

LEGOMem: Modular Procedural Memory for Multi-agent LLM Systems for Workflow Automation

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang,
Victor Rühle, Saravan Rajmohan
Microsoft

ABSTRACT

We introduce LEGOMem, a modular procedural memory framework for multi-agent large language model (LLM) systems in workflow automation. LEGOMem decomposes past task trajectories into reusable memory units and flexibly allocates them across orchestrators and task agents to support planning and execution. To explore the design space of memory in multi-agent systems, we use LEGOMem as a lens and conduct a systematic study of procedural memory in multi-agent systems, examining where memory should be placed, how it should be retrieved, and which agents benefit most. Experiments on the OfficeBench benchmark show that orchestrator memory is critical for effective task decomposition and delegation, while fine-grained agent memory improves execution accuracy. We find that even teams composed of smaller language models can benefit substantially from procedural memory, narrowing the performance gap with stronger agents by leveraging prior execution traces for more accurate planning and tool use. These results position LEGOMem as both a practical framework for memory-augmented agent systems and a research tool for studying memory design in multi-agent workflow automation.

CCS CONCEPTS

• **Computing methodologies** → **Multi-agent systems**; *Reasoning about beliefs and knowledge*.

KEYWORDS

Multi-agent systems, Procedural memory, LLM Agents, Workflow

ACM Reference Format:

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, Saravan Rajmohan. 2026. LEGOMem: Modular Procedural Memory for Multi-agent LLM Systems for Workflow Automation. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026)*, Paphos, Cyprus, May 25 – 29, 2026, IFAAMAS, 10 pages.

1 INTRODUCTION

Large Language Models (LLMs) are increasingly deployed as agents to automate complex multi-step workflows [2, 3, 5, 13, 19, 20, 22, 25–28, 33, 35, 39]. These agents are especially valuable in productivity environments such as document editing, email handling, and calendar scheduling. To manage the diversity and compositionality of such tasks, recent systems often adopt multi-agent [23, 30] designs,

where multiple LLM-based agents collaborate, specialize, or delegate responsibilities across roles and tools [4, 7, 9, 32, 36]. This trend reflects a broader shift in AI system design: the real world is inherently multi-agent, involving heterogeneous roles and coordinated decision-making. Multi-agent LLM systems offer a scalable and modular approach to reasoning, tool-use, and workflow execution, positioning them as a natural fit for these increasingly complex productivity environments.

Despite these advances, current multi-agent systems remain largely stateless and transactional: each task is solved from scratch, without reusing prior experience. This lack of memory—particularly procedural memory—limits their ability to learn from past experiences and build up execution skills over time for complex workflows. While recent works have proposed memory modules for single-agent LLMs, such as Synapse [37], Agent Workflow Memory (AWM) [29], these approaches do not address the unique coordination and specialization challenges of multi-agent systems.

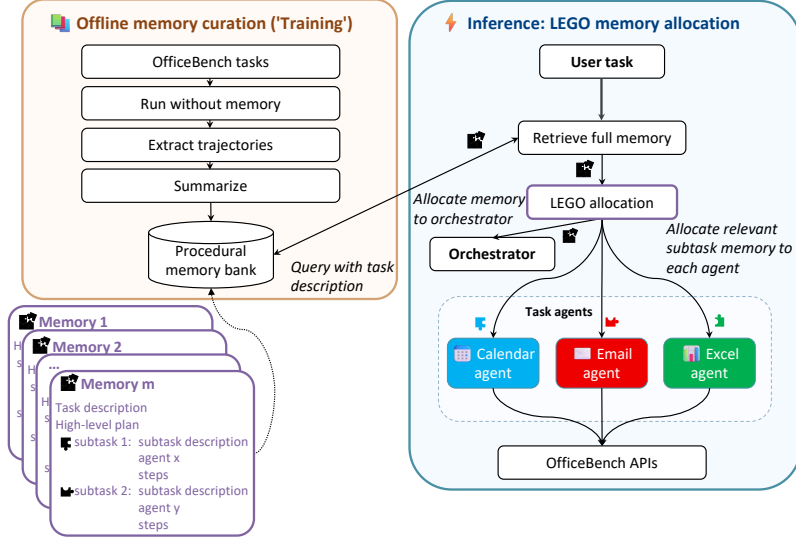
To address this gap, we introduce LEGOMem, a modular procedural memory framework designed for multi-agent LLM systems. In this work, we focus on a common and practical subclass of multi-agent architectures, where a central orchestrator performs planning and delegates subtasks to specialized tool-using task agents, as exemplified by the Magentic-One framework [9, 32]. Our goal is to equip both orchestrators and task agents with memory grounded in prior task trajectories, enabling them to perform better planning, coordination, and task executions. To this end, we design LEGOMem to distill successful executions into structured memory units: full-task memories (task-level plans and reasoning traces) and subtask memories (agent behavior and tool interactions). These modular memories are stored in a memory bank, indexed by semantic embeddings, and reused at inference time to augment planning and execution.

LEGOMem is instantiated as a retrieval augmentation (RAG) [8, 10, 15] layer over existing multi-agent systems. During a new task, the orchestrator receives relevant full-task memories to support task decomposition and agent selection, while each task agent is assigned subtask memories aligned with its delegated subtasks. We explore three memory retrieval strategies—*vanilla*, *dynamic retrieval*, and *query rewriting*—to study how retrieval and memory specificity affect multi-agent performance. This framework allows us to systematically investigate key questions in multi-agent memory design, including where memory should be placed, how it should be retrieved, and which agents benefit most from it.

We evaluate LEGOMem in the context of productivity workflow automation using the OfficeBench [28] benchmark, with agent teams composed of LLM-only, hybrid, and small language model

correspond to Dongge Han at donggehan@microsoft.com.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). This work is licenced under the Creative Commons Attribution 4.0 International (CC-BY 4.0) licence.



