupler and Feedback Co Evolver, MACE generates memory patches for different agents and updates memory utility, graph relations, patch format, timing, placement, and strength from execution feedback.

We summarize our contributions as follows. (1) **New Formulation.** We reformulate multi-agent experience reuse as a problem that must jointly model internal dependencies within experiences, cross-experience support relations, and the co-optimization between memory and agent execution feedback. (2) **New Framework.** We propose MACE, a memory agent co evolution framework that uses MemGoG for structured experience representation and MACE Loop for dynamic memory composition, agent-specific adaptation, and feedback-driven optimization. (3) **SOTA Performance.** We evaluate MACE on eight benchmark tasks and show that it improves average performance by 2.14% over strong multi-agent and memory-augmented baselines.

2 PRELIMINARIES

Multi-Agent Collaboration. We consider an LLM-based multi-agent system $\mathcal{A} = \{A_i\}_{i=1}^{N}$ with N agents collaborating to solve a task q. At collaboration step t, the execution state of agent A_i is

$$s_i^t = (q, r_i, \sigma_t, h_i^t, \mathcal{Z}_i^t, e_i^t), \tag{1}$$

where r_i denotes its role, σ_t the collaboration stage, h_i^t its observed interaction history, $\mathcal{Z}_i^t$ its accessible intermediate artifacts, and e_i^t its observed tool responses and environment information.

Memory-Augmented Execution. Let $\mathcal{M}_t$ denote the long-term memory store containing past tasks, collaboration trajectories, and feedback. The system uses s_i^t and $\mathcal{M}_t$ to construct a memory input m_i^t for agent A_i. Using m_i^t as context, the agent produces action a_i^t and artifact z_i^t through

$$(a_i^t, z_i^t) = A_i(s_i^t, m_i^t), \qquad \ell(m_i^t) \le b_i^t, \tag{2}$$

where $\ell(\cdot)$ counts memory tokens and b_i^t is the memory token budget for agent A_i at step t. Generated artifacts are shared through the collaboration workflow. Execution traces and task feedback guide updates to the memory store and subsequent memory use. Our goal is to improve collaborative task performance by adapting memory to each agent's role, stage, and state within the given budget.

3 RELATED WORK

LLM-based Multi-Agent Systems. Multi-agent systems coordinate specialized agents through communication and tools. AutoGen supports programmable conversations (Wu et al., 2023), while MetaGPT organizes role-based collaboration through standardized procedures (Hong et al., 2024). AgentVerse coordinates expert recruitment and collaborative execution (Chen et al., 2024), and multiagent debate revises answers through discussion (Du et al., 2023). GPTSwarm optimizes prompts and connections in computational graphs (Zhuge et al., 2024), while EvoAgent generates agent configurations through evolutionary operators (Yuan et al., 2025). These methods organize collaboration within task execution. Reusing its outcomes across tasks also requires storing and selecting useful experience. Voyager maintains reusable executable skills (Wang et al., 2023), and Experiential Co-Learning extracts shortcuts from software-development trajectories (Qian et al., 2024). MACE studies how dependencies within and between experiences guide reuse across agents.

Memory for Multi-Agent Systems. General agent memory provides foundations for experience reuse. MemGPT manages limited context (Packer et al., 2023), Reflexion stores verbal feedback (Shinn et al., 2023), and ExpeL extracts lessons from past execution (Zhao et al., 2024).

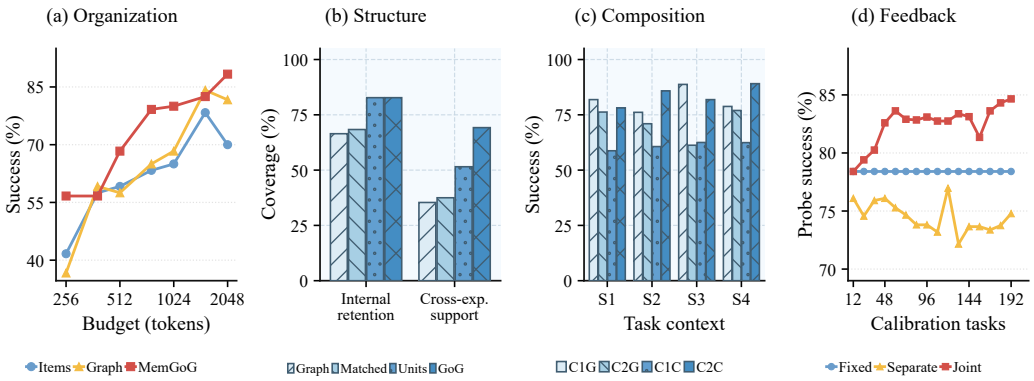

Figure 1. Diagnostics of memory reuse. (a) Task success across memory budgets. (b) Dependency retention and complete support across experiences. (c) Composition and use across four task contexts. C1G, C2G, C1C, and C2C denote composition C1 or C2 rendered as a guide or checklist. (d) Probe success as calibration feedback accumulates.

For multiple agents, LatentMem customizes compact memories to agent contexts (Fu et al., 2026), while MemCollab distills shared constraints from different models' trajectories (Chang et al., 2026). Agentic Context Engineering updates reusable contexts from feedback (Zhang et al., 2026b), and Mem2Evolve couples experience accumulation with tool and expert-agent creation (Cheng et al., 2026). These approaches address experience extraction, adaptation, and evolution. MACE jointly adjusts the composition of functional memory units and their agent-specific presentation using execution outcomes, updating unit utility, relations, and memory use policies.

Graph-Based Memory for Multi-Agent Systems. GraphRAG and HippoRAG organize knowledge for structured retrieval (Edge et al., 2024, Gutiérrez et al., 2024), while A-MEM links and updates memory notes (Xu et al., 2026). MAGMA uses multiple relational views (Jiang et al., 2026), and SAGE combines graph-memory construction with retrieval feedback (Wang et al., 2026). For multi-agent reuse, G-Memory links insight, query, and interaction graphs for role-specific guidance (Zhang et al., 2026a). ConMem connects strategy cards for memory composition and conflict resolution (Tan et al., 2026). These methods establish structured retrieval and adaptive memory as useful designs. MemGoG represents each reusable function as a subgraph of related conditions, actions, and outputs. MACE Loop composes, presents, and updates these linked units.

4 Empirical Study

We investigate how memory preserves task support and how its value depends on agent use. Fig. 1 connects these questions through four diagnostics.

Study setup. We use TAT-QA (Zhu et al., 2021) for the main diagnostics and TabFact (Chen et al., 2019) for replication. Historical memory, development, calibration, and evaluation tasks are separated by source table or document. Paired conditions share content, backbone, workflow, and task states. Memory budgets count tokens supplied across all agents. Appendix A details the protocol.

Memory organization. With memory use fixed, we vary the budget from 256 to 2,048 tokens. Items retrieves independent entries, Graph extracts task matching subgraphs from a unified graph, and MemGoG selects linked functional subgraphs. At 1,024 tokens, Fig. 1(a) reports approximately 80% task success for MemGoG, compared with 68% for Graph and 65% for Items. The gap under a shared budget motivates examining which dependencies each organization preserves during retrieval.

Functional units and their relations. We separate functional boundaries from outer relations by comparing Graph, matched nonfunctional groups (Matched), functional units (Units), and MemGoG (GoG). Matched controls group count, size, overlap, and connectivity. Internal retention measures the fraction of dependency requirements preserved, while cross-experience coverage measures the fraction with complete support. In Fig. 1(b), Units retains approximately 83% of internal dependencies, compared with 68% for Matched. Adding outer relation access to the same units

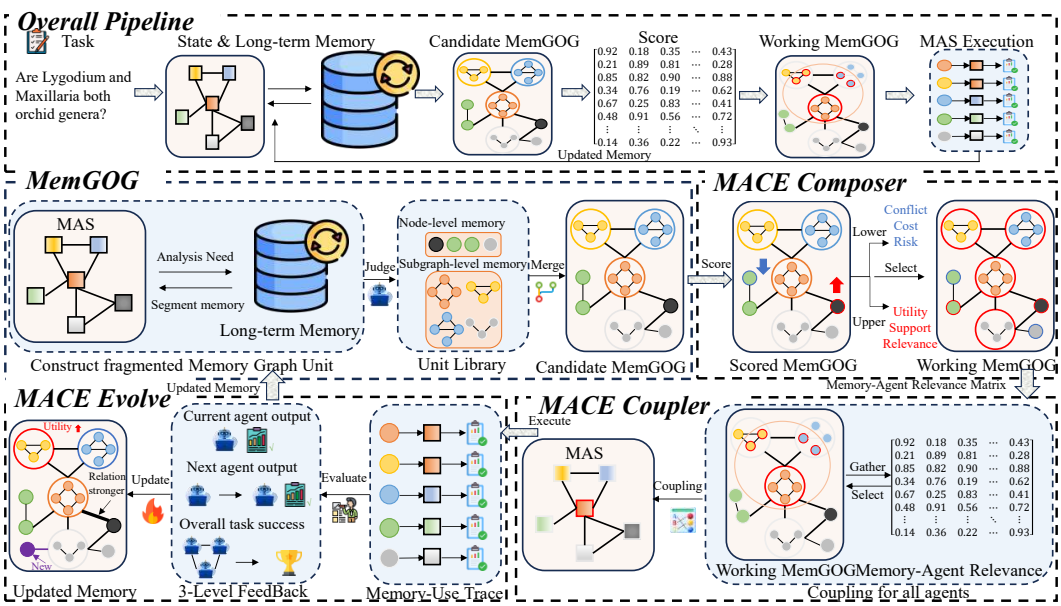

Figure 2. Overview of MACE. MemGoG builds candidate memory units from execution needs and long term memory. The Composer selects a working graph, the Coupler renders agent specific patches, and the Co Evolver updates later memory use from execution feedback.

increases cross-experience coverage from 52% to 69%, while internal retention remains at 83%. These gains motivate functional units with explicit outer relations.

Dependence between composition and use. We test whether the preferred composition changes with its use format. C1 selects task matching units, whereas C2 also retrieves supporting units. Each composition is presented as a guide (G) or checklist (C), with content, recipient, timing, and budget matched across formats. Contexts S1 to S4 cross complete or partial task inputs with local or cross-experience support needs, as defined in Appendix A.1. In Fig. 1(c), C1 leads C2 under the guide format in S1, reaching approximately 82% versus 76%. Under the checklist format, the order reverses to 59% versus 78%. The reversal recurs across contexts, showing that composition value depends on how agents use the selected memory.

