Hierarchical Experience Memory for Multi-Agent Task Planning and Subtask Guidance

Marcus Lee
Independent
mark.2025010101@gmail.com

Abstract

This study presents Hierarchical Memory Planning, a self-evolving hierarchical memory framework designed to improve multi-agent task planning for complex reasoning tasks. The framework enhances the planning capability of large language model (LLM) agents by accumulating experience from previous task trajectories and grounding future decomposition decisions in retrieved memory. In Hierarchical Memory Planning, a centralized meta-agent decomposes each user query into subtasks and assigns them to specialized subagents, while a plan-level memory retrieves similar historical cases and distills reusable insights from comparisons between lower-scoring and higher-scoring trajectories. To reduce the mismatch between high-level planning and subagent execution, we further introduce a subagent-level memory mechanism for task planning, which augments each raw subtask produced by the initial plan with task-execution guidance distilled from prior subagent experience. This hierarchical design assigns distinct functions to the two memory levels: plan-level memory guides global decomposition, while subagent-level memory improves the executability of individual subtasks. Through experiments on DROP-eval and IIRC-eval, using 150 training examples from IIRC and DROP as the experience pool, Hierarchical Memory Planning achieves accuracies of 37.0 and 43.5, respectively. These results indicate that hierarchical experience memory can support multi-agent task planning under the studied setting.

1 Introduction

Large language models (LLMs) have substantially advanced autonomous decision-making, enabling agents to address complex, long-horizon tasks through multi-step reasoning, tool use, and interaction with external environments (Kim et al., 2024; Parmar et al., 2025). Because single-agent systems often face cognitive limits, restricted context windows, and error propagation when tackling compositional goals (Hao et al., 2026), a growing body of work explores collaborative multi-agent systems (MAS) organized around a divide-and-conquer principle (Zhang et al., 2025). A common design employs a centralized meta-agent as a high-level planner that decomposes a global objective into a sequence of subordinate subtasks and assigns them to specialized worker agents (Zhang et al., 2025; Hao et al., 2026). While such modular task allocation scales well in principle, current orchestration frameworks remain constrained by two practical difficulties: meta-agent planning is often static across related tasks, and a plan-execution gap may arise when worker agents are asked to carry out abstract subtasks (Zhang et al., 2026b; Hao et al., 2026).

We observe one recurring failure mode at the interface between task decomposition and subtask execution. Even when a meta-agent produces a coherent sequence of steps and routes them to appropriate worker agents, a specialized worker may not recover the intended execution behavior from a raw, abstract instruction in isolation (Zhang et al., 2026b). When worker agents act on these underspecified subtasks, their outputs can violate downstream requirements, and these local inconsistencies may accumulate across a multi-step trajectory (Zhang et al., 2026b; Parmar et al., 2025; Zhang et al., 2025).

In this paper, we study multi-agent task planning under repeated, related task instances, where the central question is how to make high-level decomposition both adaptive over time and reliably executable by downstream agents. We propose Hierarchical Memory Planning, a self-evolving planning framework that separates experience into two memory levels: a plan-level memory module for meta-agent task planning and a subagent-execution-aware memory module for refining planned subtasks. The plan-level memory archives past plan-

Proceedings of the 19th International Natural Language Generation Conference: Long Papers, pages 559–571
October 17-21, 2026 ©2026 Association for Computational Linguistics

ning trajectories and their feedback and retrieves similar cases through a weighted combination of sentence-level and task-attribute-aware similarity. Rather than replaying verbose raw traces into the context window, it conditions future decomposition on compact insights distilled by contrasting lower- and higher-scoring trajectories across cases, together with concise plans and execution evaluations from higher-scoring cases (Hao et al., 2026; Lin et al., 2026). The subagent-execution-aware memory is used during planning to refine initially generated subtasks by adding task-execution guidance, including subtask-specific insights, constraints, dependencies, and output requirements, while preserving the original semantic intent of the plan. These subtask-specific insights are distilled by comparing the corresponding subagent's past successful and unsuccessful execution trajectories.

We evaluate Hierarchical Memory Planning on DROP-eval and IIRC-eval under both offline and online protocols, and on HuskyQA under the online protocol only. The results across the reported settings indicate gains over the Meta-Agent no-memory baseline; sensitivity analyses further examine the effects of experience-pool size and the insight-label threshold.

Our contributions are summarized as follows. First, we formulate a self-evolving multi-agent task planning setting in which the meta-agent accumulates experience from a sequence of distinct but related task instances, aiming to improve decomposition quality while producing subtasks that better support reliable execution by specialized workers. Second, we introduce a hierarchical memory framework for this setting, consisting of a plan-level memory module for meta-agent task planning and a subagent-execution-aware memory module for planning-time subtask refinement. Finally, we evaluate Hierarchical Memory Planning offline on DROP-eval and IIRC-eval and online on DROP-eval, IIRC-eval, and HuskyQA, with the observed results indicating gains over the Meta-Agent no-memory baseline under the studied protocols.

2 Related Work

LLM-based multi-agent planning. Recent agentic LLM systems have shifted from single-agent prompting toward collaborative orchestration, where agents assume specialized roles and coordinate through structured workflows (Hong et al., 2024; Kim et al., 2024). Prompting methods decompose complex problems into modular subproblems or progressively composed reasoning steps (Khot et al., 2022; Zhou et al., 2022), and multi-agent planners further assign such subtasks to differentiated workers through structured decomposition, role specialization, and inference-time strategy selection (Parmar et al., 2025; Zhang et al., 2025). Recent studies also examine planning quality under long trajectories, suggesting that optimizing over execution traces improves downstream reliability (Zhang et al., 2026b). We follow this line but improve decomposition quality across repeated task instances through experience memory, rather than searching for a better plan within a single instance.