(a) Overview of the LEGOMem framework



(b) Example LEGOMem memory structure

Figure 1: LEGOMem framework overview and example memory. The multi-agent system consists of an orchestrator and task agents. The orchestrator performs planning, next agent selection, and subtask allocation, while task agents execute subtasks by interacting with the environment via API tool calls. (Note: For clarity, additional task agents such as Word agent are omitted.)

configurations. Across these settings, all LEGOMem variants significantly improve task success rates over memory-less and baseline methods. Our ablation studies reveal that orchestrator memory is critical for high-level planning and delegation, while fine-grained subtask retrieval provides meaningful gains for smaller agents that rely more on localized execution support. These findings highlight how memory placement and retrieval strategy shape the effectiveness of multi-agent collaboration in workflow settings. Overall, LEGOMem provides a practical and extensible framework for memory-augmented multi-agent workflow automation, enabling agents to plan, coordinate, and execute more effectively by reusing structured procedural knowledge. We hope this work facilitates further research on memory design, continual learning, and efficient agent collaboration in complex productivity settings.

2 RELATED WORK

Multi-agent LLM systems for workflow automation. The recent advent of LLMs has enabled the development of multi-agent systems able to plan, decompose and solve complex workflows. Generalist multi-agent frameworks [4, 7, 9, 32, 36] such as Magentic-One [9] use a common design pattern where a lead orchestrator agent decomposes high-level goals into a step-by-step plan and directs a team of specialized agents to execute specific subtasks. This modular, multi-agent architecture simplifies development and facilitates the reuse of encapsulated skills, a significant advantage over monolithic, single-agent approaches. However, a key limitation remains that they are often stateless, solving each task from scratch and discarding valuable insights gained during execution. Without

memory, agents may repeatedly make the same errors and cannot improve over time.

Memory for LLM agents. Memory offers a natural solution to the limitations of stateless agents. However, a primary challenge is that memory in LLM agents is often designed for single-agent systems and are often episodic/semantic, replaying information from dialogue histories [18, 21, 24, 31, 38], such as A-MEM [34] which captures interactions as a network of interconnected notes that form an evolving memory structure, and Mem0 [6], which focus on managing memory from ongoing conversations. While these systems advance memory capabilities, they are not designed for agentic learning and workflow automation. Another line of works target memory optimization for agentic workflows including [12, 14, 40], which focus on short-term context optimization for workflow agents. Most closely related works on long-term, procedural memory for agents include Synapse [37], which uses successful past full trajectories as exemplars, and Agent Workflow Memory (AWM) [29] which induces frequently used subtask sequences as reusable skills. However, both works target procedural memory for single-agent scenarios. In contrast, LEGOMem introduces modular, role-aware procedural memory for multi-agent systems. By flexibly allocating memory across orchestrators and task agents, it addresses unique challenges in memory placement and allocation, improving workflow automation through better planning, execution, and coordination.

3 LEGOMEM: MODULAR PROCEDURAL MEMORY FOR MULTI-AGENT LLM SYSTEMS

In this section, we introduce LEGOMem, a modular procedural memory framework designed for multi-agent LLM systems. We begin by formalizing the problem setting of multi-agent workflow execution with procedural memory, then present the detailed LEGOMem framework, its variants and the design choices studied in our experiments.

3.1 Problem formulation

3.1.1 Multi-agent system for workflow automation. We consider a common multi-agent workflow automation framework (based on Magentic-One system [9]) with an orchestrator A_{orch} , a set of task agents $A = \{A_1, \dots, A_k\}$, and an external environment \mathcal{E} . A task T is specified by a natural language description d and must be executed within \mathcal{E} . Specifically, we implemented task agents for Word, Excel, Calendar, Email, System, and OCR-PDF apps. These task agents interact with the simulated apps in a Docker environment via tool APIs, ensuring isolated and reproducible execution.

The orchestrator first generates an initial high-level plan $\pi_0 = \{s_1, \dots, s_m\}$, outlining a possible sequence of subtasks. However, orchestration is not a static plan-following process: after each orchestration step, the orchestrator dynamically generates the next subtask based on the current state σ_t and observations returned from the agents, rather than simply selecting from the initial plan.

Formally, at each orchestration step t :

- (1) the orchestrator proposes the next subtask $s_t = \pi_{\text{orch}}(\sigma_t)$;
- (2) the subtask is assigned to an appropriate task agent A_j ;
- (3) the task agent executes s_t by issuing tool-use commands to the environment \mathcal{E} , returning an observation o_t and an execution summary r_t ;
- (4) the orchestrator updates its state $\sigma_{t+1} = f(\sigma_t, r_t)$ and continues orchestration.

If progress stalls (e.g., repeated states or looping behavior), the orchestrator may perform *re-planning*, generating a revised high-level plan π' , and resuming orchestration from the updated state. The system is considered successful if the final environment state $\sigma_{\text{final}} \in \mathcal{E}$ satisfies the task goal.

3.1.2 Multi-agent procedural memory. While the orchestration loop above defines how agents interact with the environment, it remains *stateless*: each new task T is solved from scratch, discarding knowledge from past executions. To address this limitation and enable agents to improve through experience, we introduce *multi-agent procedural memory*: modular, role-aware memories distilled from successful trajectories and reused across tasks in a multi-agent system. In contrast to episodic or semantic memory, which primarily capture events or textual information, multi-agent procedural memory abstracts workflows into reusable subroutines tailored to both orchestrators and task agents. These memories allow orchestrators to plan more effectively and select agents with greater context, while equipping task agents with execution-level guidance for more accurate and efficient tool use.

Formally, we define a memory store M as a collection of modular memory units derived from past executions. These include **full memories** that capture orchestration plans and summarized

execution traces, as well as **subtask memories** that capture agent-specific subtask executions. Together, they form a role-aware memory library that can be retrieved and allocated during inference to augment both planning and execution.

In the following section, we present LEGOMem, a concrete framework that implements this formulation through structured memory construction, inference-time allocation, and variant strategies for retrieval and memory reuse for more robust workflow automation.

3.2 The LEGOMem framework

The LEGOMem framework instantiates the problem formulation by equipping multi-agent systems with modular procedural memory. It operates in two phases: (i) an offline *memory construction* phase, where successful task trajectories are distilled into reusable memory units; and (ii) an online *memory-augmented inference* phase, where retrieved memories are allocated to the orchestrator and task agents to guide planning and execution. As illustrated in Figure 1a, past task trajectories are curated into a procedural memory bank, which is then queried at inference time to provide high-level orchestration guidance and agent-specific execution traces. Figure 1b further shows the structure of these memory units, consisting of a high-level plan, localized agent subtask traces, the final answer, and a brief reflection. This modular design enables LEGO-like recombination of past experiences to support efficient and reliable task completion across diverse multi-agent environments.