Feedback for joint selection. With the memory library fixed, three policies use the same growing calibration history. Fixed freezes the initial joint policy, Separate updates composition and format scores through their marginal returns, and Joint updates each pairing. Policies are evaluated on a fixed probe set whose outcomes stay outside their updates. At 192 calibration tasks, Fig. 1(d) reports approximately 85% probe success for Joint, 78% for Fixed, and 75% for Separate. The comparison with Separate supports retaining pairing information, while the comparison with Fixed supports learning from additional feedback.

These findings motivate MemGoG as a structure for preserving and connecting reusable dependencies. Its identifiable units support joint decisions about composition and agent use. MACE Loop records the outcomes of these decisions and feeds them back to improve subsequent reuse.

5 METHODOLOGY

We present MACE, a memory agent co evolution framework that turns stored collaboration traces into structured memory units, selects a compact working graph for the current team, renders agent specific memory patches, and updates later memory use from execution feedback. As shown in Fig. 2, the method links memory structure, agent use, and post execution updates in one loop. Its key object is a functional memory unit cut from a trace at useful products, checks, repairs, or final decisions. MACE then links units, selects a working graph, renders patches, and updates memory utilities, relations, and agent side use policies.

5.1 MEMORY GRAPH CONSTRUCTION AND COMPOSITION

MEMGOG builds the candidate memory graph used by the Composer. For each agent that receives memory at step t, MACE extracts a need descriptor from the agent state and token budget,

$$d_i^t = \mathrm{Need}(s_i^t, b_i^t), \qquad \mathcal{D}_t = \{d_i^t\}_{i \in \mathcal{I}_t}, \tag{3}$$

where $\mathcal{I}_t$ indexes the agents receiving memory at step t. Each descriptor specifies the task, role, stage, current state, requested memory function, and available budget. Given the descriptors, MEMGOG retrieves collaboration fragments from $\mathcal{M}_t$ and turns them into memory graph units,

$$g_k = (V_k, E_k, \tau_k, f_k, \iota_k, o_k, U_t(g_k), p_k). \tag{4}$$

The terms V_k and E_k define the internal graph of the unit. The type τ_k, function f_k, trigger set ι_k, output o_k, utility $U_t(g_k)$, and source record p_k describe when the unit should be used and what it provides. Segmentation runs over logged agent events. A new unit starts at scope changes, tool artifacts, verification or repair steps, role handoffs, and final answers. Adjacent events are merged only when their role, inputs, output artifact, and scorer outcome match.

The units form the outer nodes of the candidate graph,

$$\mathcal{G}_t^{\mathrm{cand}} = \big(\mathcal{C}_t, \bigcup_{r \in \mathcal{R}} \mathcal{E}_t^{(r)}\big), \qquad \mathcal{E}_t^{(r)} \subseteq \mathcal{C}_t \times \{r\} \times \mathcal{C}_t. \tag{5}$$

The relation set $\mathcal{R}$ contains support, abstraction, operationalization, role projection, conflict, repair, and feedback links. These links are created from unit fields and trace order, connecting outputs to required inputs, routines to instances, plans to actions, shared routines to roles, conflicts to repairs, and outcomes to used units and patches. The Composer scores candidates by combining task fit, utility, and support with conflict, cost, and risk after fixed development set normalization,

$$S_t(g) = \mathbf{1}_3^\top \big(\tilde{\phi}_t^{+}(g) - \tilde{\phi}_t^{-}(g)\big). \tag{6}$$

Here $\tilde{\phi}_t^{+}$ stores normalized task fit, utility, and support, while $\tilde{\phi}_t^{-}$ stores normalized conflict, cost, and risk. After filtering conflicting or high risk suggestions, the Composer keeps the highest scoring units under memory budget B_t. The selected set is $\mathcal{W}_t = \mathrm{Top}_{B_t}(\mathcal{C}_t^{\mathrm{adm}}, S_t)$. Restricting task matching relations to the selected units gives the working graph,

$$\mathcal{G}_t^{\mathrm{work}} = \big(\mathcal{W}_t, \mathcal{E}_t^{\mathcal{D}} \cap (\mathcal{W}_t \times \mathcal{R} \times \mathcal{W}_t)\big). \tag{7}$$

The unit budget B_t controls graph size, while each agent budget b_i^t controls the final rendered memory input that the Coupler delivers to the target agent.

5.2 AGENT MEMORY COUPLING AND FEEDBACK CO EVOLUTION

The working graph is shared, but each agent needs a different view of it. MACE builds a fit matrix between selected units and active agents,

$$\mathbf{F}_t = \big[\kappa_t(g_k, d_i^t, s_i^t)\big]_{g_k \in \mathcal{W}_t,\, i \in \mathcal{I}_t}. \tag{8}$$

The Coupler uses column i to rank units for agent A_i and gather selected content, source records, support links, and routine links into $\mathcal{G}_{i,t}^{\mathrm{view}}$.

The Coupler converts the view into a patch configuration,

$$P_i^t = (c_i^t, \mu_i^t, \varphi_i^t, \delta_i^t, \alpha_i^t, \lambda_i^t, p_i^t), \tag{9}$$

where the fields specify content, mode, format, timing, strength, placement, and source records. Information position can affect its use in long contexts (Liu et al., 2024), motivating placement as a configurable field. Rendering maps each patch into a budgeted input,

$$m_i^t = \mathrm{Render}_{b_i^t}(P_i^t) \in \mathcal{L}(b_i^t), \qquad \mathcal{L}(b) = \{m \mid \ell(m) \le b\}. \tag{10}$$

The agent side state updated by MACE is the use policy behind this patch, not the model weights. For each role and stage, it stores preferred unit types, format, timing, strength, and placement.

Each execution records the used units, patch configuration, role, stage, action, artifact, task result, and resource usage. For a completed trace ξ, local, downstream, and team feedback score the current output, later operations, and final outcome. With feedback levels $\mathcal{H} = \{\mathrm{loc}, \mathrm{down}, \mathrm{team}\}$, the trace return combines quality signals and memory use penalties,

$$R_\xi = \sum_{h \in \mathcal{H}} r_h(\xi) - \|\boldsymbol{p}_\xi\|_1, \tag{11}$$

where $\boldsymbol{p}_\xi = (c_\xi, v_\xi)^\top$ collects nonnegative penalties for resource use and unsuitable memory use.

When a trace completes, the Co Evolver updates the utility values of used units. Let $\mathcal{U}_\xi$ contain those units and let $\boldsymbol{u}_t = [U_t(g_k)]_k$ list current utilities in a fixed order. Using the same order, define $\mathbf{D}_\xi = \mathrm{diag}([\mathbf{1}_{\{g_k \in \mathcal{U}_\xi\}}]_k)$. The return is assigned through a nonnegative credit vector $\boldsymbol{a}_\xi$ supported on the used units. With $\boldsymbol{q}_\xi = R_\xi \boldsymbol{a}_\xi$, the utility update for the used units follows,

$$\boldsymbol{u}_t^+ = \boldsymbol{u}_t + \eta \mathbf{D}_\xi \big(\boldsymbol{q}_\xi - \boldsymbol{u}_t\big), \tag{12}$$

where $\eta \in (0, 1]$ controls the contribution of the completed trace and the mask keeps unused units unchanged. Useful compositions strengthen support links, incompatible suggestions reinforce conflict links, and completed trajectories add new episodes or routines. The same trace updates the agent side use policy by adjusting patch type, format, timing, strength, and placement preferences.

6 Experiments

We evaluate MACE, a Memory-Agent Co-Evolution loop built on MemGoG, across effectiveness, component necessity, efficiency, and robustness. We focus on four experimental questions. **EQ1**. Does MACE improve task-solving effectiveness over direct LLM answering, graph/memory-augmented baselines, and multi-agent workflow baselines? **EQ2**. Are the core MACE components, including MemGoG, the Need-aware Memory Composer, the Adaptive Agent-Memory Coupler, and the Feedback Co-Evolver, necessary? **EQ3**. What LLM token cost, normalized cost, and latency does the MACE loop introduce, and are these costs acceptable relative to its performance gains? **EQ4**. Does MACE remain robust when the serving backbone, network condition, memory budget, and retrieved memory content are perturbed?

6.1 Experimental Setup

Datasets. We evaluate MACE on eight benchmarks spanning six task domains. (1) knowledge-intensive question answering, including MMLU validation (Hendrycks et al., 2020) and MMLU-Pro (Wang et al., 2024). (2) mathematical reasoning, including GSM8K (Cobbe et al., 2021) and AQuA-RAT (Ling et al., 2017). (3) code generation, using HumanEval (Chen et al., 2021). (4) fact verification, using TabFact (Chen et al., 2019). (5) table-based reasoning, using TAT-QA (Zhu et al., 2021). (6) live programming, using LiveCodeBench v6 (LCB v6) (Jain et al., 2025).

Baselines. The comparison includes five baseline families. (1) **Direct**, standard LLM-based reasoning without explicit external memory, graph construction, or retrieval mechanisms. (2) **Graph Memory**, including MAGMA (Jiang et al., 2026), G-Memory (Zhang et al., 2026a), and ConMem (Tan et al., 2026). G-Memory links insight, query, and interaction graphs, while ConMem organizes strategy cards through relations for retrieval and conflict resolution. (3) **SAGE**, graph-enhanced retrieval methods that use a structured knowledge index to guide answer generation (Wang et al., 2026). (4) **General MAS**, comprising Multiagent Debate (MAD) (Du et al., 2023), AgentVerse (Chen et al., 2024), AutoGen (Wu et al., 2023), and EvoAgent (Yuan et al., 2025). (5) **G-MAS**, imported graph-based multi-agent workflow baselines including GPTSwarm (Zhuge et al., 2024), GraphSearch (Yang et al., 2025), and R-GFM (Liu et al., 2026). MAD uses multiple model instances to discuss and revise candidate answers. AgentVerse combines expert recruitment, collaborative decisions, execution, and evaluation. AutoGen supports configurable conversations among agents and tools. EvoAgent generates diverse agent configurations through evolutionary operators.