Planning-execution mismatch and constraint grounding. A recurring challenge in hierarchical orchestration is that a globally coherent plan is not always locally executable by underspecified workers. Prior work reports such misalignment and shows that explicit interfaces, structured intermediate representations, verification-aware decomposition, and explicit task constraints improve execution stability and robustness (Zhang et al., 2026b; Guo et al., 2024; Zhang et al., 2025; Parmar et al., 2025). Instead of modifying worker architectures, we refine the planning-to-execution interface by attaching memory-derived execution guidance to each subtask before dispatch.

Memory-augmented and self-evolving agent systems. External memory allows agents to reuse experience across episodes without updating model parameters, from episodic reflections (Park et al., 2023; Shinn et al., 2023) and experiential distillation of successful and failed trajectories (Zhao et al., 2024), to scalable memory extraction and storage (Chhikara et al., 2025) and autonomous, stateful reflective memory (Wu et al., 2026; Zhou et al.; Wang; Liang et al., 2025). In multi-agent settings, self-evolution frameworks use reflection and experience distillation to inform coordination (Hao et al., 2026; Lin et al., 2026), and G-Memory traverses insight, query, and interaction graphs to retrieve general insights and condensed trajectories for framework-level augmentation (Zhang et al., 2026a).

Hierarchical Memory Planning differs from the two most related lines in both scope and mechanism. Unlike ExpeL (Zhao et al., 2024), which improves a single agent's task-solving policy from same-task repeated attempts retrieved by task simi-

larity, Hierarchical Memory Planning targets centralized multi-agent planning and applies experience at two decision points—meta-agent decomposition and worker-specific subtask guidance—distilling from scored positive and negative records collected across cases and retrieving with a hybrid of semantic and task-attribute similarity. Unlike G-Memory (Zhang et al., 2026a), which builds a graph over collaboration history, Hierarchical Memory Planning keeps separate trajectory and worker-specific memory stores and appends a worker-conditioned directive to each subtask before execution, targeting the planning-to-execution interface rather than general collaboration-history retrieval.

3 Problem Formulation

3.1 Centralized Multi-Agent Task Planning

We model a multi-agent system as a tuple $\mathcal{S} = (A, E, C, \Omega)$, where $A = \{a_0, a_1, \ldots, a_n\}$ is a collection of agents, E is the external environment, C is the communication topology, and Ω is the orchestration policy. We focus on the centralized topology, in which a_0 is a meta-agent and $a_1, \ldots, a_n$ are specialized worker agents. Each worker agent a_j has a natural-language capability description d_j and a tool set $\mathcal{T}_j$. Let $\mathcal{D} = \{d_1, \ldots, d_n\}$ denote the set of worker agent descriptions.

Given a user query Q, the meta-agent produces an ordered plan consisting of m subtasks and their worker-agent assignments:

$$\mathcal{P}(Q, \mathcal{D}, A) = \left\langle (q_i, a_i') \right\rangle_{i=1}^{m}, \quad a_i' \in \{a_1, \ldots, a_n\}. \tag{1}$$

where q_i is the i-th subtask in execution order and a_i' is its assigned worker agent. Each worker agent returns a response $r_i = a_i'(q_i)$. Following the output convention of AOP (Li et al., 2025), the response produced by the last worker agent in the ordered plan is used as the final answer.

Following AOP (Li et al., 2025), plans are represented as ordered JSON objects containing subtasks, worker assignments, rationales, and dependencies.

3.2 Memory-Augmented Task Planning

We consider a sequence of related task instances $\{Q^{(1)}, \ldots, Q^{(T)}\}$. After each task, the system obtains feedback at two levels: an overall task score and subtask-level scores. Let

$$\tau^{(t)} = \Big(Q^{(t)}, \Big\langle (q_i^{(t)}, a_i'^{(t)}, r_i^{(t)}, s_i^{(t)}, e_i^{(t)}) \Big\rangle_{i=1}^{m_t}, s^{(t)}, e^{(t)}\Big) \tag{2}$$

denote the trajectory for instance t, where $s^{(t)}$ is the overall task score, $e^{(t)}$ is the explanation for the overall task score, $s_i^{(t)}$ is the score for the i-th subtask, and $e_i^{(t)}$ is the explanation for the i-th subtask score.

To construct memory labels, we evaluate the final response and each worker-agent response without providing ground-truth answers to the evaluator. This memory-label evaluator is distinct from the benchmark evaluator used to report final accuracy in Section 5.1. We use `gpt-4o-2024-11-20` as the ground-truth-free LLM judge for memory labeling. Following the response-scoring dimensions used by AOP (Li et al., 2025), the evaluator assigns each of correctness, relevance, and completeness an integer score in $\{0, 1, 2\}$ (not, partially, and fully satisfied, respectively). For an execution result r, let $c(r)$, $v(r)$, and $p(r)$ denote these three scores. The aggregate score is

$$s(r) = c(r) + v(r) + p(r), \qquad s(r) \in \{0, \ldots, 6\}. \tag{3}$$

Given a labeling threshold γ, an execution result is treated as a positive example if $s(r) \geq \gamma$ and as a negative example otherwise. Unless otherwise specified, we set $\gamma = 6$; consequently, a positive example must receive the maximum score on all three dimensions. The evaluator also provides a brief explanation for each assessment, which is stored as $e^{(t)}$ or $e_i^{(t)}$ together with the corresponding aggregate score.