3.2.1 Memory construction. The first phase of LEGOMem is offline memory construction, where successful task trajectories are distilled into structured and reusable memory units. From each trajectory, we extract two complementary types of memory: (i) *full-task memories* that capture the task description, the high-level plan executed, and (ii) *subtask memories*, that encapsulate the subtask description, the localized agent behavior and tool-use, and observations. These modular units are stored in a procedural memory bank M for future reuse. At inference time, the orchestrator receives the full-task memory in its entirety, while task agents are provided with the relevant subtask memories.

Concretely, the construction process operates on execution logs of successfully completed tasks. Each log records the task description, the orchestrator’s planning and orchestration steps, the subtasks delegated to agents, and the corresponding agent executions (tool-use commands, observations, and outcomes). We use an LLM to transform these logs into structured LEGOMem units, as shown in Figure 1b. The resulting memory bank M is implemented as a vector database, indexed using dense embeddings. Let $\phi(\cdot)$ denote the embedding model used for indexing; for full-task memories, we compute $\phi(d)$ based on the task description d to enable semantic similarity retrieval. We implement and compare three retrieval and allocation strategies: vanilla LEGOMem, LEGOMem-Dynamic, and LEGOMem-QueryRewrite. For vanilla LEGOMem, the entire memory bank M is indexed using $\phi(d)$, enabling direct retrieval of relevant full-task memories at inference time and subtask allocation to task agents. For the other two variants, we separate the global memory bank M of full task memories, and subtask memory banks $\{M_{A_j} | A_j \in A\}$ per task agent, which contain the subtask memories that are easily extracted from the global memories, and

Algorithm 1 Multi-agent Execution with Vanilla LEGOMem

- 1: **Input:** task description d_{new} , memory bank \mathcal{M} , orchestrator A_{orch} , task agents $A = \{A_1, \dots, A_k\}$
 - 2: Compute embedding of $\phi(d_{\text{new}})$ and retrieve top- K semantically similar full-task memories $m = \{m_1, \dots, m_K\}$ from \mathcal{M} .
 - 3: Extract subtask memories $\{m_1^1, \dots, m_n^K\}$ from the full-task memories and assign subtask memories corresponding to each agent.
 - 4: Initialize environment \mathcal{E} and start task d_{new} .
 - 5: Augment retrieved full-task memories m to the orchestrator, which then generates initial plan π_0 .
 - 6: **while** task not completed **do**
 - 7: Orchestrator A_{orch} selects next agent $A_t \in A$, generates the next subtask s_t and assign to A_t ,
 - 8: Augment subtask memories to the task agent A_t
 - 9: Task agent A_t generates a list of tool-use actions, which are executed in the environment.
 - 10: Agent receives observation o_t , summarize subtask execution and sends summary message r_t to orchestrator A_{orch}
 - 11: **if** progress stalls **then**
 - 12: Orchestrator performs re-planning and update plan π'
 - 13: **end if**
 - 14: **end while**
 - 15: **return** orchestrator final response.
-

subtask memory banks are indexed by the embeddings of the subtask descriptions $\phi(d_{\text{subtask}})$. More details regarding the LEGOMem variants will be discussed in Section 3.3.

3.2.2 Memory-augmented inference. In the second phase, LEGOMem augments the task execution loop by supplying the *orchestrator* with full-task memories (end-to-end for planning and detailed orchestration) and augment *task agents* with subtask memories (localized execution guidance). Given a new task d_{new} , the system retrieves relevant memories from the memory banks and allocates them accordingly. We designed and tested three different LEGOMem variants which exhibit different memory retrieval strategies, which will be detailed in Section 3.3. Here we will describe the vanilla LEGOMem inference approach, as shown in Algorithm 1.

Given a new task with description d_{new} , we obtain the embedding $\phi(d_{\text{new}})$ and the system first retrieves top- K relevant memories from the global memory bank \mathcal{M} using semantic similarity. Then we allocate the full-task memory to the orchestrator, and extract the subtask memories from the retrieved full-task memories and allocate the subtask memories to the corresponding task agents. As shown in Figure 1a, for the vanilla LEGOMem variant, the orchestrator receives full-task memories that provide end-to-end workflows, while task agents are supplied with subtask memories that offer localized execution guidance. This design enables orchestrators to leverage prior trajectories for informed planning, agent capability grounding, and error recovery, while task agents improve their accuracy and efficiency in tool-use. As the task starts, the orchestrator receives the full memories and perform initial planning. Then, at each orchestration step, the orchestrator dynamically generates the next subtask using both the current state and retrieved full-task memories. The selected agent then executes the subtask with its allocated subtask memories, returning observations and summaries

to update the orchestrator state. If progress stalls, the orchestrator can re-plan using memory as additional guidance. Through this loop, LEGOMem integrates past experiences to make more informed decision during planning and coordination, improving both reliability and efficiency of the multi-agent workflows.

3.3 LEGOMem variants

To explore the impact of subtask retrieval granularity in multi-agent systems, we compare three variants of LEGOMem: (vanilla) LEGOMem, LEGOMem-Dynamic, and LEGOMem-QueryRewrite. These variants differ in how they store and retrieve subtask memories and allocate them to the task agents.

As discussed in 3.2, vanilla LEGOMem keeps a global procedural memory bank \mathcal{M} , and during inference, retrieves full-task memories using the task description and augment them to the orchestrator. Subtask memories are then extracted from these retrieved memories straightforwardly and are statically assigned to the relevant task agents. This approach is simple and efficient, and provides strong performance across teams. However, it may occasionally fail to surface relevant subtask memories for certain agents if the retrieved full-task memories differ in subtask structures from the current task. In such cases, even if the overall task appears similar, the subtask components may diverge. To address this, we implement two variants that enable finer-grained subtask-level retrieval, improving task agent-level memory relevance:

LEGOMem-Dynamic: As illustrated in Figure 2a, LEGOMem-Dynamic performs subtask-level retrieval during execution. The orchestrator memory storage and retrieval remain the same as the vanilla version, while the system maintains per-agent subtask memory banks segmented from the global memory bank. When the orchestrator generates a subtask s_t for an agent A_t , we compute its embedding $\phi(s_t)$ and query the agent’s memory bank \mathcal{M}_{A_t} to retrieve only the most relevant past subtask traces. This just-in-time retrieval provides more precise execution guidance for task agents and reduces noise from irrelevant memories.