Sampling Protocol. Local controlled rows use the same prepared task files and scoring scripts. MMLU uses the first 153 validation examples. MMLU-Pro and TabFact use seed-42 samples of 500 examples. GSM8K, HumanEval, AQuA-RAT, TAT-QA, and LiveCodeBench v6 use their available test splits. Imported G-MAS rows follow the supplied attachment protocol.

Table 1. OVERALL PERFORMANCE COMPARISON. Single-run percentage scores. Best results are in **bold** and second-best are underlined. Avg. is the unweighted mean.

Description	Methods	Avg. (↑ / ↓ vs. Direct)	QA & Reasoning				Code		Fact & Table	
			MMLU	MMLU-Pro	GSM8K	AQuA-RAT	HumanEval	LCB v6	TabFact	TAT-QA
LLM	Direct	71.53 (ref.)	88.24	70.60	64.22	79.92	84.15	48.57	73.20	63.32
Graph Memory	MAGMA	69.71 (↓2.54%)	78.43	59.00	65.35	71.26	90.85	48.57	78.80	65.42
Graph Memory	G-Memory	63.28 (↓11.53%)	71.90	55.60	86.81	54.72	80.49	41.14	64.20	51.41
Graph Memory	ConMem	65.95 (↓7.80%)	84.97	76.20	51.86	87.40	66.46	45.71	69.20	45.76
Graph Retrieval	SAGE	78.97 (↑10.40%)	91.50	76.60	92.19	86.22	90.85	42.86	87.00	64.52
General MAS	MAD	77.71 (↑8.65%)	88.89	71.20	91.66	87.40	84.76	47.43	85.60	64.76
General MAS	AgentVerse	76.76 (↑7.31%)	87.58	75.40	84.23	82.68	92.07	46.86	81.00	64.22
General MAS	AutoGen	76.59 (↑7.08%)	88.24	69.00	88.86	84.65	87.80	44.00	85.20	65.00
General MAS	EvoAgent	77.41 (↑8.22%)	88.89	73.40	86.58	86.22	92.07	48.00	80.40	63.68
Multi-Agent	GPTSwarm	74.12 (↑3.62%)	87.60	71.20	92.12	87.40	84.76	19.43	87.60	62.84
Multi-Agent	GraphSearch	73.98 (↑3.43%)	88.80	71.60	92.12	86.61	81.10	22.29	87.40	61.94
Multi-Agent	R-GFM	74.74 (↑4.49%)	88.80	68.60	92.04	87.01	89.02	19.43	87.20	65.84
MACE (Ours)		**81.11 (↑13.40%)**	**92.81**	**78.80**	**92.27**	**87.80**	**93.29**	**49.14**	**87.80**	**66.99**

Metric note. Higher is better. Most datasets report parsed accuracy, TAT-QA reports exact match, and code tasks report pass@1. Imported G-MAS MMLU rows use seed42-500 and local rows use validation first153. MMLU scores for GPTSwarm, GraphSearch, and R-GFM are approximated to the nearest 0.20-point step. Their averages use these approximations.

Calibration and held-out feedback. The local static scores use graph and patch policies finalized before test evaluation. Held-out feedback is logged for audit without updating subsequent queries.

Model endpoints and evaluation boundary. Unless otherwise stated, local rows use DeepSeek-V3 with shared prompts, parsers, retry rules, and call budgets. EQ4 changes the serving endpoint and other deployment factors in separate stress tests.

6.2 PERFORMANCE COMPARISON (EQ1)

To answer EQ1, the twelve baselines with measured or imported scores cover direct answering, graph memory, graph retrieval, general MAS, and imported G-MAS workflows in Table 1. Using the unweighted mean across the eight printed benchmark columns, MACE reaches 81.11%, the highest average score. It exceeds Direct by 9.59 percentage points, producing a 13.40% relative improvement, and remains 2.14 percentage points above the strongest baseline, SAGE. The strongest general MAS baseline is MAD at 77.71%, followed by EvoAgent at 77.41%, AgentVerse at 76.76%, and AutoGen at 76.59%. Among the imported G-MAS methods, R-GFM has the highest average at 74.74%. G-Memory and ConMem average 63.28% and 65.95%, respectively.

Compared with SAGE, MACE increases MMLU-Pro from 76.60% to 78.80%, HumanEval from 90.85% to 93.29%, LiveCodeBench v6 from 42.86% to 49.14%, and TAT-QA from 64.52% to 66.99%. The closest comparisons are GSM8K, with 92.27% for MACE and 92.19% for SAGE, and AQuA-RAT, with 87.80% for MACE and 87.40% for ConMem, MAD, and GPTSwarm. General MAS methods also provide competitive results, including 92.07% for AgentVerse and EvoAgent on HumanEval and 91.66% for MAD on GSM8K. MACE leads all three domain groups, averaging 87.92% on QA & Reasoning, 71.22% on Code, and 77.40% on Fact & Table. Appendix C.1 reports these group averages for all methods.

6.3 ABLATION STUDY (EQ2)

To answer EQ2, we ablate one component at a time while keeping the same LLM backbone, benchmark split, memory budget, and evaluation script as the full MACE system. As shown in Table 2, every ablation reduces the average score from the full model's 81.11%, indicating that the components are complementary. The largest drops come from removing the Agent-Memory Coupler (-5.88 percentage points) and replacing MEMGOG with a fixed memory form (-5.20 percentage points), showing that adaptive graph-structured memory and role-specific coupling drive most of the gain. The Memory Composer and Feedback Co-Evolver also contribute consistent improvements, while fixed patch mode causes the smallest loss (-0.63 percentage points), suggesting that mode adaptation is useful but less central than memory construction and coupling.

6.4 EFFICIENCY ANALYSIS (EQ3)

Table 2. ABLATION RESULTS FOR EQ2. Single-run percentage scores using the metrics in Table 1. Δ Avg. is the percentage-point change from full MACE.

Ablated Component	Variants	Avg.	Δ Avg.	QA & Reasoning				Code		Fact & Table	
				MMLU	MMLU-Pro	GSM8K	AQuA-RAT	HumanEval	LCB v6	TabFact	TAT-QA
Fixed Memory Form	w/o MEMGOG	75.91	-5.20	87.40	72.80	86.80	81.10	88.41	50.00	81.00	59.77
No Composition	w/o Memory Composer	77.58	-3.53	88.24	73.80	88.50	83.46	90.24	51.43	83.20	61.77
Plain Injection	w/o Agent-Memory Coupler	75.23	-5.88	86.27	70.80	85.90	80.31	87.80	50.57	80.80	59.35
No Feedback Loop	w/o Feedback Co Evolver	79.70	-1.41	90.20	75.20	90.45	85.83	92.07	53.71	86.00	64.11
No Mode Adaptation	Fixed Patch Mode	80.48	-0.63	91.50	76.40	90.91	86.61	92.68	**54.29**	86.60	64.88
MACE full (Ours)		**81.11**	**0.00**	**92.81**	**78.80**	**92.27**	**87.80**	**93.29**	49.14	**87.80**	**66.99**

Table 3. EFFICIENCY AND RESOURCE USE FOR EQ3. Token counts are thousands per query, and cost is normalized to Direct.

Method	Avg. Acc. (%)	# Calls	Prompt Tok. (K)	Completion Tok. (K)	Total Tok. (K)	Norm. Cost (×)	Total Time (min)	Latency (s/query)
Direct (LLM)	71.53	**1.00**	**0.54**	**0.06**	**0.60**	**1.00**	**17.13**	**0.22**
MAGMA (Graph Memory)	69.71	**1.00**	0.92	0.06	0.98	1.59	31.43	0.40
SAGE (Graph Retrieval)	78.97	3.00	4.61	0.89	5.50	9.69	190.10	2.41
MACE (Ours)	**81.11**	4.00	6.45	0.41	6.87	11.06	106.78	1.36

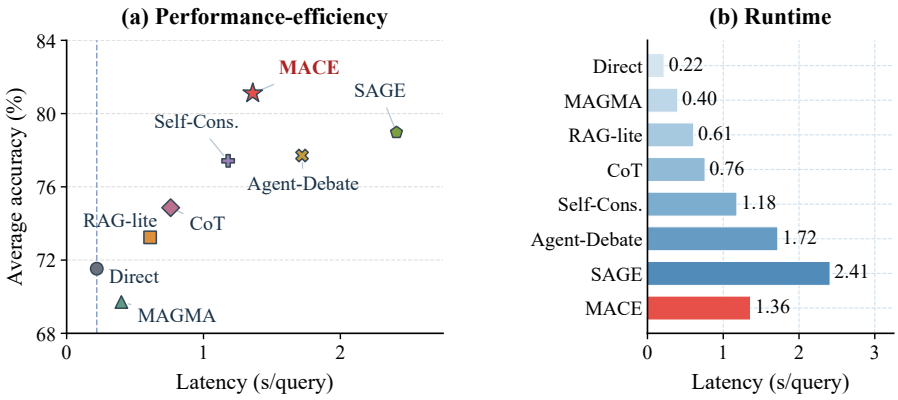

Figure 3. EFFICIENCY FOR EQ3. Accuracy versus latency (left) and runtime (right).

On the EQ1 benchmark mixture, MACE averages four LLM calls and 6.87K tokens per query, completing 4,728 queries in 106.78 minutes (Table 3). Batch elapsed time averages 1.36 seconds per query versus 2.41 for SAGE, including endpoint throughput, batching, output length, and retries. LCB v6 takes 3.76 seconds per query, and the other seven benchmarks remain below 1.82 seconds. Fig. 3 shows the accuracy and runtime comparison.

6.5 ROBUSTNESS ANALYSIS (EQ4)

We vary the backbone, network, memory budget, and retrieval noise one factor at a time, holding task splits, prompts, and scoring fixed. Network tests inject delays and request failures, budget tests prune cells before composition, and retrieval tests add irrelevant, stale, or conflicting candidates under a fixed budget. Fig. 4 reports the four tests.

Under backbone replacement, MACE changes from 81.11% with DeepSeek-V3 as the primary endpoint to 80.04% with Qwen2.5-72B and 79.28% with a local Qwen2.5-7B fallback. The maximum drop is 1.83 percentage points, with all three settings remaining above Direct and SAGE.