The goal is to learn from previous trajectories without updating the underlying LLM weights. We therefore maintain an external memory M_t and condition future planning on retrieved and distilled experience:

$$\mathcal{P}_M(Q^{(t)}, \mathcal{D}, A \mid M_{t-1}) = \Big\langle (q_i^{(t)}, a_i'^{(t)}) \Big\rangle_{i=1}^{m_t}. \tag{4}$$

4 Approach

4.1 Overview

Hierarchical Memory Planning consists of two interacting memory modules. The first is a plan-level trajectory memory module that stores whole-task

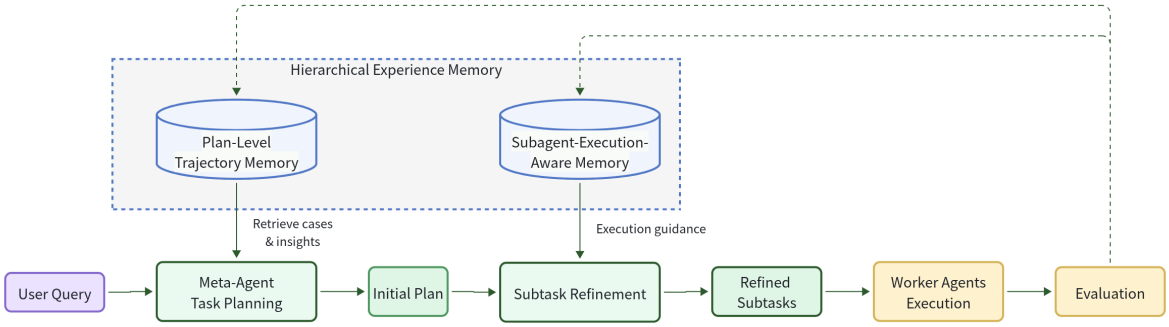


Figure 1: Overview of Hierarchical Memory Planning.

trajectories and supports meta-agent plan generation by retrieving historical cases and deriving compact planning insights, concise plans, and execution evaluations from them. The second is a subagent-execution-aware memory module that refines initially generated subtasks during planning by adding task-execution guidance while preserving the semantic intent of the plan. As illustrated in Figure 1, the pipeline retrieves similar cases to condition the meta-agent, generates an initial plan, refines each planned subtask with relevant subagent-execution-aware memory, executes the refined subtasks with worker agents, and finally scores the task and subtask outputs to update memory.

4.2 Hierarchical Experience Memory

The memory bank $M_t = (M_t^{\text{plan}}, M_t^{\text{sub}}) = (\mathcal{H}_t, \mathcal{W}_t)$ comprises plan-level trajectory memory and subagent-execution-aware memory. M_t^{plan} stores whole-task trajectories and distilled experience for meta-agent planning, whereas M_t^{sub} stores experience for planning-time subtask refinement. Specifically, $\mathcal{H}_t = \{\tau^{(1)}, \ldots, \tau^{(t)}\}$ contains whole-task historical trajectories and $\mathcal{W}_t$ contains worker-specific records of subtasks and their execution trajectories. Each whole-task trajectory records the query, planned subtasks, worker assignments, worker responses, final and subtask-level scores, and the corresponding score explanations.

Hybrid similarity retrieval. For a new query Q, we retrieve relevant cases from $\mathcal{H}_t$ using a weighted combination of sentence-level similarity and task-attribute-aware similarity. We implement $emb(\cdot)$ using `sentence-transformers/all-MiniLM-L6-v2`, following the sentence-embedding retrieval paradigm introduced by Sentence-BERT (Reimers and Gurevych, 2019). Let $\alpha(Q)$ denote task attributes extracted from the query, such as whether it requires multi-hop reasoning, aggregation, comparison, or reasoning over multiple entities. We obtain these attributes with an LLM attribute-extraction prompt, and construct the attribute inventory by applying the same extractor to sampled dataset queries; the inventory covers recurring dataset-specific attributes and can be expanded when new ones are identified. For an inventory of d attributes, we map $\alpha(Q)$ to a binary vector $\mathbf{z}(Q) \in \{0,1\}^d$, whose j-th entry is 1 if the j-th attribute is present and 0 otherwise. We then define attribute similarity as

$$\text{Sim}_{\text{attr}}(\alpha(Q), \alpha(Q_\tau)) = \cos(\mathbf{z}(Q), \mathbf{z}(Q_\tau)) \,. \tag{5}$$

If either binary vector is all zeros, we set the attribute similarity to 0. For a historical trajectory τ, we compute

$$\begin{aligned}\text{Sim}(Q, \tau) = \lambda \cdot \cos(emb(Q), emb(Q_\tau)) \\ + (1-\lambda) \cdot \text{Sim}_{\text{attr}}(\alpha(Q), \alpha(Q_\tau)) \,,\end{aligned} \tag{6}$$

where Q_τ is the query associated with τ and $\lambda \in [0,1]$ controls the contribution of the two similarity sources. We set $\lambda = 0.5$, assigning equal weight to sentence-level and attribute-level similarity. The top-K trajectories are retrieved as

$$\mathcal{R}_K(Q, M_t) = \text{TopK}_{\tau \in \mathcal{H}_t} \text{Sim}(Q, \tau). \tag{7}$$

We use $K = 5$ throughout the reported experiments.