LEGOMem-QueryRewrite: While LEGOMem-Dynamic performs just-in-time retrieval at each orchestration step, it incurs repeated subtask embedding and retrieval during execution. LEGOMem-QueryRewrite shifts this to the planning stage using query rewriting [16, 17]. As shown in Figure 2b, after retrieving full-task memories, a query rewriter LLM ψ uses the memories to generate a draft plan for the new task $\pi'_{\text{draft}} = \{s'_1, s'_2, \dots, s'_n\}$ consisting of rewritten subtasks. Each s'_j is then embedded via $\phi(s'_j)$ and used to retrieve relevant subtask memories from the corresponding agent’s memory bank \mathcal{M}_{A_j} before task execution starts. This approach preserves the fine-grained retrieval benefits of LEGOMem-Dynamic while avoiding repeated queries at runtime, enabling more efficient execution and smoother orchestration.

Interestingly, our experiments show that all three variants achieve similar overall performance in full memory settings, demonstrating the robustness across variants. Furthermore, our ablation study shows that LEGOMem-Dynamic and LEGOMem-QueryRewrite outperform vanilla LEGOMem when only task agent-level memory is used and with small language model task agents. This indicates

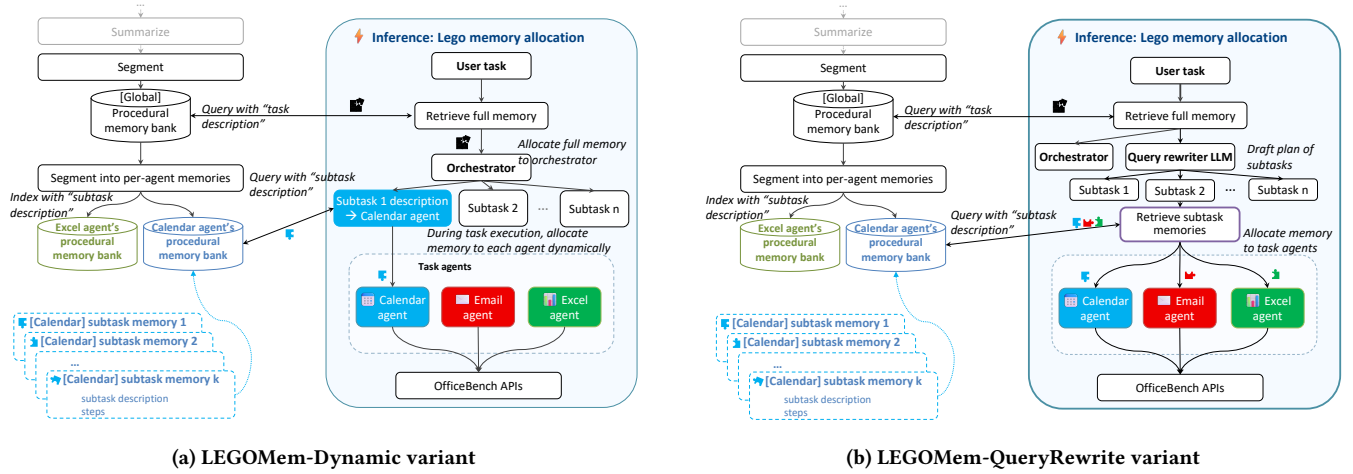


Figure 2: Comparison of LEGOMem variants: (a) LEGOMem-Dynamic dynamically retrieves subtask memories during execution, and (b) LEGOMem-QueryRewrite employs query rewriting to retrieve multiple candidate memories for each subtask.

that fine-grained subtask retrieval may offer more relevant guidance to task agents and may be particularly beneficial in settings with weaker orchestrator support.

Together, the LEGOMem framework and its variants provide a general and modular approach to procedural memory for multi-agent LLM systems, enabling both orchestrators and task agents to learn from and reuse prior task executions. In the following section, we empirically evaluate these variants across different agent team configurations and memory settings.

4 EXPERIMENTS

We evaluate LEGOMem on the OfficeBench benchmark, comparing its variants with strong baselines across LLM-only, hybrid, and SLM-only multi-agent teams. Beyond overall performance, we conduct ablations on memory placement, retrieval strategies, and representation formats to analyze the contributions of different design choices. Our results show that LEGOMem consistently improves task success rates across team configurations, and that memory design, particularly the placement of orchestrator memory, plays a central role in enabling effective multi-agent coordination.

4.1 Experimental setup

4.1.1 Dataset and metrics. We evaluate the agents on the OfficeBench, which consists of multi-step office automation tasks with varying levels of complexity. We split the 300 tasks into training (148 instances, for memory curation) and test (152 instances, for evaluation) sets. Tasks span three difficulty levels: Level 1 (single application), Level 2 (two-application), and Level 3 (multi-application workflows).

The evaluation metric is the **success rate**, i.e. the percentage of tasks solved correctly. The success of a task is evaluated programmatically according to the final state of the environment, including exact match or fuzzy keyword match of the final outputs and expected outputs (e.g., correctly updated spreadsheet entries, calendar events, emails sent and received, and question answering).

4.1.2 Implementation details. We experiment with three team configurations with agents of different sizes and capabilities:

- **LLM team:** for the full LLM team, we use GPT-4o [11] for both the orchestrator and task agents
- **Hybrid (LLM + SLM) team:** GPT-4o for the orchestrator, and GPT-4o-mini for the task agents
- **SLM team:** GPT-4o-mini [11] for all components

Additionally, for memory storage and retrieval, we use the OpenAI text-embedding-3-large model for embedding the task descriptions, and the FAISS library [8] for the vector database. For the OCR app, we use the Phi-3.5-mini model [1] as the vision language model for image parsing.

We compare the LEGOMem variants with three baselines: (i) **No memory**, and two state-of-the-art methods on procedural memory for workflow automation (ii) **Synapse**, which augments agents with semantically similar memories using raw action sequences and full trajectories, and (iii) **AWM**, which augments agents with summarized subtask memories extracted from full trajectories.