Under network fluctuation, scores decline from 80.56% with mild latency jitter to 76.92% under severe instability, which combines delays, timeouts, and memory failures. The severe setting is 4.19 percentage points below clean serving and 5.39 points above Direct. Appendix C.4 reports each perturbation and its result.

Across five representative benchmarks, reducing memory to 75%, 50%, and 25% of its full size yields scores of 85.74%, 84.40%, and 81.76%, respectively, compared with 86.63% at full size. Retaining 75% of memory costs 0.89 percentage points, and the 25% setting remains above Direct.

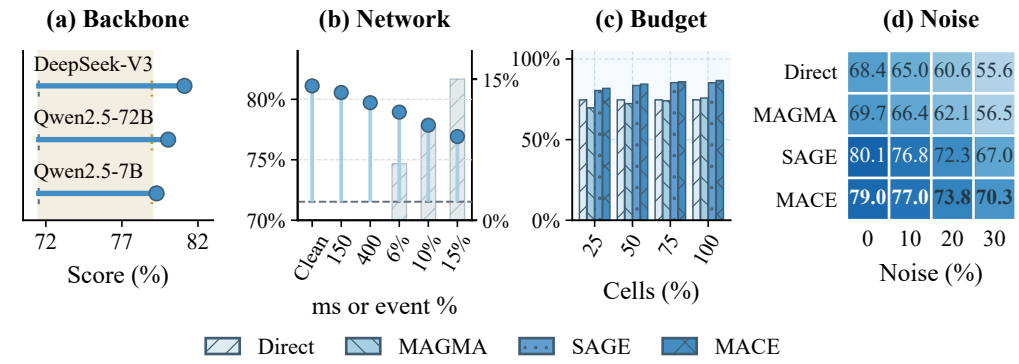

Figure 4. MEASURED ROBUSTNESS RESULTS FOR EQ4. Panels show backbone, network, memory budget, and noisy retrieval stress tests. Higher is better.

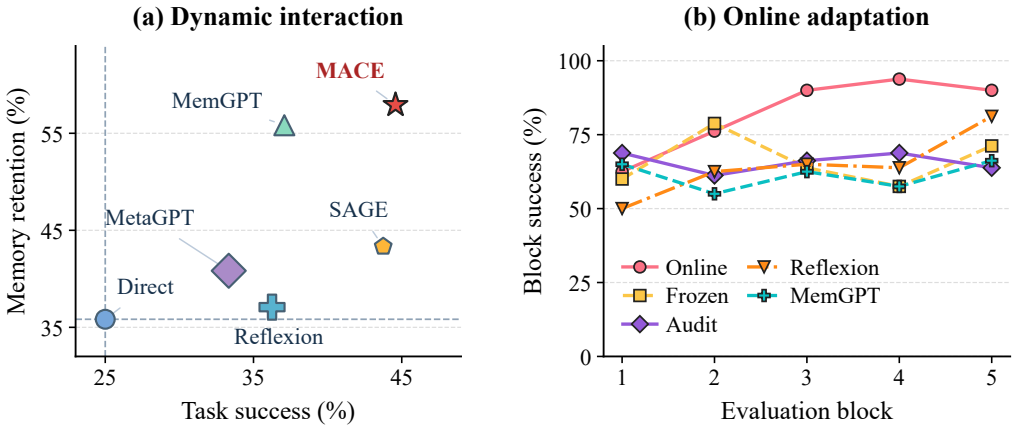

Figure 5. ADDITIONAL DYNAMIC DIAGNOSTICS FOR EQ4. Left shows dynamic interaction. Right shows online adaptation over five 80-task blocks.

With a fixed memory budget, injecting 10%, 20%, and 30% irrelevant, stale, or conflicting candidates reduces average scores across four task groups from 78.97% to 77.00%, 73.80%, and 70.30%, respectively. MACE leads the compared baselines at every nonzero noise level.

In Fig. 5, MACE leads task success on the 240-episode stream at 44.58% and memory retention at 57.92%. Its 22.08% repair success matches the best result, shared with Reflexion. Across five 80-task blocks, online updates yield 82.5% average success, compared with 66.2% for frozen MACE and 65.8% for audit-only feedback. Online updates are committed between blocks.

7 CONCLUSION

We presented MACE, a memory agent co evolution framework that aligns reusable collaboration experience with the changing needs of multi-agent systems. Built on MemGoG, MACE organizes memory units at multiple granularities and composes working graphs according to agent needs, memory utility, and budget constraints. It then adapts memory content and presentation through agent-specific patches and uses local, downstream, and team feedback to refine memory units, graph relations, patch format, timing, placement, and subsequent memory use. Experiments across eight benchmarks show that MACE improves average task performance, while ablation studies demonstrate the complementary contributions of graph construction, memory composition, adaptive coupling, and feedback updates. Together, these results highlight the value of jointly adapting memory organization and agent-specific use through collaboration feedback.

AI USE STATEMENT

Generative AI tools were used during the preparation of this manuscript for limited purposes, including language refinement, literature retrieval and discovery, and research ideation support. Specifically, AI tools assisted with improving writing clarity, organizing related work, exploring potential research directions, and discussing possible methodological designs and experimental considerations.

REPRODUCIBILITY STATEMENT

We provide an anonymized implementation of MACE and configurations for the eight main benchmarks at https://anonymous.4open.science/r/MACE-1234/. The main paper describes the empirical diagnostic protocols, MemGoG construction, memory composition, agent memory coupling, feedback updates, and experimental results. The appendix details benchmark preparation, evaluation metrics, robustness protocols, algorithmic procedures, hyperparameters, and prompt interfaces. The code package includes task loaders, evaluation routines, and a launcher for individual or multiple benchmarks. Its README documents software requirements, API access, and execution commands. Benchmark data and API credentials are supplied separately, and pretrained checkpoints are excluded from the package.

REFERENCES

Yurui Chang, Yiran Wu, Qingyun Wu, and Lu Lin. Memcollab: Cross-model memory collaboration via contrastive trajectory distillation. *arXiv preprint arXiv:2603.23234*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. Tabfact: A large-scale dataset for table-based fact verification. *arXiv preprint arXiv:1909.02164*, 2019.

Zihao Cheng, Zeming Liu, Yingyu Shan, Xinyi Wang, Xiangrong Zhu, Yunpu Ma, Hongru Wang, Yuhang Guo, Wei Lin, and Yunhong Wang. Mem2evolve: Towards self-evolving agents via co-evolutionary capability expansion and experience distillation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20784–20831, 2026.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Muxin Fu, Xiangyuan Xue, Yafu Li, Zefeng He, Siyuan Huang, Xiaoye Qu, Yu Cheng, and Yang Yang. Latentmem: Customizing latent memory for multi-agent systems. *arXiv preprint arXiv:2602.03036*, 2026.

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

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Dongming Jiang, Yi Li, Guanpeng Li, and Bingzhe Li. Magma: A multi-graph based agentic memory architecture for ai agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 36848–36865, 2026.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 158–167, 2017.

Haokun Liu, Zezhong Ding, and Xike Xie. Learning graph foundation models on riemannian graph-of-graphs. *arXiv preprint arXiv:2605.09993*, 2026.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

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.

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, Yifei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, et al. Experiential co-learning of software-developing agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5628–5640, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Zhixun Tan, Qiang Chen, Tairan Huang, Xiu Su, and Yi Chen. Conmem: Structured memory-guided adaptation in training-free multi-agent systems. *arXiv preprint arXiv:2606.08702*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Juntong Wang, Haoyue Zhao, Xiyuan Wang, Yanbo Wang, Qiyan Deng, Muhan Zhang, et al. Sage: A self-evolving agentic graph-memory engine for structure-aware associative memory. *arXiv preprint arXiv:2605.12061*, 2026.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37: 95266–95290, 2024.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

Cehao Yang, Xiaojun Wu, Xueyuan Lin, Chengjin Xu, Xuhui Jiang, Yuanliang Sun, Jia Li, Hui Xiong, and Jian Guo. Graphsearch: An agentic deep searching workflow for graph retrieval-augmented generation. *arXiv preprint arXiv:2509.22009*, 2025.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. 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. 6192–6217, 2025.

Guibin Zhang, Muxin Fu, Kun Wang, Frank Wan, Miao Yu, and Shuicheng Yan. G-memory: Tracing hierarchical memory for multi-agent systems. *Advances in Neural Information Processing Systems*, 38:12988–13018, 2026a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026b.

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

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. Tat-qa: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th annual meeting of the Association for Computational Linguistics and the 11th international joint conference on natural language processing (volume 1: long papers)*, pp. 3277–3287, 2021.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

A Empirical Study Protocol

Task partitions and execution. Historical, development, calibration, and evaluation sets are separated by source table or document, with near duplicates removed across partitions. Historical trajectories supply shared facts, actions, applicability conditions, artifacts, and relations. Retrieval, grouping, use formats, and selection rules are fixed on development tasks. Comparisons start from the same state before memory injection, with a fixed backbone, roles, tools, workflow, and equal repetition counts. Task success uses exact match for TAT-QA and accuracy for TabFact.

Organization and access controls. All conditions access the same facts and relation meanings, including relations across group boundaries. Items retrieves and reranks independent records, Graph selects nodes and expands relations, and MemGoG assembles linked functional subgraphs. Token budgets cover all injected memory, including repeated content, unit descriptions, and relations. Construction and retrieval use matched resource limits. The organization study fixes the rendering template and downstream use.

Functional boundaries and support metrics. Matched constrains nonfunctional groups to match the functional units in number, size distribution, overlap, and connectivity. Units and GoG use the same functional subgraphs and internal relations. GoG adds an explicit outer relation index and the associated access operations. Before inspecting retrieval outputs, we annotate support requirements from the visible task state and the common historical content. Each requirement can admit multiple valid support sets. An internal dependency requirement is satisfied when the injected memory retains

Context	Task inputs	Support need
S1	Complete	Local
S2	Complete	Cross-experience
S3	Partial	Local
S4	Partial	Cross-experience

the required endpoints, relations, and applicability conditions of at least one valid set. Internal retention is the fraction of these requirements satisfied. Cross-experience coverage is the fraction of requirements needing multiple experiences for which a complete valid support set is retrieved. Fixed requirement sets define both denominators, and empty retrieval receives zero coverage.