Plan-level insight distillation. The plan-level memory does not only replay retrieved trajectories. It distills reusable planning insights by comparing lower-scoring trajectories with high-scoring trajectories across cases. Let $\mathcal{H}^+ = \{\tau \in \mathcal{H}_t \mid s(\tau) \geq$

$\gamma\}$ denote trajectories whose final-response scores meet the labeling threshold, and let $\mathcal{H}^- = \mathcal{H}_t \setminus \mathcal{H}^+$ denote the remaining lower-scoring trajectories. As defined in Section 3, we use $\gamma = 6$ unless otherwise specified; thus, $\mathcal{H}^+$ contains only trajectories receiving the maximum aggregate score. We update the insight set as $\mathcal{I}_{t+1} = \mathcal{I}_t \cup \text{Distill}_{\text{plan}}(\mathcal{H}^-, \mathcal{H}^+)$, where $\text{Distill}_{\text{plan}}$ extracts lessons about recurring missing dependencies, agent-assignment mistakes, and decomposition patterns associated with higher scores. Each distillation prompt jointly contains five trajectories from $\mathcal{H}^+$ and five from $\mathcal{H}^-$, allowing the LLM to contrast multiple successful and failed trajectories within a shared context rather than pairing trajectories one-to-one. For higher-scoring trajectories, the plan-level memory also preserves concise plans together with evaluations of their final execution outcomes. This cross-case comparison is related to experiential insight extraction from successful and failed trajectories (Zhao et al., 2024); it is intended to avoid overfitting to a single failed trajectory and to produce compact guidance that can be inserted into the meta-agent prompt.

4.3 Plan-Level Memory for Meta-Agent Task Planning

Given a new query Q, the meta-agent receives the query, worker-agent descriptions, and retrieved memory content, producing $\langle (q_i, a'_i) \rangle_{i=1}^{m} = \Phi_{\text{meta}}(Q, \mathcal{D}, \mathcal{R}_K(Q, M_t), \mathcal{I}_t)$, where Φ_{meta} denotes its LLM-based planning policy. The retrieved cases provide concrete analogues, while $\mathcal{I}_t$ provides compact strategic guidance. Conditioning generation on retrieved records is broadly related to retrieval-augmented generation and retrieval-based selection of in-context demonstrations (Lewis et al., 2020; Rubin et al., 2022); here, the retrieved units are evaluated agent trajectories and distilled planning guidance rather than external factual passages. This design is meant to reduce repeated planning errors across related tasks while keeping the prompt shorter than direct trajectory replay.

4.4 Subagent-Execution-Aware Memory for Subtask Refinement

Plan-level memory can improve high-level decomposition, but a raw subtask may still be difficult for a worker to execute. We therefore introduce subagent-execution-aware memory for planning-time subtask refinement. Rather than optimizing the worker agents themselves, this memory stores worker-specific subtasks and their execution trajectories, from which execution-oriented insights can be distilled. Given an initially generated subtask q_i, the refinement module retrieves relevant subagent-execution-aware memory and maps q_i into a refined subtask $\tilde{q}_i$:

$$\tilde{q}_i = \text{Refine}_{\text{sub}}\big(q_i, a'_i, \mathcal{R}_K(q_i, M_t), \mathcal{W}_t\big). \quad (8)$$

Concretely, the refinement module uses the retrieved worker-specific memory to generate a concise insight text w_i that targets execution errors observed in related subtasks. The generation prompt contrasts successful and failed executions and asks the LLM to express the resulting guidance as a short directive beginning with "Note:"; the complete prompt is provided in the supplementary material. We append this memory-derived directive rather than replacing the original subtask: $\tilde{q}_i = q_i \parallel \quad \parallel w_i$, where $\parallel$ denotes string concatenation. This preserves the original subtask semantics while adding execution guidance at its end, and the refined subtask may include: (i) task context from the original query; (ii) local constraints and preconditions; (iii) required output format; (iv) dependency information from preceding or subsequent subtasks; and (v) a subtask insight generated from the relevant records in $\mathcal{W}_t$.

Analogous to plan-level insights, subagent-execution-aware memory is distilled by comparing subtask-level outputs across cases. Let $\mathcal{H}^+_{\text{sub}}$ denote subtask executions with $s_i \geq \gamma$ and let $\mathcal{H}^-_{\text{sub}}$ denote the remaining lower-scoring subtask executions. With the default threshold $\gamma = 6$, $\mathcal{H}^+_{\text{sub}}$ likewise contains only maximum-scoring executions. Each subtask-level distillation prompt similarly contains five positive and five negative execution records for the corresponding worker agent. We update subagent-execution-aware memory as

$$\mathcal{W}_{t+1} = \mathcal{W}_t \cup \text{Distill}_{\text{sub}}\big(\mathcal{H}^-_{\text{sub}}, \mathcal{H}^+_{\text{sub}}\big). \quad (9)$$

The assigned worker then produces $r_i = a'_i(\tilde{q}_i)$.

This module addresses a different failure mode: plan-level memory supports what the meta-agent plans, whereas subagent-execution-aware memory improves how each planned subtask is specified before execution by the assigned worker. In this sense, Hierarchical Memory Planning separates global decomposition adaptation from planning-time subtask refinement, while keeping both mechanisms within a unified hierarchical memory view.