4.1.3 Memory curation and agent inference details. Memory construction uses the 148 training tasks, where we first run the full LLM agent team without memory, and filter for successful trajectories and extracted 93 full task memories from the successful trajectories. For the LEGOMem variants, we further extracted 250 subtask memories for the task agents from the 93 full task memories. Both Synapse and AWM focus on single-agent systems; for a fair comparison we use the same 93 successful trajectories, and adapt both baselines to the multi-agent team, augmenting the memories to both orchestrators and task agents. For Synapse, we augment both orchestrators and task agents with the full trajectories. For AWM, we cluster the successful trajectories, to extract and consolidate subtask memories from each cluster, and during inference, we augment the task agents with their corresponding extracted subtask memories and augment the orchestrator with a list of extracted subtask memories. For all variants, we use 5 memories for orchestrator and 3 memories for each task agent from the successful trajectories.

Table 1: Performance comparison across memory variants, task levels, and multi-agent teams. Results show mean success rates across different LEGOMem variants compared with baseline methods, each data-point is averaged over three random seeds.

	LLM team				Hybrid (LLM + SLM) team				SLM team			
	Level 1	Level 2	Level 3	Overall	Level 1	Level 2	Level 3	Overall	Level 1	Level 2	Level 3	Overall
Baseline methods												
No memory	49.31	58.52	33.33	45.83	45.14	48.89	16.95	35.31	36.81	34.81	7.34	24.78
Synapse	59.72	75.56	43.50	58.11	46.53	68.15	29.94	46.49	36.81	42.22	20.90	32.24
AWM	54.17	58.52	35.03	48.03	43.75	55.56	18.64	37.50	35.42	36.30	12.99	26.97
Our methods												
LEGOMem	57.99	73.33	47.46	58.44	49.31	62.22	36.16	48.03	38.89	54.07	25.42	38.16
LEGOMem-Dynamic	56.25	75.56	43.79	57.12	44.44	65.93	36.16	47.59	38.89	50.37	27.12	37.72
LEGOMem-QueryRewrite	54.17	72.59	42.94	55.26	47.22	66.67	40.11	50.22	36.81	48.89	26.55	36.40

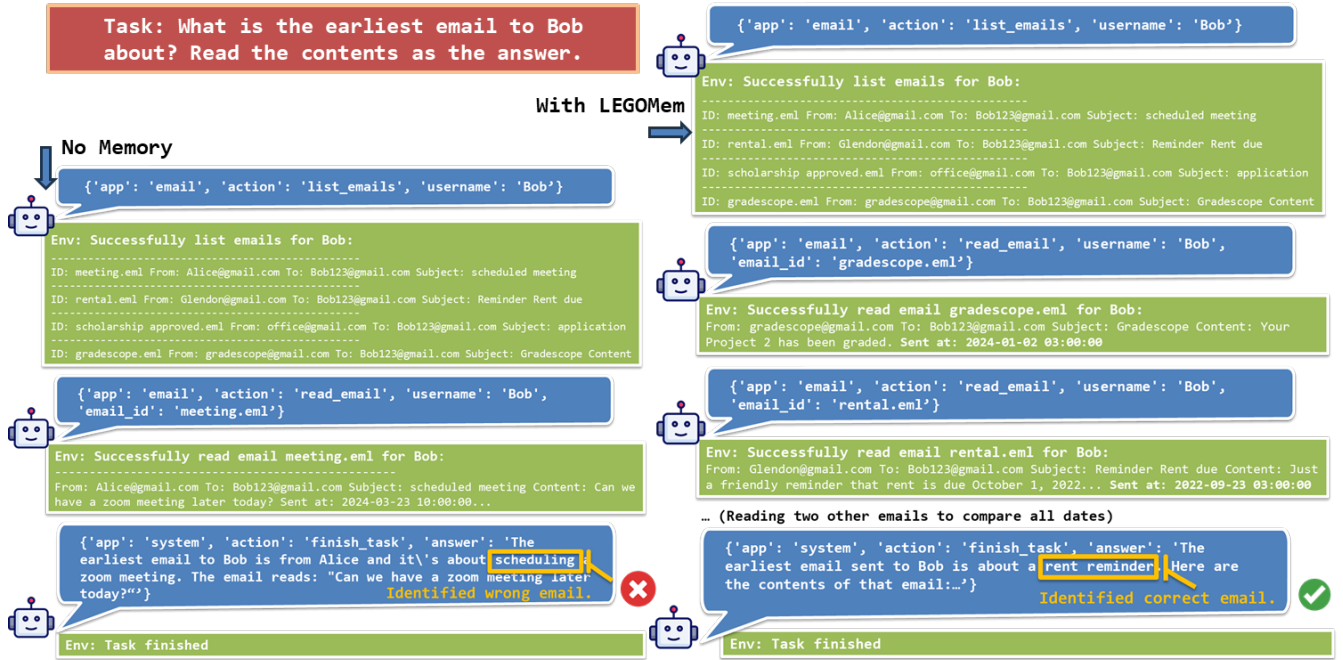


Figure 3: Qualitative example of agent execution with and without memory. The memory-less team fails to identify the earliest email due to incomplete planning, stopping after reading the first email, while the team with LEGOMem systematically reads to obtain and compare all email timestamps, producing the correct answer.

4.2 Main results

Table 1 presents the main experiment results, comparing the performance of LEGOMem with baseline methods across different task levels and agentic team configurations.

Across all scenarios and agent team configurations, LEGOMem variants consistently outperform baseline methods in terms of overall success rate. All three LEGOMem variants show similar, consistent performance, with the vanilla LEGOMem variant being lightweight while achieving the best overall performance. The performance improvement shows the effectiveness of modularized memory representations and allocation for multi-agent systems. Compared with memory-less teams, LEGOMem improves overall

task success rate by +12.61%, +12.72%, and +13.38% absolute points on LLM, Hybrid and SLM teams, respectively.

Importantly, LEGOMem enables smaller models to close the gap with, and sometimes outperform, larger ones. For example, the Hybrid team with LEGOMem-QueryRewrite achieves 50.22%, surpassing the memory-less LLM team (45.83%). Likewise, a full SLM team with vanilla LEGOMem (38.16%) outperforms the Hybrid team without memory (35.31%). While Synapse remains competitive in LLM teams, reflecting the ability of LLMs to interpret raw procedural traces, its effectiveness is less consistent for Hybrid and SLM teams. In contrast, LEGOMem maintains strong performance across all team settings, highlighting the importance of modularized procedural memory for enabling efficient, smaller-model teams.