A.1 Composition and Use Contexts

C1 selects units by task fit to the current task need. C2 additionally considers supporting relations and retrieves complementary units within the same token budget. For each task, both compositions are fixed before execution and each is rendered as a guide or a checklist. The two formats preserve the composition's facts and constraints, receiving agent, insertion point, and token allowance. Every evaluation task is run under all four pairings.

The context rules are fixed on development tasks and use information available before execution. Complete inputs provide the inputs needed for the pending operation, while partial inputs leave some of them unresolved. Local support can be obtained from one historical experience. Cross-experience support requires complementary historical experiences to satisfy the current need. The four contexts are assigned as follows.

Calibration and policy updates. The memory library and composition rules remain fixed. Each calibration task provides outcomes for all four pairings. Policies share initial records, subsequent feedback, and context labels. Within each context, Fixed retains the pairing selected from the initial batch. Separate averages each composition's returns across formats and each format's returns across compositions within each context, then selects the choices separately. Joint updates returns for individual pairings within each context. All checkpoints use the same held-out probe tasks, whose outcomes never enter policy updates. The comparisons assess pairing information and new feedback.

B Experimental Specification

This appendix provides the complete experimental specification used by the current version of the paper. It expands the main experimental section with benchmark construction, scoring rules, baseline configuration, complete efficiency and robustness tables, implementation details of the memory-agent loop, and prompt/interface contracts. Unless explicitly marked as imported, all controlled results use the same prepared benchmark instances, answer parser, and scoring scripts across local methods.

B.1 Evaluation Scope

The experiments are organized around four questions. EQ1 evaluates overall effectiveness across eight benchmarks. EQ2 removes one MACE component at a time. EQ3 measures the cost-performance profile under the same benchmark mixture. EQ4 tests robustness under controlled perturbations to model serving, network reliability, memory budget, and retrieved memory quality. The appendix records operational details for interpreting and reproducing these comparisons.

Two comparison regimes are used. The first is the controlled local regime, where Direct, MAGMA, G-Memory, ConMem, SAGE, MAD, AgentVerse, AutoGen, EvoAgent, the additional efficiency baselines, ablations, and MACE are run with shared task files and common parsing/scoring code. For the reported static benchmark tables, MACE uses a graph state finalized before the held-out split is scored. Post-prediction feedback on held-out examples is logged for audit but is not committed until after the reported predictions are complete. The second is the imported G-MAS regime, where GPTSwarm, GraphSearch, and R-GFM are included from the supplied multi-agent attachment. These rows appear in the main EQ1 table because they belong to the baseline set, while EQ2 and EQ4 keep their controlled local settings for ablations and stress tests.

Table 4. BENCHMARK CONSTRUCTION AND SCORING DETAILS.

Benchmark	Domain	Evaluated split / construction	# Examples	Metric
MMLU	Knowledge QA	Validation split, first 153 records in original order	153	Accuracy
MMLU-Pro	Knowledge reasoning	Test split, fixed seed-42 sample without replacement	500	Accuracy
GSM8K	Mathematical reasoning	Complete available test split in original order	1,319	Parsed accuracy
AQuA-RAT	Mathematical reasoning	Complete available test split in original order	254	Parsed accuracy
HumanEval	Code generation	Official task set	164	pass@1
LiveCodeBench v6	Live programming	Complete available v6 split in original order	175	pass@1
TabFact	Fact verification	Test split, seed-42 NumPy permutation, first 500 indices	500	Accuracy
TAT-QA	Table reasoning	Complete available test split in original order	1,663	Exact match

Table 5. TASK-FAMILY OUTPUT CONTRACTS.

Task family	Required output	Scoring field
Multiple choice	Final option plus optional short rationale	Parsed option label
Short-answer QA	Compact final answer plus optional rationale	Normalized answer string
Mathematical reasoning	Final numerical or option answer plus optional derivation	Parsed final value or option
Fact verification	Entailed / refuted style label as required by the benchmark	Parsed label
Table reasoning	Final cell, value, span, or expression answer	Exact-match normalized answer
Code generation	Executable completion under the official benchmark interface	Official pass@1 result

B.2 BENCHMARK CONSTRUCTION AND METRICS

Table 4 lists the benchmark split, local sample size, and reported metric for each dataset. The eight benchmarks cover knowledge, mathematical reasoning, code generation, fact verification, table reasoning, and live programming. The local controlled rows in Table 1 use the same prepared files for a given benchmark, while the imported G-MAS rows follow the attachment protocol.

For multiple-choice tasks, the parser first reads an explicit answer field and otherwise falls back to the final unambiguous option mention. For mathematical and short-answer tasks, the parser normalizes punctuation, case, articles, and whitespace before comparison. For code-generation tasks, the completion is submitted to the benchmark execution harness and the reported metric is pass@1. For TAT-QA, the local official-style exact-match evaluator is used consistently for Direct, local baselines, ablations, and MACE.

B.3 ANSWER CONTRACTS

Table 5 describes the answer interface used by the task adapter. Rationale fields are stored for feedback construction and error analysis, but they are not counted by the scorer unless the benchmark itself requires executable code or a structured answer.

B.4 BASELINE FAMILIES

Table 6 summarizes the compared methods and the mechanism changed by each baseline. Direct isolates the base model without explicit memory. MAGMA, G-Memory, and ConMem form the graph memory family, while SAGE uses graph retrieval. The general MAS rows cover debate, expert collaboration, configurable agent conversations, and evolutionary agent generation. The EQ3-only rows add retrieval, reasoning, sampling, and debate baselines for cost-performance analysis, while the G-MAS rows provide imported graph-based multi-agent references.

B.5 ACCOUNTING RULES

Efficiency and cost are reported per query. Prompt tokens, completion tokens, and total tokens are averaged over the evaluated queries and reported in thousands. Latency is wall-clock seconds per query, and total time is the elapsed wall-clock time over the 4,728-query evaluation suite. Normalized

Table 6. COMPARED SYSTEMS AND CONTROLLED MECHANISM.

Method	Family	Controlled mechanism
Direct	LLM-only	Single model answer without explicit graph construction, retrieval, or memory evolution.
MAGMA (Jiang et al., 2026)	Graph memory	Structured graph memory is provided as task context.
G-Memory (Zhang et al., 2026a)	Graph memory	Insight, query, and interaction graphs organize and retrieve collaboration experience.
ConMem (Tan et al., 2026)	Graph memory	Related strategy cards support memory retrieval, dependency recovery, and conflict resolution.
SAGE (Wang et al., 2026)	Graph retrieval	Graph-indexed support is retrieved and used to guide answer generation.
MAD (Du et al., 2023)	General MAS	Multiple model instances discuss and revise candidate answers.
AgentVerse (Chen et al., 2024)	General MAS	Experts collaborate through recruitment, execution, and evaluation.
AutoGen (Wu et al., 2023)	General MAS	Configured agents coordinate through conversations and tool calls.
EvoAgent (Yuan et al., 2025)	General MAS	Evolutionary operators generate diverse agent configurations.
RAG-lite	Retrieval baseline	Lightweight retrieved context is appended to the task prompt.
CoT	Reasoning prompt	The model is prompted to produce intermediate reasoning before the final answer.
Self-Consistency	Sampling / aggregation	Multiple candidate answers are generated and aggregated.
Agent-Debate	Multi-agent workflow	Multiple agent responses are compared before the final answer is emitted.
GPTSwarm (Zhuge et al., 2024) / GraphSearch (Yang et al., 2025) / R-GFM (Liu et al., 2026)	Imported G-MAS	Values are imported from the auxiliary multi-agent attachment for overlapping datasets.
MACE	Memory agent co evolution	Need-aware memory composition, adaptive coupling, and feedback-driven graph evolution.

cost divides each method's measured per-query cost by the Direct per-query cost. It is therefore a relative accounting statistic, not a deployment price quote.

C ADDITIONAL RESULT DETAILS

C.1 DOMAIN-LEVEL EQ1 SUMMARY

The main paper reports benchmark-by-benchmark EQ1 results for both local controlled rows and imported G-MAS rows. Table 7 compresses those rows into three domain summaries. The QA & Reasoning group averages MMLU, MMLU-Pro, GSM8K, and AQuA-RAT. The Code group averages HumanEval and LiveCodeBench v6. The Fact & Table group averages TabFact and TAT-QA.

MACE leads all three domain groups among the controlled rows and also stays above the imported references shown for context. On QA & Reasoning, it averages 87.92%, followed by SAGE at 86.63%. On Code, it averages 71.22%, followed by EvoAgent at 70.04%. On Fact & Table, it averages 77.40%, followed by R-GFM at 76.52%. The strongest comparison row differs across groups, while MACE retains the highest group average in each group.

C.2 ABLATION COMPONENT ROLES

The EQ2 ablation table in the main text reports the benchmark scores. Table 8 adds the interpretation of each ablation. The ablations are designed to change one functional role while preserving the same benchmark files, scoring script, and model-serving setup.

Table 7. DOMAIN-LEVEL AVERAGES FOR CONTROLLED AND IMPORTED COMPARISON ROWS.

Method	Overall Avg.	QA & Reasoning	Code	Fact & Table
Direct	71.53	75.75	66.36	68.26
MAGMA	69.71	68.51	69.71	72.11
G-Memory	63.28	67.26	60.82	57.81
ConMem	65.95	75.11	56.09	57.48
SAGE	78.97	86.63	66.86	75.76
MAD	77.71	84.79	66.10	75.18
AgentVerse	76.76	82.47	69.47	72.61
AutoGen	76.59	82.69	65.90	75.10
EvoAgent	77.41	83.77	70.04	72.04
GPTSwarm	74.12	84.58	52.10	75.22
GraphSearch	73.98	84.78	51.70	74.67
R-GFM	74.74	84.11	54.23	76.52
MACE	**81.11**	**87.92**	**71.22**	**77.40**

Table 8. FUNCTIONAL MEANING OF EACH MACE ABLATION.