4.5 Memory Update

After task completion, the system evaluates the final answer and intermediate outputs, then updates the memory as $M_{t+1} = \mathrm{Update}(M_t, \tau^{(t)}, \mathrm{Distill}_{\mathrm{plan}}, \mathrm{Distill}_{\mathrm{sub}})$. Following the predefined experimental protocol, each case is executed once to approximate a realistic deployment setting in which repeated attempts on the same instance may be unavailable. During sequential online evaluation, both insight sets are re-distilled after every 20 processed cases using the positive and negative records accumulated at that point. The memory bank therefore improves across cases, rather than through multiple retries on the same case.

4.6 Conceptual Interpretation

Hierarchical Memory Planning can be viewed as retrieval-conditioned textual feedback optimization, related to textual criticism in TextGrad (Yuksekgonul et al., 2024) and episodic verbal feedback in Reflexion (Shinn et al., 2023). Its distillation functions convert task- and subtask-level evaluations into guidance g_t^{plan} and g_t^{sub}, yielding the external-memory update $M_{t+1} = \mathcal{U}(M_t, g_t^{\mathrm{plan}}, g_t^{\mathrm{sub}})$ and a form of hierarchical credit assignment. This interpretation is analogical: the textual guidance is not a numerical derivative and provides no guarantee of monotonic improvement.

5 Experiments

5.1 Experimental Setup

Benchmarks. We evaluate on three multi-step reasoning benchmarks from the Husky agent suite (Kim et al., 2024): DROP (Dua et al., 2019), IIRC (Ferguson et al., 2020), and HuskyQA. DROP and IIRC are evaluated under both the training-derived offline protocol and the sequential online protocol, whereas HuskyQA is evaluated under the online protocol only. For DROP and IIRC, the experience pool is accumulated from the corresponding training data, and held-out evaluation uses 200 instances per dataset.

Benchmark selection. We exclude HuskyQA from the training-derived offline evaluation because it provides no training split from which a disjoint experience pool can be constructed, and building offline memory from its evaluation instances would violate the held-out protocol used for DROP and IIRC. This restriction does not apply to the online protocol, where memory starts empty and is updated sequentially, so we include HuskyQA online only. Accordingly, our offline comparisons should not be read as a reproduction of AOP's complete benchmark suite (Li et al., 2025).

Experience pool. We construct the experience pool from the training data by ranking queries in descending order of word count, retaining the 1,000 longest, and randomly sampling 150 for memory construction. Neither this sampling nor the subsequent trajectory-memory construction uses the reference answers; memory labels are instead derived from the ground-truth-free GPT-4o evaluator defined in Section 3.2.

Metrics. We use accuracy (Acc.) as the primary metric. Following AOP (Li et al., 2025), final benchmark predictions are evaluated by a `gpt-4o-2024-11-20` LLM judge that compares each generated answer with the corresponding ground-truth answer; accuracy is the proportion judged correct. This ground-truth-aware benchmark evaluator is separate from the ground-truth-free GPT-4o evaluator used to label trajectories for memory construction.

Memory construction details. We use the retrieval and memory-labeling procedures defined in Sections 4.2 and 3.2. For the matched trajectory-retrieval baseline, we retrieve the same number of labeled trajectories as for our method ($K = 5$) and use the retrieved records to condition plan generation without insight distillation or subtask refinement. To control for differences in supervision, its trajectories are labeled using the same memory-label evaluator, three-dimensional scoring rubric, and threshold $\gamma = 6$ as Hierarchical Memory Planning.

Systems compared. We compare the full Hierarchical Memory Planning system with a variant that removes subagent insight and with a set of single-agent and multi-agent baselines whose definitions and reported results follow the AOP study (Li et al., 2025). For single-agent systems, we include GPT-4o with the user query provided directly, as well as GPT-4o with Chain-of-Thought prompting (Wei et al., 2022) and Zero-Shot CoT prompting following Kojima et al. (2022), respectively. For multi-agent systems, we include two meta-agent baselines built on GPT-4o: Meta-Agent, which selects one agent for each decomposed subtask, and Meta-Agent: Traversal, which queries

Table 1: Accuracy with a 150-example training-derived experience pool. †Results quoted from Li et al. (2025); DROP and IIRC values are from its appendix.

Method	DROP	IIRC
GPT-4o†	23.0	33.0
CoT†	26.0	35.0
Zero-Shot CoT†	24.5	33.0
Meta-Agent (no memory)	25.5	32.5
Meta-Agent: Traversal†	27.5	34.0
ReAct†	29.0	36.0
HUSKY†	28.0	36.5
AOP†	34.0	41.5
Our method (w/o subagent insight)	35.5	41.0
Our method (w/o history)	36.0	42.0
Our method (full)	**37.0**	**43.5**

Table 2: Accuracy under a matched trajectory-retrieval protocol ($K = 5$; shared memory-label evaluator, rubric, and threshold $\gamma = 6$).

Method	DROP	IIRC
Matched trajectory retrieval	32.5	36.0
Our method (full)	**37.0**	**43.5**

Table 3: Online accuracy with memory initialized empty and updated sequentially.