Table 2: Comparing performance with various memory placement mechanism across LEGOMem variants.

	LLM variants				Hybrid (LLM + SLM) variants			
	Level 1	Level 2	Level 3	Overall	Level 1	Level 2	Level 3	Overall
Orchestrator + Agent memory								
LEGOMem	57.99	73.33	47.46	58.44	49.31	62.22	36.16	48.03
LEGOMem-Dynamic	56.25	75.56	43.79	57.12	44.44	65.93	36.16	47.59
LEGOMem-QueryRewrite	54.17	72.59	42.94	55.26	47.22	66.67	40.11	50.22
Orchestrator memory (planning) + Agent memory								
LEGOMem	54.86	76.30	35.03	53.51	45.14	63.70	30.51	44.96
LEGOMem-Dynamic	54.86	73.33	41.81	55.26	46.53	64.44	32.77	46.49
LEGOMem-QueryRewrite	51.39	70.37	42.94	53.73	49.31	59.26	35.59	46.93
Orchestrator memory								
LEGOMem	51.39	74.07	38.98	53.29	45.83	68.89	32.77	47.59
Task Agent memory								
LEGOMem	50.00	63.70	38.98	49.78	44.44	46.67	19.21	35.31
LEGOMem-Dynamic	49.31	62.96	38.98	49.34	47.22	55.56	23.16	40.35
LEGOMem-QueryRewrite	54.86	66.67	35.03	50.66	44.44	54.81	24.29	39.69
No memory								
No memory	49.31	58.52	33.33	45.83	45.14	48.89	16.95	35.31

Table 3: Comparing memory with and without reasoning across different LEGOMem variants.

	LLM team				Hybrid (LLM + SLM) team			
	Level 1	Level 2	Level 3	Overall	Level 1	Level 2	Level 3	Overall
No reasoning								
LEGOMem	54.17	75.93	43.22	56.36	48.61	68.15	36.72	49.78
LEGOMem-Dynamic	56.94	73.33	44.63	57.02	50.00	68.15	36.72	50.22
LEGOMem-QueryRewrite	48.61	69.63	48.02	54.61	37.50	65.19	36.16	45.18
With reasoning								
LEGOMem	57.99	73.33	47.46	58.44	49.31	62.22	36.16	48.03
LEGOMem-Dynamic	56.25	75.56	43.79	57.12	44.44	65.93	36.16	47.59
LEGOMem-QueryRewrite	54.17	72.59	42.94	55.26	47.22	66.67	40.11	50.22

To better illustrate the effect of memory on agent behavior, Figure 3 presents a qualitative case study. Without memory, the agent fails to identify the earliest email due to incomplete planning, stopping after reading only the first entry. With LEGOMem, the agent systematically reads and compares all emails, correctly identifying the earliest one. This example highlights how LEGOMem improves reasoning consistency and task completeness beyond what is reflected in aggregate success rates.

4.3 Ablations experiments

This section investigates how different memory retrieval, allocation, and placement strategies affect the performance of LEGOMem.

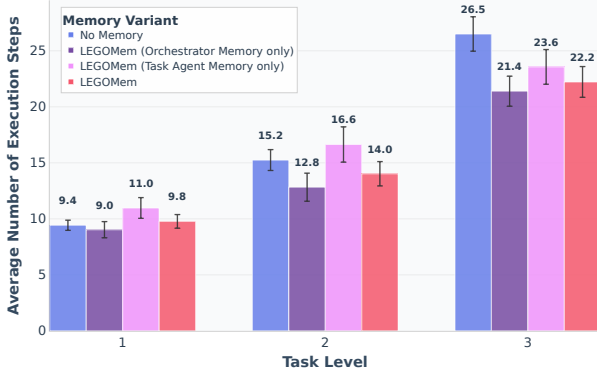
4.3.1 Memory retrieval, allocation, and placement. Table 2 summarizes our ablation results across different memory retrieval variants, memory allocation strategies, and memory placement settings.

Memory retrieval. The three subtask memory retrieval strategies—vanilla LEGOMem, LEGOMem-Dynamic, and LEGOMem-QueryRewrite—all perform robustly and achieve similar overall

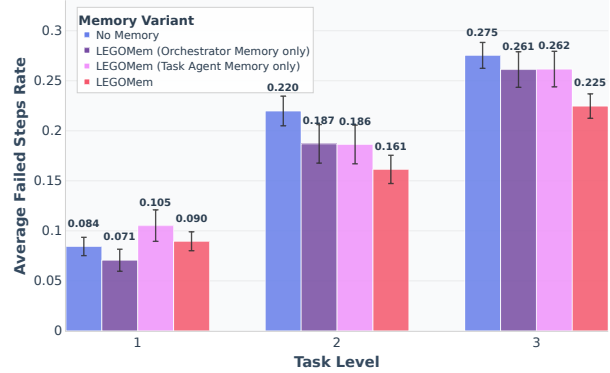
success rates. While dynamic retrieval enables more targeted allocation and query rewriting improves robustness to subtask phrasing variations, these differences are modest compared to the impact of memory placement and allocation strategy.

In the task-agent-only memory setting, both LEGOMem-Dynamic and LEGOMem-QueryRewrite outperform vanilla LEGOMem by 4–5% on average in the Hybrid team where task agents are smaller models and agent-level memory plays a more critical role. These results highlight the advantage of fine-grained subtask retrieval in providing more relevant and contextual guidance to task agents, especially when global planning signals are weaker. We hypothesize that the similar overall performance of all three variants in full-memory settings may be due to the strength of the orchestrator memory, where the orchestrator receives the complete trajectory of prior solutions, compensating for weaker task agent execution by enabling better task decomposition and delegation.

Overall, these findings demonstrate the flexibility of the LEGOMem framework: even the lightweight vanilla variant performs competitively, while more advanced variants offer additional benefits in settings that demand finer-grained memory retrieval.