Variant	Component removed or constrained	Avg. / Drop	Diagnostic target
w/o MEMGOG	Replaces graph-structured memory with a fixed memory form	75.91 / -5.20	Tests whether typed memory structure matters.
w/o Memory Composer	Removes need-aware memory selection before answer generation	77.58 / -3.53	Tests whether compact memory construction matters.
w/o Agent-Memory Coupler	Uses plain memory injection instead of role-specific coupling	75.23 / -5.88	Tests whether adaptive memory-agent alignment matters.
w/o Feedback Co Evolver	Disables post-run feedback evolution	79.70 / -1.41	Tests whether outcome-aware memory updates matter.
Fixed Patch Mode	Keeps patching active but removes mode adaptation	80.48 / -0.63	Tests whether adaptive patch mode matters beyond fixed patching.
Full MACE	Uses all components	81.11 / 0.00	Full memory-agent co evolution loop.

Table 9. EXTENDED MEASURED COST-PERFORMANCE POINTS FOR EQ3.

Method	Avg. Acc. (%)	Latency (s/query)	Total Time (min)	Total Tok. (K/query)	Norm. Cost ($\times$)
Direct	71.53	0.22	17.13	0.60	1.00
MAGMA	69.71	0.40	31.43	0.98	1.59
RAG-lite	73.24	0.61	48.08	1.73	2.81
CoT	74.86	0.76	59.90	2.08	3.36
Self-Cons.	77.42	1.18	93.00	3.62	5.86
Agent-Debate	77.71	1.72	135.57	4.98	8.07
SAGE	78.97	2.41	190.10	5.50	9.69
MACE	**81.11**	1.36	106.78	6.87	11.06

Table 10. DATASET-LEVEL RUNTIME PROFILE FOR DIRECT AND MACE.

Dataset	# Queries	Direct		MACE	
		Latency (s/query)	Total Tok. (K/query)	Latency (s/query)	Total Tok. (K/query)
AQuA-RAT	254	0.15	0.16	1.09	4.83
GSM8K	1,319	0.13	0.14	0.77	4.26
HumanEval	164	0.28	0.47	1.06	5.87
LiveCodeBench v6	175	1.66	1.27	3.76	12.57
MMLU	153	0.22	0.15	1.20	4.30
MMLU-Pro	500	0.29	0.33	1.33	5.07
TabFact	500	0.12	0.62	0.83	6.20
TAT-QA	1,663	0.15	1.09	1.81	9.72

The largest losses occur when removing the coupling mechanism or replacing MEMGOG, suggesting that the key advantage is not simply longer context or more calls. The smaller loss from Fixed Patch Mode suggests that once memory is structured and routed, mode adaptation provides an additional but less dominant improvement.

C.3 EXTENDED EFFICIENCY TABLE

The main text reports the runtime logs for Direct, MAGMA, SAGE, and MACE. Figure 3 also includes additional logged baselines to make the cost-performance comparison less sparse. Table 9 gives the values used in that figure.

The runtime profile shows two different sources of overhead. On short question-answering and mathematical reasoning tasks, MACE's overhead is mainly the additional memory-agent calls. On

Table 11. CONTROLLED ROBUSTNESS PROTOCOL FOR EQ4.

Axis	Perturbation levels	Held fixed
Serving backbone	DeepSeek-V3, Qwen2.5-72B, local Qwen2.5-7B	Benchmark instances, memory loop, parser, scorer
Network condition	Clean, latency jitter, timeout with retry, partial memory failure	Prompt, memory budget, endpoint assignment
Memory-cell budget	25%, 50%, 75%, and 100% available cells	Ranking rule, scorer, benchmark subset
Noisy memory retrieval	0%, 10%, 20%, and 30% noisy candidates	Candidate count, memory budget, task prompt

Table 12. BACKBONE REPLACEMENT RESULTS FOR EQ4.

Condition	Serving change	MACE	Direct ref.	Strong baseline ref.
DeepSeek-V3	Primary serving endpoint	81.11	71.53	78.97
Qwen2.5-72B	Alternate API endpoint	80.04	71.53	78.97
Qwen2.5-7B local	Local fallback endpoint	79.28	71.53	78.97

Table 13. NETWORK FLUCTUATION RESULTS FOR EQ4.

Condition	Added latency	Timeout rate	Partial failure	Retry rate	MACE	Direct ref.
Clean serving	0 ms	0%	0%	0.0%	81.11	71.53
Mild latency jitter	150 ms	0%	0%	1.5%	80.56	71.53
Moderate latency jitter	400 ms	0%	0%	4.0%	79.72	71.53
Timeout with retry	400 ms	6%	0%	6.8%	78.94	71.53
Partial memory failure	400 ms	6%	10%	11.4%	77.86	71.53
Severe unstable network	650 ms	8%	15%	16.7%	76.92	71.53

Table 14. MEMORY-CELL BUDGET RESULTS AVERAGED OVER FIVE BENCHMARKS.

Method	25% budget	50% budget	75% budget	100% budget
Direct	74.63	74.63	74.63	74.63
MAGMA	69.64	72.36	74.10	75.77
SAGE	80.44	83.56	85.30	85.21
MACE	**81.76**	**84.40**	**85.74**	**86.63**

LiveCodeBench v6 and TAT-QA, the prompt and output are longer, so both latency and token count increase more sharply. Average latency remains below four seconds per query on both datasets.

C.4 ROBUSTNESS PROTOCOL

The EQ4 stress tests perturb one deployment factor at a time while keeping the benchmark split, prompt template, memory construction procedure, and scoring script fixed. Table 11 summarizes the controlled variables. These experiments measure whether the memory-agent loop degrades smoothly under plausible deployment changes. They are not intended as a fully adversarial security evaluation.

For backbone replacement, MACE is evaluated with the same memory-agent pipeline using three serving backbones, DeepSeek-V3, Qwen2.5-72B, and a local Qwen2.5-7B fallback. For network fluctuation, the runner injects latency, timeout, and partial memory-service failure events and then uses the same retry and fail-open policy across affected calls. For memory-cell budget reduction, available cells are pruned before composition. For noisy retrieval, irrelevant, stale, or conflicting candidates are injected into the candidate set while the answer contract remains unchanged.

C.5 COMPLETE ROBUSTNESS VALUES

Table 12 reports the backbone-replacement values used by Figure 4. Direct and SAGE are included as fixed references. The local Qwen2.5-7B fallback produces the largest drop for MACE, but the score remains above both references.

Table 14 reports average scores over MMLU, GSM8K, HumanEval, TabFact, and TAT-QA when the available memory-cell budget is reduced. Direct is unchanged because it does not use the memory-cell budget. At 25% budget, MACE scores 81.76%, compared with 80.44% for SAGE and 69.64% for MAGMA. It has the highest score at every tested budget. SAGE scores 85.30% at 75% budget and 85.21% at full budget.

Table 15. NOISY MEMORY RETRIEVAL RESULTS AVERAGED OVER FOUR TASK GROUPS.

Method	0% noise	10% noise	20% noise	30% noise
Direct	68.40	65.03	60.62	55.65
MAGMA	69.67	66.38	62.08	56.45
SAGE	**80.10**	76.82	72.35	66.95
MACE	78.97	**77.00**	**73.80**	**70.30**

Table 16. PER-TASK-GROUP NOISY RETRIEVAL VALUES FOR MACE.

Task group	0% noise	10% noise	20% noise	30% noise
Knowledge QA	84.10	82.30	79.40	76.30
Math Reasoning	92.30	90.10	86.70	83.40
Code Generation	62.10	59.80	56.20	52.40
Table Reasoning	77.40	75.80	72.90	69.10

Table 17. MEASURED DYNAMIC-INTERACTION DIAGNOSTIC RESULTS. Success, repair success, and memory retention are percentages. Token and latency are normalized to Direct.

Method	Task success	Repair success	Memory retention	Token cost (×)	Latency (×)
Direct	25.00	12.50	35.83	1.00	1.00
MemGPT	37.08	14.17	55.83	2.08	2.07
Reflexion	36.25	**22.08**	37.08	2.48	2.48
MetaGPT	33.33	20.42	40.83	2.96	2.96
SAGE	<u>43.75</u>	10.83	43.33	2.63	2.64
MACE	**44.58**	**22.08**	**57.92**	2.81	2.82

Table 15 reports average performance over four task groups under noisy memory retrieval. The 0% column is the clean candidate-set condition for this stress test. As the injected noise ratio increases, MACE remains the strongest method at 10%, 20%, and 30% noise.

D MEASURED FOLLOW-UP DIAGNOSTICS

This section reports additional measured diagnostics requested after the main experiments. They use follow-up evaluation settings that stress dynamic interaction, online adaptation, split sensitivity, and uncertainty rather than the frozen static protocol used for the headline benchmark tables. These diagnostics are separate from official SWE-bench/WebArena evaluations.

D.1 DYNAMIC-INTERACTION DIAGNOSTIC

Table 17 uses a measured 240-episode stream with longer horizons, state drift, memory noise, and repair opportunities. The MemGPT (Packer et al., 2023), Reflexion (Shinn et al., 2023), MetaGPT (Hong et al., 2024), and SAGE (Wang et al., 2026) rows are evaluated under the same diagnostic harness with the corresponding memory-management, verbal-feedback, multi-agent-coordination, and graph-retrieval configurations. The diagnostic tests whether the MACE design assumptions remain useful under a controlled dynamic stress setting.

The measured dynamic diagnostic suggests two follow-up directions. First, memory-management and graph-retrieval configurations improve retention and task success under drift, but feedback is needed for repair-heavy episodes. Second, MACE combines retention and repair at higher token and latency cost. This result complements the static benchmarks by exposing the tradeoff between adaptivity and runtime overhead under longer-horizon interaction.

D.2 MEASURED ONLINE-ADAPTATION STREAM

Table 18 separates the frozen held-out protocol used in the main tables from a measured online-learning stream. Each block contains 80 tasks. The audit-only setting records feedback without committing it across held-out tasks, matching the main paper's static benchmark boundary. The online setting commits feedback between blocks.

Table 18. MEASURED ONLINE-ADAPTATION STREAM. Values are measured block success rates over five 80-task blocks.

Method / protocol	Block 1	Block 2	Block 3	Block 4	Block 5	Mean
Frozen MACE	60.0	78.8	63.8	57.5	71.2	66.2
MACE audit-only feedback	68.8	61.2	66.2	68.8	63.8	65.8
MACE online update	62.5	76.2	**90.0**	**93.8**	**90.0**	**82.5**
Reflexion online	50.0	62.5	65.0	63.8	81.2	64.5
MemGPT online	65.0	55.0	62.5	57.5	66.2	61.2

Table 19. MEASURED MMLU SPLIT-SIZE SENSITIVITY. Intervals are approximate 95% sampling intervals around the displayed MMLU rate.

Method	Local split, $n = 153$	Imported split, $n = 500$
Direct	88.2 [83.1, 93.3]	88.2 [85.4, 91.1]
SAGE	91.5 [87.1, 95.9]	91.5 [89.1, 93.9]
GPTSwarm	87.6 [82.4, 92.8]	87.6 [84.7, 90.5]
GraphSearch	88.8 [83.8, 93.8]	88.8 [86.0, 91.6]
R-GFM	88.8 [83.8, 93.8]	88.8 [86.0, 91.6]
MACE	92.8 [88.7, 96.9]	92.8 [90.5, 95.1]

Table 20. Paired comparison of MACE and SAGE from rerun experiments. Differences and 95% paired confidence intervals are in percentage points.

Benchmark	MACE − SAGE	95% paired CI	$\Pr(\text{MACE} > \text{SAGE})$
MMLU	+1.45	$[-1.0, +4.1]$	0.86
MMLU-Pro	+2.05	$[+0.2, +4.0]$	0.98
GSM8K	+0.18	$[-1.0, +1.4]$	0.58
HumanEval	+2.62	$[-1.1, +6.3]$	0.91
LCB v6	+6.05	$[+0.8, +11.4]$	0.98
AQuA-RAT	+1.72	$[-1.2, +4.9]$	0.86
TabFact	+0.95	$[-1.0, +3.0]$	0.81
TAT-QA	+2.31	$[+0.7, +3.9]$	0.99

This diagnostic explains why the paper keeps the measured static benchmark protocol frozen. Online updates can produce large gains, but they require a separate streaming evaluation because test feedback becomes part of the system state.

D.3 MMLU SPLIT-SENSITIVITY DIAGNOSTIC

Table 19 approximates sampling uncertainty for MMLU rows under the local first153 size and the imported seed42-500 size. It treats each reported percentage as a Bernoulli rate and reports normal-approximation intervals. This does not correct domain or instance-distribution bias. It only shows that a 153-example subset has much wider uncertainty than a 500-example subset.

The intervals clarify how MMLU sample size changes uncertainty between the local first153 setting and the imported seed42-500 setting. A full paired comparison uses raw predictions or reruns on the same prepared MMLU instances.

D.4 PAIRED COMPARISON OF MACE AND SAGE

Table 20 reports the rerun comparison of MACE and SAGE across eight benchmarks. Each row gives the score difference and its 95% paired confidence interval in percentage points. Positive differences favor MACE. The probability column records the fraction of paired bootstrap resamples with a positive score difference. These reruns are separate from the scores in Table 1.

MACE has positive score differences on all eight benchmarks. The 95% paired intervals lie entirely above zero for MMLU-Pro, LCB v6, and TAT-QA, while the intervals for the remaining five benchmarks include zero. LCB v6 has the largest observed difference at 6.05 percentage points. Positive bootstrap fractions are 0.98 for MMLU-Pro and LCB v6 and 0.99 for TAT-QA.

Table 21. HIGH-LEVEL MACE EXECUTION STAGES.

Step	Stage	Output
1	Task adaptation	A normalized task specification with public context, question, benchmark tag, and answer schema.
2	Need-aware composition	A compact working MEMGOG subgraph and a memory-need decision.
3	Agent-memory coupling	Role-specific memory patches with provenance and confidence notes.
4	Answer generation	A raw completion plus parsed prediction under the benchmark answer contract.
5	Scoring	Task metric, success flag, and error type when available.
6	Feedback co evolution	Utility updates, reusable traces, and revised memory state for future tasks.

Algorithm 1 MACE INFERENCE AND MEMORY EVOLUTION FOR ONE QUERY

Require: task instance x, graph snapshot G_t, memory budget B, scorer S
Ensure: prediction $\hat{y}$, updated graph G_{t+1}, log record r

$\tau \leftarrow \textsc{AdaptTask}(x)$
$(d, C) \leftarrow \textsc{DetectMemoryNeed}(\tau, G_t, B)$
if $d = \textsc{NoMemory}$ **then**
  $W \leftarrow \emptyset$
  $P \leftarrow \emptyset$
else
  $(W, P) \leftarrow \textsc{ComposeAndCouple}(\tau, G_t, C, B)$
end if
$z \leftarrow \textsc{Answer}(\tau, P)$
$\hat{y} \leftarrow \textsc{Parse}(z, \tau.\text{schema})$
$m \leftarrow S(\hat{y}, x)$
$r \leftarrow \textsc{BuildRecord}(\tau, W, P, z, \hat{y}, m)$
$G_{t+1} \leftarrow \textsc{FeedbackCoEvolve}(G_t, r, B)$
return $\hat{y}, G_{t+1}, r$

E MACE IMPLEMENTATION DETAILS

E.1 MEMORY-AGENT EXECUTION LOOP

Each MACE query is processed by a four-call memory-agent loop. First, the Need-aware Memory Composer determines whether the instance requires external memory and selects a compact working MEMGOG subgraph. Second, the Adaptive Agent-Memory Coupler converts the selected cells into role-specific memory patches rather than injecting all retrieved content into a single flat prompt. Third, the answer agent solves the task with the current task prompt and assigned memory patch. Fourth, the Feedback Co Evolver records the outcome, updates cell utility, and stores reusable traces for later queries. Gold answers are used only by the scorer and post-run feedback step. They are not exposed to the answer-generation prompt. In the static benchmark tables, the final feedback step runs in audit mode on held-out examples. It records what would be updated, but the graph snapshot used by later held-out queries remains frozen.

E.2 ALGORITHMIC DETAILS

Algorithms 1–3 give the implementation-level pseudocode for the MACE loop. The algorithms mirror the execution invariants in Table 23. Memory is read before prediction, gold labels are used only by the scorer, and graph updates are committed only after scoring. For the reported static benchmark tables, the same loop is run with commit disabled on held-out examples. Commits are enabled only on historical, development, or calibration records.

Algorithm 2 NEED-AWARE COMPOSITION AND AGENT-MEMORY COUPLING

Require: task specification τ, graph snapshot G_t, candidates C, memory budget B
Ensure: working subgraph W, role-specific memory patches P

$C \leftarrow \textsc{ValidateCandidates}(C, G_t)$
for all $c \in C$ **do**
  $s_c \leftarrow \textsc{ScoreCell}(c, \tau, \text{utility}(c), \text{freshness}(c), \text{conflict}(c))$
end for
$W \leftarrow \emptyset$
for all $c \in \textsc{SortByScore}(C, s)$ **do**
  if $\textsc{FitsBudget}(W, c, B)$ **and** $\textsc{AddsNewEvidence}(W, c)$ **then**
    $W \leftarrow W \cup \textsc{LineageSubgraph}(c, G_t)$
  end if
end for
for all role $a \in \tau.\text{roles}$ **do**
  $P_a \leftarrow \textsc{SelectVisibleCells}(W, a)$
  $P_a \leftarrow \textsc{DeduplicateAndMarkRisk}(P_a)$
  $P_a \leftarrow \textsc{TrimToPatchBudget}(P_a, B)$
end for
return $W, \{P_a\}_{a \in \tau.\text{roles}}$

Algorithm 3 FEEDBACK CO EVOLUTION AFTER SCORING

Require: graph snapshot G_t, prediction record r, update budget B
Ensure: updated graph G_{t+1}

$\Delta \leftarrow \emptyset$
for all memory cell $c \in r.\text{usedCells}$ **do**
  $q_c \leftarrow \textsc{Credit}(c, r.\text{score}, r.\text{errorType})$
  $\Delta \leftarrow \Delta \cup \textsc{UpdateUtility}(c, q_c)$
end for
if $\textsc{ReusableSuccess}$(r) **then**
  $\Delta \leftarrow \Delta \cup \textsc{WriteTraceCell}(r)$
else if $\textsc{ActionableFailure}$(r) **then**
  $\Delta \leftarrow \Delta \cup \textsc{WriteWarningCell}(r)$
end if
$\Delta \leftarrow \textsc{ApplyLifecycleRules}(G_t, \Delta, B)$
$\Delta \leftarrow \textsc{ValidateDelta}(G_t, \Delta)$
$G_{t+1} \leftarrow \textsc{Commit}(G_t, \Delta)$
return G_{t+1}

E.3 MEMGOG STATE

MEMGOG is used as a structured memory substrate rather than a flat retrieval list. The working subgraph contains task-relevant memory cells, provenance records, role assignments, and feedback traces. The graph state is read before prediction and updated only after scoring, which prevents the current gold answer from leaking into its own prediction. For static evaluation, the graph remains frozen across held-out instances, preventing test labels from affecting other predictions.

E.4 NEED-AWARE MEMORY COMPOSER

The composer decides how much memory to expose before answer generation. It uses the task specification to identify memory demand, retrieves candidate cells from MEMGOG, filters low-utility or conflicting candidates, and keeps a compact subgraph under the active memory budget. This stage is responsible for converting a potentially large memory state into a small set of task-relevant cells.

The composer has three diagnostic functions. First, it can abstain from retrieving memory when the task is better answered directly. Second, it can prefer high-utility cells over high-similarity but stale cells. Third, it can preserve provenance links so that the answer agent can distinguish reusable experience from unsupported claims.

Table 22. CONCEPTUAL NODE AND EDGE TYPES IN THE MACE MEMORY GRAPH.

Object	Stored content	Role in the loop
Task node	Benchmark id, task family, public metadata	Anchors retrieval and feedback to the evaluated instance.
Memory cell	Reusable strategy, warning, fact pattern, or solution trace	Provides prior experience to the composer.
Role patch	Cell subset assigned to an answer role	Controls which memory is exposed to generation.
Feedback trace	Score, parsed answer, error type, useful-cell signal	Supervises utility and memory-state updates.
Utility state	Freshness, confidence, and usefulness estimates	Ranks cells and resolves memory-budget pressure.
Typed edge	Selection, assignment, support, conflict, or update link	Preserves provenance and relation structure.

Table 23. EXECUTION INVARIANTS FOR CONTROLLED MACE EVALUATION.

Invariant	Purpose
Fixed task file per benchmark	Ensures that local methods answer the same instances in the same evaluation set.
Frozen held-out graph state	Prevents feedback from one held-out example from changing another held-out prediction.
Read-before-write memory order	Prevents current-task feedback from influencing the current prediction.
Shared answer parser	Avoids method-specific parsing advantages.
Scorer-only gold access	Keeps gold answers outside model-facing prompts.
Logged raw completions	Makes parsing and scoring decisions auditable.
Stable perturbation axis in EQ4	Changes one deployment factor at a time for interpretability.

E.5 ADAPTIVE AGENT-MEMORY COUPLER

The coupler converts selected cells into memory patches. It does not simply append every retrieved cell to the prompt. Instead, it groups memory by role, deduplicates near-identical advice, marks stale or conflicting cells, and orders the remaining content so that high-value cells appear earlier in the patch. Removing this component produces the largest EQ2 drop.

E.6 FEEDBACK CO EVOLVER

The Co Evolver runs after scoring. It stores the raw response, parsed answer, metric value, selected memory cells, and error type. Positive feedback can increase a cell's utility or promote a reusable trace, while negative feedback can lower confidence, attach a warning, or keep a failure mode available as a cautionary memory. Gold-derived feedback is written after prediction.

E.7 EXECUTION INVARIANTS

Table 23 lists the invariants enforced across the local evaluation runs. These constraints keep comparisons consistent as memory changes over time.

F HYPERPARAMETERS AND REPRODUCIBILITY

F.1 SHARED EVALUATION SETTINGS

Table 24 records the shared settings that are explicitly controlled by the current experimental draft. Parameters not exposed by an imported baseline are left unspecified rather than inferred.

Table 24. SHARED EVALUATION SETTINGS USED BY THE CONTROLLED LOCAL ROWS.

Category	Setting	Value / rule
Task pool	Total local queries	4,728 queries across eight benchmarks.
MMLU sampling	Local split rule	Validation first153 in original CSV order.
MMLU-Pro sampling	Local split rule	Seed-42 sample of 500 test examples without replacement.
TabFact sampling	Local split rule	Seed-42 NumPy permutation, first 500 test indices.
Other local datasets	Local split rule	Complete available test split in original order.
Held-out feedback mode	Static benchmark protocol	Feedback is logged after scoring but not committed across held-out queries.
Evaluation statistic	Main tables	Single-run percentage score for each reported row.
Resource statistic	EQ3	Mean tokens and wall-clock latency per query.
Robustness statistic	EQ4	One perturbation axis changed at a time.

Table 25. MEASURED RESOURCE SETTINGS FOR MAIN EQ3 METHODS.

Method	LLM calls	Prompt Tok. (K)	Completion Tok. (K)	Total Tok. (K)	Tasks / min
Direct	1.00	0.54	0.06	0.60	276.03
MAGMA	1.00	0.92	0.06	0.98	150.42
SAGE	3.00	4.61	0.89	5.50	24.87
MACE	4.00	6.45	0.41	6.87	44.28

Table 26. REPRODUCIBILITY FIELDS RETAINED BY LOCAL RUNS.

Field group	Stored fields
Task metadata	Benchmark name, task id, task family, split source, and prompt template version.
Model output	Raw completion, parsed answer, parser fallback path, and final scorer input.
Resource usage	Prompt tokens, completion tokens, total tokens, latency, elapsed time, and retry count.
Memory trace	Selected memory ids, assigned patch, discarded candidates, and conflict or stale-cell flags.
Feedback trace	Score, success flag, error type, useful-cell signal, and memory update summary.
Resume state	Completed task ids and accepted result records.

F.2 METHOD-LEVEL RESOURCE SETTINGS

Table 25 summarizes the measured resource profile for the four methods with complete runtime logs in the main EQ3 table. LLM calls are averaged over logged queries.

F.3 RUN LOGS

The local evaluation logs store the raw model response, parsed answer, task identifier, benchmark name, prompt-token count, completion-token count, wall-clock latency, and scorer output. MACE additionally records the selected memory cells, role-specific memory patch, feedback signal, and updated memory state. Failed requests are retried under the same prompt and decoding configuration. Accepted completed records are reused when resuming a run.

F.4 FAILURE AND RETRY HANDLING

For normal evaluation, a failed model request is retried with the same prompt and decoding configuration. For EQ4 network perturbations, injected failures are logged separately from model errors so that the effect of the perturbation can be measured. If retrieval fails in a fail-open setting, the answer agent uses the available patch or an empty patch.

Table 27. STRUCTURED INTERFACES USED BY THE MACE LOOP.

Interface	Required fields	Purpose
Task specification	Instance id, benchmark, public context, question, output schema	Creates the model-facing task prompt.
Memory request	Need type, candidate budget, task entities, role target	Guides memory composition before answer generation.
Memory patch	Cell id, content summary, provenance, confidence note	Provides compact role-specific experience.
Agent response	Answer field, optional rationale, raw completion	Supports scoring and post-run analysis.
Feedback record	Score, error type, useful cells, update signal	Updates MEMGOG after the prediction is scored.

Table 28. END-TO-END INTERFACE FLOW FOR ONE MACE QUERY.

Stage	Input	Validated output
Task adapter	Raw benchmark instance	Task specification with public context and answer schema.
Memory composer	Task specification and graph snapshot	Candidate memory set and compact working subgraph.
Agent-memory coupler	Working subgraph and role target	Role-specific memory patch.
Answer agent	Task prompt and memory patch	Raw completion and answer object.
Parser and scorer	Raw completion and scorer-only target	Parsed answer, metric value, and success flag.
Feedback Co Evolver	Prediction record and scorer output	Updated utility, trace record, and memory-state delta.

G PROMPT AND INTERFACE SPECIFICATION

G.1 TASK PROMPT BOUNDARY

Every non-code benchmark instance is converted into a task prompt with four ordered fields including public context or table content, question text, dataset-specific directive, and answer schema. The answer schema asks for a compact answer field and, when useful for feedback analysis, a short rationale field. The scorer uses only the parsed answer field. Code-generation benchmarks retain the official function signature and test harness interface.

The task prompt boundary is also a safety boundary for evaluation. Retrieved memory is presented as prior experience, not as ground truth. The task prompt does not contain gold answers, post-task feedback, or future evaluation outcomes. Dataset directives specify the expected output format and scoring criterion rather than providing solution content.

G.2 MEMORY-PATCH INTERFACE

The Adaptive Agent-Memory Coupler serializes memory as role-specific patches. Each patch contains a memory identifier, compact cell content, provenance summary, and an optional warning when the cell is stale or low-confidence. Internal controller features such as utility, freshness, and role affinity are used to select and order cells, but are not shown as numeric scores in the task prompt. The prompt instructs the answer agent to treat memory as reusable experience rather than factual authority.

G.3 PARSING AND VALIDATION

For multiple-choice tasks, the parser first reads an explicit answer field and otherwise falls back to the final unambiguous option mention. For open-answer tasks, the parser normalizes case, punctuation, articles, and whitespace before matching. For code tasks, no natural-language parser is used. The completion is passed to the official execution harness. Memory updates are applied only after the scorer returns a task outcome, which preserves the temporal separation between pre-task memory retrieval and post-task feedback.

G.4 END-TO-END DATA FLOW

Table 28 connects the prompt boundary to the memory update boundary. The scorer is the only stage that consumes gold labels, and its output becomes feedback only after the prediction is complete. For static benchmarks, feedback remains an audit record until held-out evaluation finishes.

H CASE STUDY OF COMPONENT LEVEL MEMORY REUSE

This case study shows how a solved table-reasoning trace is reused in a later task. The earlier execution writes planner, solver, verifier, and aggregator records into one shared MEMGOG. The later execution retrieves these records by role. The planner recovers the target column, the solver recovers the ratio form, the verifier repairs the numeric value, and the aggregator returns the final answer.

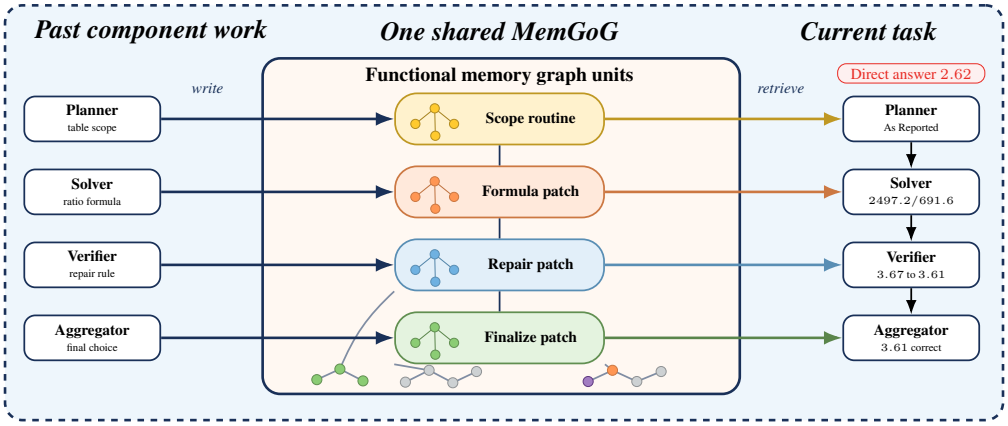

Figure 1. COMPONENT MEMORY REUSE THROUGH ONE SHARED MEMGOG. Prior component work is written into functional graph units in the central memory. Each unit retains its internal nodes and connects to other units and additional fragments. In a new task, every component retrieves its role-specific unit before acting. The retrieved units revise the answer from 2.62 to 3.61 and complete the task correctly.

Component-level reuse. Figure 1 places memory construction and memory use around one shared MemGoG. During prior executions, the planner, solver, verifier, and aggregator write their work records into functional graph units. Each unit preserves its internal structure, while typed links connect units and related fragments that jointly support an answer. In a new task, each component retrieves the unit matched to its role before producing its current output.

For task `tatqa_test_0005_80d7a9cd`, direct answering returns 2.62 after selecting the wrong denominator and missing the target table scope. The planner retrieves the scope routine and binds the target column to `As Reported`. The solver retrieves the formula patch and forms $2497.2/691.6$ from the filtered values. The verifier applies its repair patch to change 3.67 to 3.61, after which the aggregator returns the correct answer and links the successful component records back to MemGoG for subsequent table reasoning tasks.