Method	DROP	IIRC	HuskyQA
Meta-Agent (no memory)	25.5	32.5	30.0
Our method (w/o history)	36.0	35.5	27.7
Our method (w/o subagent insight)	38.5	40.5	39.7
Our method (full)	**42.5**	**46.5**	**42.5**

all agents for each subtask and selects the best response. Meta-Agent follows the agent configuration and prompting protocol reported by AOP (Li et al., 2025) and serves as the no-memory baseline in both offline and online evaluation. We also compare against representative decomposition-based agent systems, including ReAct (Yao et al., 2023), HUSKY (Kim et al., 2024), and AOP (Li et al., 2025). As a memory-specific comparator, we include the matched trajectory-retrieval baseline described above.

Involving agents. Following the agent configuration used in AOP (Li et al., 2025) and HUSKY (Kim et al., 2024), the system comprises a meta-agent and four specialized workers: a code agent (Python for precise computation), a math agent (step-by-step reasoning), a search agent (Bing Search API), and a commonsense agent. The meta-agent decomposes each query into subtasks and assigns each to the most suitable worker. All agents use `gpt-4o-2024-11-20` as the backbone with decoding temperature 0, following the agent descriptions and prompt design of AOP. The complete plan-generation and subtask-guidance templates are provided in Appendix A.

5.2 Main Results

Table 1 shows that Hierarchical Memory Planning with subagent insight obtains the highest recorded accuracy on both DROP and IIRC when 150 training examples are used as the experience pool. Relative to the Meta-Agent no-memory baseline, Hierarchical Memory Planning increases accuracy from 25.5 to 37.0 on DROP and from 32.5 to 43.5 on IIRC, corresponding to differences of 11.5 and 11.0 points, respectively. The ablations without subagent insight and without history obtain 35.5/41.0 and 36.0/42.0 on DROP/IIRC, respectively. These results indicate that the complete configuration performs best among the evaluated variants under the training-derived experience-pool protocol.

Table 2 reports the matched trajectory-retrieval baseline under the shared retrieval and labeling protocol. The baseline obtains 32.5 accuracy on DROP and 36.0 on IIRC, whereas Hierarchical Memory Planning obtains 37.0 and 43.5, respectively. These results correspond to improvements of 4.5 points on DROP and 7.5 points on IIRC.

Table 3 reports the online-evaluation setting, where memory is initialized from an empty state and updated sequentially as evaluation proceeds. Relative to the Meta-Agent no-memory baseline, Hierarchical Memory Planning increases accuracy from 25.5 to 42.5 on DROP, from 32.5 to 46.5 on IIRC, and from 30.0 to 42.5 on HuskyQA, corresponding to differences of 17.0, 14.0, and 12.5 points, respectively. On DROP and IIRC, the online-evaluation results also exceed those obtained with the 150-example training-derived pool, suggesting that accumulating experience on the evaluation distribution can provide stronger task-adaptive guidance. This cross-protocol comparison should be interpreted carefully because online evaluation intentionally allows learning from the test-task stream; no corresponding offline comparison is available for HuskyQA. We also observe a failure case: on HuskyQA, the *w/o history* configuration, which applies subagent-execution-aware guidance without plan-level trajectory memory, reaches only 27.7 and falls below the 30.0 no-memory baseline. This indicates that subagent guidance is not uncon-

Table 4: Accuracy by insight-label threshold γ; scores $\geq \gamma$ are labeled positive.

γ	DROP	IIRC
3	30.5	40.5
6	**37.0**	**43.5**

Table 5: Effect of experience-pool size.

Pool size	DROP	IIRC
50	30.5	39.5
100	31.5	40.0
150	**37.0**	**43.5**

ditionally beneficial: when the underlying plan is weak, refining individual subtasks in isolation can reinforce a poor decomposition and reduce overall performance, whereas the full configuration, which applies subagent guidance on top of plan-level memory, recovers and improves to 42.5. A more detailed failure analysis of when subagent guidance helps or hurts is left to future work.

5.3 Sensitivity Analyses

Table 4 shows that the labeling threshold affects performance: threshold 6 outperforms threshold 3 on DROP and slightly improves IIRC, possibly because a stricter positive/negative split reduces noisy or weakly relevant insights. This interpretation requires further qualitative validation of the generated insights.

Table 5 shows that increasing the experience pool from 50 to 150 examples improves performance, consistent with the hierarchical memory benefiting from more trajectories to retrieve cases and distill insights; whether performance saturates beyond 150 examples remains unclear.

Table 6 reports the average token usage per case. Relative to the Meta-Agent no-memory baseline, the plan-memory-only configuration increases average prompt and completion usage from 1,200 to 8,745 tokens and from 290 to 1,477 tokens, respectively. The full memory configuration has the highest average usage, at 12,138 prompt tokens and 1,919 completion tokens per case, reflecting the additional retrieved trajectories and memory-derived guidance injected into the context.

5.4 Preliminary Qualitative Analysis of Generated Subtask-Guidance Directives

We conducted a preliminary qualitative thematic analysis of the generated subtask-guidance directives, hereafter referred to as subtask guidance. We manually consolidated near-duplicate instructions before interpreting recurring themes, while allowing each instruction to receive multiple thematic labels because a single directive can simultaneously specify content, reasoning, and output constraints. This analysis is descriptive; we do not report category frequencies or inter-annotator agreement. Search-agent directives primarily emphasize preserving query-specific entities, key verbs, dates, locations, and contextual qualifiers, together with focused retrieval, evidence verification, and direct answer extraction. A representative search-agent directive is:

> "Note: When generating search queries, carefully analyze the user's question to identify and include specific details, key verbs, dates, locations, and constraints to ensure the query accurately reflects the original intent."