(a) Average execution steps across task levels



(b) Average failed steps rate across task levels

Figure 4: Ablations study: execution steps comparison for different LEGOMem memory placement for LLM teams. (a) shows that LEGOMem variants reduce the number of execution steps required, with up to 16.2% reduction for Level 3 tasks. (b) shows lower failure rates of steps, indicating more reliable task execution with procedural memory.

Memory allocation. Regarding memory allocation, we find that joint allocation of orchestrator and task agent memory (Orchestrator + Agent memory variant) yields the strongest overall results, with orchestrator memory supporting effective planning, task decomposition and subtask orchestration, and task agent memory enabling execution-level precision. Orchestrator memory emerges as essential: when memory is removed from the orchestrator and provided only to task agents (Task Agent memory variant), performance drops noticeably.

Memory placement. Looking at memory placement, even when restricted to the planning and replanning stages, orchestrator memory still improves over task-agent-only variants, confirming its central role in guiding high-level planning and task decomposition. Finally, Task-Agent-only memory while facilitating more accurate tool use and outperforming the no-memory baseline, remains less effective than orchestrator-level memory – indicating that local memory without global coordination is insufficient.

4.3.2 Effectiveness of adding reasoning in memory. We also examine whether augmenting procedural memories with lightweight reasoning improves performance. As shown in Table 3, the differences are minor: overall scores change by less than two points across variants and team types. For example, vanilla LEGOMem improves slightly on LLM teams (56.36% \rightarrow 58.44%) but decreases on Hybrid teams (49.78% \rightarrow 48.03%). These results suggest that LEGOMem is robust, with its modularized structure already providing sufficient procedural guidance without additional reasoning steps.

4.3.3 Effectiveness of memory on execution steps and failure rates. As an additional ablations study, Figure 4 compares the average number of execution steps taken by the agent with different memory placement variants and the step failure rate (due to wrong tool-use actions issued) per task for the LLM team. As shown in Figure 4a Compared to the no memory variant, the agents equipped with LEGOMem can reduce the number of execution steps required to complete the tasks, for example, a -16.2% drop from an average of

26.5 to 22.2 steps for Level 3 tasks. The task memory only variant where we remove the orchestrator memory required more steps to complete a task compared with the variant with orchestrator memory, due to the effectiveness of the orchestrator memory for improved planning.

Similarly, Figure 4b shows that LEGOMem reduces the average failure rate of agent steps. At Level 3, the failure rate decreases from 0.275 in the no-memory setting to 0.225 with LEGOMem. These results indicate that LEGOMem not only improves task success rates but also enables more efficient and reliable task execution.

In summary, our experiments show that LEGOMem consistently outperforms baseline methods, improving task success by over 12 absolute percentage points compared with memory-less teams. LEGOMem can enable smaller and hybrid teams to match or even surpass LLM-only teams, highlighting its value for efficient multi-agent configurations. Ablations reveal that, as one may expect, the memory placement strategy is critical: orchestrator memory is essential for effective planning, while subtask memory complements execution. Additional analysis also show reductions in execution steps required and per-step failure rates with LEGOMem.

5 CONCLUSION

We introduced LEGOMem, a modular procedural memory framework for multi-agent systems that enables orchestrators and task agents to learn from prior task executions. By representing workflows as reusable memory units—split into full-task and subtask components—LEGOMem supports efficient task planning and execution through memory retrieval and allocation. We implemented and evaluated three LEGOMem variants to explore the design space of memory retrieval and placement strategies. Across extensive experiments on workflow automation tasks, we show that LEGOMem significantly improves task success rates over memory-less and baseline methods, with orchestrator memory playing a critical role in planning and coordination, and memory can also benefit smaller

agents, highlighting the flexibility and effectiveness of the framework. Our work shows that integrating procedural memory into multi-agent systems enables more reliable and reusable solutions. Future work may explore continual learning also from failed past trajectories, and scaling LEGOMem to open-ended environments and tool ecosystems.

REFERENCES

- [1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. 2024. Phi-4 technical report. *arXiv preprint arXiv:2412.08905* (2024).
- [2] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691* (2022).
- [3] Ruisheng Cao, Fangyu Lei, Haoyuan Wu, Jixuan Chen, Yeqiao Fu, Hongcheng Gao, Xinzhuang Xiong, Hanchong Zhang, Wenjing Hu, Yuchen Mao, et al. 2024. Spider2-v: How far are multimodal agents from automating data science and engineering workflows? *Advances in Neural Information Processing Systems* 37 (2024), 107703–107744.
- [4] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848* 2, 4 (2023), 6.
- [5] Yuheng Cheng, Ceyao Zhang, Zhengwen Zhang, Xiangrui Meng, Sirui Hong, Wenhao Li, Zihao Wang, Zekai Wang, Feng Yin, Junhua Zhao, et al. 2024. Exploring large language model based intelligent agents: Definitions, methods, and prospects. *arXiv preprint arXiv:2401.03428* (2024).
- [6] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. <https://doi.org/10.48550/arXiv.2504.19413> arXiv:2504.19413 [cs].
- [7] Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, et al. 2025. Multi-Agent Collaboration via Evolving Orchestration. *arXiv preprint arXiv:2505.19591* (2025).
- [8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).
- [9] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, et al. 2024. Magentic-one: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468* (2024).
- [10] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).
- [11] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).
- [12] Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. 2025. ACON: Optimizing Context Compression for Long-horizon LLM Agents. *arXiv preprint arXiv:2510.00615* (2025).
- [13] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An llm compiler for parallel function calling. In *Forty-first International Conference on Machine Learning*.
- [14] Dongjun Lee, Juyong Lee, Kyuyoung Kim, Jihoon Tack, Jinwoo Shin, Yee Whye Teh, and Kimin Lee. 2025. Learning to contextualize web pages for enhanced decision making by LLM agents. *arXiv preprint arXiv:2503.10689* (2025).
- [15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.
- [16] Zhicong Li, Jiahao Wang, Hangyu Mao, ZhiShu Jiang, Zhongxia Chen, Du Jiazhen, Fuzheng Zhang, Di ZHANG, and Yong Liu. [n.d.]. DMQR-RAG: Diverse Multi-Query Rewriting in Retrieval-Augmented Generation. ([n.d.]).
- [17] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting in retrieval-augmented large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 5303–5315.
- [18] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753* (2024).
- [19] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.
- [20] Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. 2023. Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning. *arXiv preprint arXiv:2307.06135* (2023).
- [21] Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. 2025. Zep: a temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956* (2025).
- [22] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*. 2998–3009.
- [23] Peter Stone and Manuela Veloso. 2000. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots* 8, 3 (2000), 345–383.
- [24] Haoran Sun and Shaoning Zeng. 2025. Hierarchical Memory for High-Efficiency Long-Term Reasoning in LLM Agents. *arXiv preprint arXiv:2507.22925* (2025).
- [25] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.
- [26] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (2023).
- [27] Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. 2025. OdysseyBench: Evaluating LLM Agents on Long-Horizon Complex Office Application Workflows. *arXiv preprint arXiv:2508.09124* (2025).
- [28] Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. 2024. Officebench: Benchmarking language agents across multiple applications for office automation. *arXiv preprint arXiv:2407.19056* (2024).
- [29] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024. Agent Workflow Memory. <https://doi.org/10.48550/arXiv.2409.07429> arXiv:2409.07429 [cs].
- [30] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.
- [31] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. 2024. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813* (2024).
- [32] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.
- [33] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. Os-world: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems* 37 (2024), 52040–52094.
- [34] Wujiang Xu, Kai Mei, Hang Gao, Juntao Tan, Zujie Liang, and Yongfeng Zhang. 2025. A-MEM: Agentic Memory for LLM Agents. <https://doi.org/10.48550/arXiv.2502.12110> arXiv:2502.12110 [cs].
- [35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- [36] Chaoyun Zhang, Lihun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, et al. 2024. Ufo: A ui-focused agent for windows os interaction. *arXiv preprint arXiv:2402.07939* (2024).
- [37] Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2023. Synapse: Trajectory-as-exemplar prompting with memory for computer control. *arXiv preprint arXiv:2306.07863* (2023).
- [38] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19724–19731.
- [39] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* (2023).
- [40] Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. 2025. MEM1: Learning to Synergize Memory and Reasoning for Efficient Long-Horizon Agents. *arXiv preprint arXiv:2506.15841* (2025).