Commonsense-agent directives mainly encourage context-aware logical analysis, factual verification, and avoidance of unsupported inferences based on names or other superficial cues. Math-agent directives focus on explicit intermediate calculations, temporal and comparative reasoning, data-sufficiency checks, and structured answer presentation. Code-agent directives emphasize requirement parsing and procedural completeness.

Across agents, the themes can be summarized as input fidelity, task-specific processing, evidence validation, uncertainty control, and output alignment. Although near-duplicate instructions were consolidated before analysis, some remaining directives may still be redundant or overlap substantially in function. These observations characterize the directive content but do not by themselves establish which themes cause the performance differences reported above.

Table 6: Average prompt and completion tokens per case.

Method	Prompt	Completion
Meta-Agent (no memory)	1,200	290
Our method (plan memory only)	8,745	1,477
Our method (full memory)	12,138	1,919

5.5 Discussion

On DROP and IIRC, the online setting yields larger gains than the fixed training-derived pool, but it permits sequential adaptation to the test-task stream and is therefore not directly equivalent to held-out evaluation; HuskyQA is evaluated online only

for the same lack of a disjoint offline pool. The memory-based configurations also consume more tokens per case (Table 6), so their accuracy gains should be weighed against the additional inference cost.

6 Conclusion

We presented Hierarchical Memory Planning, a self-evolving multi-agent task planning framework with separate plan-level and subagent-execution-aware memory. The plan-level memory conditions future decomposition on compact insights, plans, and evaluations retrieved from similar historical cases, while the subagent-execution-aware memory refines each subtask with guidance distilled from the corresponding subagent's past executions. Future work could evaluate Hierarchical Memory Planning across a broader range of tasks, domains, and multi-agent configurations.

7 Limitations

Our evaluation is limited to three multi-step question-answering benchmarks from the Husky suite (offline: DROP, IIRC; online: DROP, IIRC, HuskyQA), a relatively narrow setting that does not establish whether the results transfer to other domains, interactive environments, longer-horizon workflows, or different agent configurations. Further evaluation across more diverse reasoning, tool-use, and environment-interaction scenarios is needed to assess the generalizability of the proposed memory framework. Our ablations also reveal a failure mode in which subagent-execution-aware guidance, applied without plan-level memory, underperforms the no-memory baseline on HuskyQA; a systematic analysis of when such guidance helps or harms is left to future work.

References

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

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378.

James Ferguson, Matt Gardner, Hannaneh Hajishirzi, Tushar Khot, and Pradeep Dasigi. 2020. Iirc: A dataset of incomplete information reading comprehension questions. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pages 1137–1147.

Yanjiang Guo, Yen-Jen Wang, Lihan Zha, and Jianyu Chen. 2024. Doremi: Grounding language model by detecting and recovering from plan-execution misalignment. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12124–12131. IEEE.

Zhezheng Hao, Tianfu Wang, Huanshuo Dong, Ziyan Liu, Hong Wang, Xiankun Lin, Qiang Lin, Can Wang, Hande Dong, and Jiawei Chen. 2026. Evolve as a team: Collaborative self-evolution for llm-based multi-agent systems. *arXiv preprint arXiv:2605.29790*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pages 23247–23275.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*.

Joongwon Kim, Bhargavi Paranjape, Tushar Khot, and Hannaneh Hajishirzi. 2024. Husky: A unified, open-source language agent for multi-step reasoning. *arXiv preprint arXiv:2406.06469*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. 2025. Agent-oriented planning in multi-agent systems. In *International Conference on Learning Representations*, volume 2025, pages 19495–19517.

Xuechen Liang, Meiling Tao, Yinghui Xia, Jianhui Wang, Kun Li, Yijin Wang, Jingsong Yang, Tianyu Shi, Yuantao Wang, Miao Zhang, and 1 others. 2025. Mars: Memory-enhanced agents with reflective self-improvement. *arXiv preprint arXiv:2503.19271*.

Minhua Lin, Zhiwei Zhang, Hanqing Lu, Hui Liu, Xianfeng Tang, Qi He, Xiang Zhang, and Suhang Wang. 2026. Memma: Coordinating the memory cycle through multi-agent reasoning and in-situ self-evolution. *arXiv preprint arXiv:2603.18718*.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Mihir Parmar, Xin Liu, Palash Goyal, Yanfei Chen, Long Le, Swaroop Mishra, Hossein Mobahi, Jindong Gu, Zifeng Wang, Hootan Nakhost, and 1 others. 2025. Plangen: A multi-agent framework for generating planning and reasoning trajectories for complex problem solving. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20651–20677.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pages 3982–3992.

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 2655–2671.

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

Jun Wang. Memento 2: Learning by stateful reflective memory, 2026. *URL https://arxiv.org/abs/2512.22716*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

Xinle Wu, Rui Zhang, Mustafa Anis Hussain, and Yao Lu. 2026. Towards autonomous memory agents. *Preprint*, arXiv:2602.22406.

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*.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Cong Zhang, Xin Deik Goh, Dexun Li, Hao Zhang, and Yong Liu. 2025. Planning with multi-constraints via collaborative language agents. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 10054–10082.

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

Tuo Zhang, Alin-Ionut Popa, Yan Xu, Rui Song, and Dimitrios Dimitriadis. 2026b. Pivot: Bridging planning and execution in llm agents via trajectory refinement. *arXiv preprint arXiv:2605.11225*.

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

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and 1 others. Memento: Fine-tuning llm agents without fine-tuning llms, 2025. *URL https://arxiv.org/abs/2508.16153*.

A Supplementary Material: Prompt Templates

This supplementary material reports the prompt templates used for plan generation and subtask-guidance generation. Placeholders enclosed in braces are replaced with instance-specific content at inference time.

A.1 Plan-Generation Prompt

```
You are a planning agent responsible for decomposing a user query into a coherent sequence of
    subtasks and assigning the most suitable primary agent to each subtask.

Your goal is to produce an efficient and accurate execution plan. Each subtask should preserve the
    key entities, numerical values, temporal constraints, and logical requirements from the
    original query. Dependencies between subtasks must be explicitly specified so that each
    subtask can use information produced by its prerequisite subtasks.

Available agents:
- code_agent: Generates Python code for precise computations, including arithmetic, date
    calculations, and processing long lists of values.
- math_agent: Solves mathematical questions through step-by-step reasoning.
- search_agent: Generates Bing Search queries to retrieve external information relevant to the task
    .
- commonsense_agent: Solves questions using commonsense knowledge, logical reasoning, and basic
    arithmetic.

Output requirements:
- Output only a valid JSON list. Do not include Markdown code fences, explanations, or any text
    outside the JSON list.
- Each subtask must follow this schema:
[
  {
    "task": "A self-contained subtask description",
    "id": 1,
    "name_1": "name_of_the_primary_agent",
    "reason": "A concise justification for assigning this agent",
    "dep": []
  }
]
- `task` is the subtask description that will be directly provided to the assigned agent.
- `id` is a unique positive integer, ordered according to the execution sequence.
- `name_1` must be one of: `code_agent`, `math_agent`, `search_agent`, or `commonsense_agent`.
- `reason` must explain why the selected agent is suitable.
- `dep` is a list of IDs of prerequisite subtasks. Use an empty list when the subtask has no
    dependency.
- Do not create redundant subtasks.
- If the query can be solved by one agent, output exactly one subtask.

Here is an example:
{
  "query": "If a plane can carry 300 passengers and decides to fly from China to Indonesia, how
    many full flights are needed to transport 1% of the population of China to Indonesia?",
  "output": [
    {
      "task": "Determine the population of China.",
      "id": 1,
      "name_1": "search_agent",
      "reason": "The search_agent is appropriate for retrieving the population of China.",
      "dep": []
    },
    {
      "task": "Calculate 1% of the population of China using the retrieved population value.",
      "id": 2,
      "name_1": "math_agent",
      "reason": "The math_agent can perform the percentage calculation step by step.",
      "dep": [1]
    },
    {
      "task": "Calculate the number of full flights required to transport 1% of China's population
    when each plane carries 300 passengers.",
      "id": 3,
```

```
      "name_1": "math_agent",
      "reason": "The math_agent can compute the division and apply the required rounding logic.",
      "dep": [2]
    }
  ]
}

## Your Own Past Successes (Execution Patterns)
Here are examples of successful execution processes you have previously used on similar tasks. Pay
    special attention to their step-by-step procedures, decomposition strategies, dependency
    structures, and ways of handling obstacles:

{memory_few_shots}

## Key Insights from Related Tasks
The following insights were gathered from related task executions. Use them to improve planning
    accuracy, but do not copy irrelevant details or introduce information that is not supported by
     the current user query:

{insights}

Given the user's query and the available references above, output the task plan directly in the
    required JSON-list format.
```

A.2 Subtask-Guidance Generation Prompt

```
**Role:** You are an expert Prompt Engineer and AI Behavior Analyst specializing in error diagnosis
     and query optimization.

**Task:** I will provide you with a description of an AI Agent's capabilities, along with a set of
    successful interactions and a set of failed interactions. Your goal is to contrast these
    examples, diagnose the root cause of the failures, and generate a concise "meta-instruction" (
    Wrap Text).

This Wrap Text must be a short, highly specific directive that will be appended to the end of
    future user queries (e.g., inside parentheses) to preemptively correct the agent's behavior
    and guide it to the correct output.

**Example of the desired Wrap Text mechanism:**
* *Original Query:* Identify the date when Abaza-Pasha crossed the Dniester
* *Wrapped Query:* Identify the date when Abaza-Pasha crossed the Dniester. **(Note: To ensure
    search accuracy, include the original phrasing and specific verbs in the search query, do not
    over-simplify.)** -> *The bold part is the Wrap Text you must generate.*

**Inputs:**
* **Agent Capabilities:** {capabilities}
* **Successful Examples:**
   {Successful_Examples}
* **Failed Examples:**
   {Failed_Examples}

**Instructions:**
1. **Analyze the Failures:** Compare the successful and failed examples. Identify the specific
    operational misstep the agent is making in the failed cases (e.g., over-simplifying search
    terms, ignoring constraints, failing to reason step-by-step).
2. **Formulate the Wrap Text:** Create a single, concise meta-instruction starting with "(Note:
    ...)" that directly addresses the root cause of the failure. It must be universally applicable
     to similar queries to prevent the error.
3. **Output Format:** You must strictly follow this output format:

### reason:
[Detailed analysis contrasting the successful and failed cases, explaining the exact point of
    failure.]

### wrap text:
Note: [Your generated meta-instruction here]
```