A PROMPTS FOR MEMORY CURATION

In this section, we provide the detailed prompts for memory curation and the prompt for the query rewriting LLM.

Prompt 1: Memory Curation Prompt

From the following agent trajectory, generate memory that can be useful for future LLM agents' reference.

Trajectory: {full_trajectory}

Example:

{start_tag}

```
{{ "high_level_plan": "1. Check Bob's calendar availability for the specified time slot. 2. Add the meeting to Bob's calendar for 5172024 from 10:30 a.m. to 11:00 a.m.", "subtasks": [ {{ "agent": "calendar_agent", "description": "Check Bob's schedule on 5/17/2024 from 10:30 a.m. to 11:00 a.m. to ensure there are no conflicts", "steps": "<think>I need to check Bob's existing calendar events to ensure no scheduling conflicts</think><action>{{\"app\" : \"calendar\", \"action\" : \"list_events\", \"username\" : \"Bob\"}}</action>", "observations": "No events found for Bob - calendar is available for the requested time slot" }}, {{ "agent": "calendar_agent", "description": "Add a meeting to Bob's calendar on 5/17/2024 from 10:30 a.m. to 11:00 a.m.", "steps": "<think>Since no conflicts were found, I can now create the new calendar event for Bob</think><action>{{\"app\" : \"calendar\", \"action\" : \"create_event\", \"user\" : \"Bob\", \"summary\" : \"Meeting\", \"time_start\" : \"2024 - 05 - 17 10 : 30 : 00\", \"time_end\" : \"2024 - 05 - 17 11 : 00 : 00\"}}</action>", "observations": "Successfully created a new event in Bob's calendar for the specified date and time" }}, {{ "agent": "calendar_agent", "description": "Task completed successfully without any conflicts or errors. The calendar check confirmed availability, and the meeting was created with proper date/time formatting" }} ] }, "final_answer": "The meeting has been successfully added to Bob's calendar on 5172024 from 10:30 a.m. to 11:00 a.m.", "reflections": "Task completed successfully without any conflicts or errors. The calendar check confirmed availability, and the meeting was created with proper date/time formatting" }}}
```

Instructions:

Please analyze the trajectory and extract structured memory with clear thinking and well-formed actions. Use the following format for each subtask step: <think>reasoning about what needs to be done and why this action is appropriate</think> <action>precise tool call command in structured format</action>

The memory object should be formatted as follows: {{ "high_level_plan": "<a string that lists the high-level steps taken and which agent performs each subtask>", "subtasks": [{{ "agent": "<copy the exact name of agent that performed the subtask>", "description": "<description of the subtask given by the orchestrator>", "steps": "<Copy the precise actions taken with think-action structure: <think>reasoning</think><action>tool_call</action>, repeat for each action. Omit some actions if there are too many similar commands (>10). Remove actions that yielded errors or were malformed.>", "observations": "<a very brief summary of the key observations from the function execution results>", }, ...], "final_answer": "<The final answer given by the orchestrator or answer agent>", "reflections": "<a concise summary that lists what was successful, what were specific failures, root cause of which action and how to avoid, if any>" }}}

Rules to follow:

1. Group together actions into subtasks if they are related and can be done together.
2. For each action in the steps field, use the think-action format with clear reasoning followed by structured tool calls.
3. When copying actions, remove function call IDs but keep the essential tool call structure.
4. Only include successful actions; omit actions that resulted in errors. If there are too many repeated similar actions, truncate and omit some, and if the action parameters (such as contents to write to a word document) are too long, you can summarize it.
5. Keep observations very concise but informative.
6. Do not include orchestrator coordination steps in the subtasks.
7. For the subtask steps field, use a string format with think-action pairs, not a list.

Follow the JSON format exactly to ensure it can be parsed automatically, and put the json object between the tags {start_tag} # your json here {end_tag} and do not use markdown.

Prompt 2: Query Rewriting Prompt

Based on the following similar task examples, break down the new task into a step-by-step plan.

Similar Task Examples: {memory_context}

New Task: {task_description}

Please provide a numbered list of 3-5 high-level steps that would be needed to complete this task. Focus on the main phases/subtasks, not detailed actions.

Format your response as a simple numbered list enclosed within <start> and <end> tags:

<start> 1. [First step] 2. [Second step] 3. [Third step] ... <end>

Steps: