LEARNING TO CONTINUALLY LEARN VIA META-LEARNING AGENTIC MEMORY DESIGNS

Yiming Xiong
University of British Columbia

Shengran Hu
University of British Columbia
Vector Institute

Jeff Clune
University of British Columbia
Vector Institute
Canada CIFAR AI Chair

ABSTRACT

The statelessness of foundation models bottlenecks agentic systems' ability to continually learn, a core capability for long-horizon reasoning and adaptation. To address this limitation, agentic systems commonly incorporate memory modules to retain and reuse past experience, aiming for continual learning during test time. However, most existing memory designs are human-crafted and fixed, which limits their ability to adapt to the diversity and non-stationarity of real-world tasks. In this paper, we introduce ALMA (**A**utomated meta-**L**earning of **M**emory designs for **A**gentic systems), a framework that meta-learns memory designs to replace hand-engineered memory designs, therefore minimizing human effort and enabling agentic systems to be continual learners across diverse domains. Our approach employs a Meta Agent that searches over memory designs expressed as executable code in an open-ended manner, theoretically allowing the discovery of arbitrary memory designs, including database schemas as well as their retrieval and update mechanisms. Extensive experiments across four sequential decision-making domains demonstrate that the learned memory designs enable more effective and efficient learning from experience than state-of-the-art human-crafted memory designs on all benchmarks. When developed and deployed safely, ALMA represents a step toward self-improving AI systems that learn to be adaptive, continual learners. All code is open-sourced at `https://github.com/zksha/alma.git`.

1 INTRODUCTION

Agentic systems powered by Foundation Models (FMs) have enabled autonomous decision-making and task execution across a wide range of domains (Yao et al., 2023; Wang et al., 2023; Yang et al., 2024). However, the stateless nature of FMs during inference challenges agentic systems to accumulate experience and continually learn from past interactions, causing them to repeatedly solve tasks from scratch and limiting their ability to improve over time (Child et al., 2019; Bulatov et al., 2022). Memory addresses this limitation by enabling agentic systems to continually store and reuse past experiences (Zhong et al., 2024; Packer et al., 2024). This allows accumulated experience to inform decision-making in future tasks, aiming for continual learning over time.

However, memory design, the architectural specification that determines how memories are represented, stored, retrieved, and updated, is still predominantly handcrafted by humans. Diverse domains require distinct memory designs to leverage unique aspects of experience, resulting in human researchers manually tailoring a wide array of designs to specific tasks (Zhang et al., 2024b; Hu et al., 2026). For example, in conversational agents, memory focus on retaining facts about the user, such as preferences and personal information (Chhikara et al., 2025; Rasmussen et al., 2025). In contrast, for strategic games, memory should extract skills and strategies from previous interactions rather than details that may change across episodes (Tang et al., 2025; Ouyang et al., 2025). Therefore, manually identifying optimal memory design for each domain is difficult and labor-intensive.

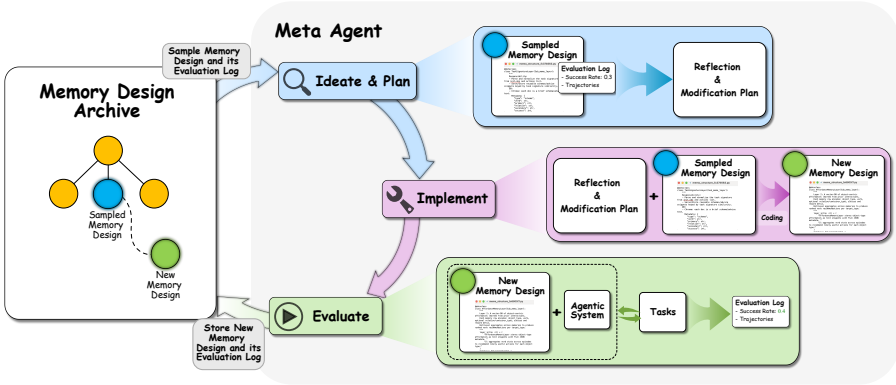

Figure 1: Open-ended Exploration Process of ALMA. The Meta Agent first ideates and proposes a plan by reflecting on the code and evaluation logs of the sampled memory design. It then implements the plan by programming the new design in code. Finally, it verifies the correctness of the new memory design and evaluates it with an agentic system. The evaluated memory design is subsequently added to the memory design archive for future sampling.

A recurring theme in machine learning history is that handcrafted components in AI systems are eventually replaced by learned, more effective ones (Clune, 2020; Hutter et al., 2019; Sutton, 2019). A classic example of this paradigm is the transition from hand-designed features to learned representations in computer vision (Krizhevsky et al., 2012). More recently, the trend of "learning to learn" has expanded to learning neural architectures (Zoph & Le, 2017), optimization algorithms (Andrychowicz et al., 2016), training environments (Wang et al., 2019; Zhang et al., 2024a; Faldor et al., 2025), and designs for agentic systems (Hu et al., 2025; Zhang et al., 2025e). In this paper, we investigate whether agentic systems can learn to continually learn by automating the design of their memory components. This enables memory designs to be specialized for diverse domains without relying on manual engineering.

We propose ALMA (**A**utomated meta-**L**earning of **M**emory designs for **A**gentic systems), a framework that adopts a Meta Agent to explore novel memory designs for agentic systems through open-ended exploration. We use code as the search space, giving ALMA the theoretical potential to build any memory design. As shown in fig. 1, the Meta Agent samples previously explored memory designs from an archive, where all explored memory designs and their evaluation logs are stored. Then the Meta Agent reflects on the sampled memory designs to generate new ideas and plans, which are implemented in code. The new designs are subsequently validated and evaluated, with the resulting logs added back to the archive to guide future sampling.

We evaluate our method across four sequential decision-making domains, including ALFWorld (Shridhar et al., 2021), TextWorld (Côté et al., 2019), Baba Is AI (Cloos et al., 2025), and MiniHack (Samvelyan et al., 2021). These domains serve as testbeds for evaluating an agent's capability to continually learn from experience. In these domains, agents benefit from storing and reusing their experiences to acquire effective strategies and useful information that is absent from the FMs' pre-trained knowledge. The results show that ALMA can discover memory designs that are tailored for the needs of different domains, consistently surpassing state-of-the-art human-designed memory baselines (Table 1). Beyond superior performance, our learned memory designs are more cost-efficient than most human-designed memory baselines (Appendix C.1). Furthermore, the learned memory designs scale performance more effectively with memory size and learn faster when facing task distribution shifts (Figure 4). Overall, the memory designs learned by ALMA demonstrate a superior ability to help agentic systems continually learn, paving the way towards developing lifelong learning agents in dynamic environments.

2 Related Work

Memory for Agentic Systems. Memory enables agentic systems to continually learn despite the statelessness of FMs (Hu et al., 2026; Zhang et al., 2025i). Existing work has categorized memory mechanisms into token-level memory, parametric memory, and latent memory (Hu et al., 2026). Token-level memory defines databases and update rules to store information and extract insights

from interaction trajectories between agents and environments. When addressing a new task, the system retrieves relevant experiences via defined procedures, which are then appended to the prompts of agentic systems as context (Chhikara et al., 2025; Rasmussen et al., 2025; Nan et al., 2025; Zhang et al., 2025a; Ouyang et al., 2025; Suzgun et al., 2025; Zhao et al., 2024). For example, G-Memory (Zhang et al., 2025a) extracts experiences and stores them in a graph-based database, and retrieves relevant experiences for new tasks via graph traversal. Reasoning Bank (Ouyang et al., 2025) performs task-wise experience generation during memory updates, while Dynamic Cheatsheet (Suzgun et al., 2025) includes accumulation of insights across trajectories. Current works also explore training models to select from predefined operations for storing and reusing experience in memory (Zhou et al., 2025; Yan et al., 2026; Liang et al., 2026; Wang et al., 2025). Other categories of memory include parametric and latent memory, which encode experience implicitly within the model's weights (Qin et al., 2024; Zhang et al., 2025g) or latent hidden states (Wang et al., 2024a; Zhang et al., 2025b; Zou et al., 2025) rather than relying on explicit text-based retrieval. The "Learning to learn" paradigm can theoretically support the learning of all categories of memory. In this paper, we focus on token-level memory, as it allows for faster evaluation without additional model training and provides more interpretable, transferable memory representations.

AI-generating algorithms and the "Learning to learn" paradigm. AI-generating algorithms (Clune, 2020) and automated machine learning (Hutter et al., 2019) aim to replace hand-engineered components with automatically learned ones. The approach has three key pillars: (1) meta-learning architectures, (2) meta-learning learning algorithms, and (3) the automatic generation of learning environments (Clune, 2020). Neural Architecture Search (Elsken et al., 2019; Lu et al., 2019; Hu et al., 2023) primarily contributes to the first pillar by automating the discovery of neural architectures. Research in MAML (Finn et al., 2017), Meta-RL (Wang et al., 2017; Duan et al., 2016; Norman & Clune, 2024), and recent Automated Algorithm Design (Liu et al., 2024; 2026) has explored the second pillar by focusing on "Learning to learn", developing algorithms that enable agents to adapt their learning behavior across tasks (Beaulieu et al., 2020; Lu et al., 2024a). The third pillar includes approaches such as POET (Wang et al., 2019; 2020), as well as more recent work on dynamic environment generation with agentic systems (Aki et al., 2024; Guo et al., 2025; Zhang et al., 2024a). Our work is more related to the second pillar, as ALMA learns how memory designs can better enable agentic systems to continually learn. Furthermore, recent AI-generating algorithms and automated machine learning approaches have leveraged FMs to explore new components in AI systems through code generation (Hu et al., 2025; Zhang et al., 2025e; Faldor et al., 2025; Yamada et al., 2025; Lu et al., 2024b;a; 2025b; Lehman et al., 2024). Similarly, we adopt a Meta Agent powered by FMs to propose novel memory designs by programming in code.

Automated Design of Agentic Systems. Previous work has studied the automated design of agentic systems to learn better components in agentic systems (Hu et al., 2025; Zhang et al., 2025f; Zhou et al., 2024; Yin et al., 2025; Zhuge et al., 2024; Rosser & Foerster, 2025; Zhang et al., 2025c; Shang et al., 2025; Zhang et al., 2025h;c; Ye et al., 2025; Zhang et al., 2025e). Since recent works such as ADAS (Hu et al., 2025) employ code as the search space for agentic system designs, they theoretically enable the learning of the memory component. For example, AgentSquare (Shang et al., 2025) attempts to co-learn memory designs along with other components like tools in agentic systems. However, while existing works evaluate only the one-shot performance of agentic systems, ALMA explicitly optimizes for a memory design's capability to facilitate continual learning from past experience, enabling the discovery of more effective memory designs. A concurrent work to ALMA also explores an approach for learning memory designs (Zhang et al., 2025d). However, it relies on the initialization with many existing handcrafted memory designs and greedy selection of top-performing designs, which limits open-ended exploration. In contrast, ALMA adopts open-ended exploration and learns from scratch. Open-ended exploration is shown to be important to search for high-performance structure (Lehman & Stanley, 2008; 2011a;b; Conti et al., 2018; Stanley, 2019). Therefore, incorporating open-ended exploration into searching for optimal memory designs presents a promising avenue. Our ablation studies show that indeed open-ended exploration can learn better memory designs than greedy-selection-based optimization (Appendix C.3).

3 Learning of Memory Designs

We propose ALMA, a framework for learning memory designs within a code-based search space via open-ended exploration, discovering memory designs that effectively collect, summarize, and reuse experience. Specifically, a Meta Agent (Hu et al., 2025) iteratively proposes a new candidate

memory design by reflecting on the results of previously discovered designs, implements the new design through code generation and debugging, and evaluates its performance by integrating it into an agentic system. The pseudocode of our learning process is provided in Appendix A.6.

3.1 Search Space for Memory Designs

The search space defines all memory designs that can be represented and discovered. In this paper, we use code as the search space, thus theoretically allowing all possible memory designs to be discovered due to the Turing completeness of many programming languages, i.e., Python in our case. Representing memory designs in code also enables interpretability and allows FMs to leverage their prior knowledge acquired during pretraining about agentic systems and coding (Hu et al., 2025). Due to the vastness of the code-based search space, it is inefficient for the Meta Agent to construct all basic functions of a memory design from scratch. We therefore provide the Meta Agent a simple abstraction that facilitates exploration while preserving flexibility. Our abstraction design is inspired by common patterns observed in existing handcrafted memory designs (Nan et al., 2025; Chhikara et al., 2025; Ouyang et al., 2025; Zhang et al., 2025a; Tang et al., 2025; Suzgun et al., 2025). The abstraction allows the memory module to interact with the agentic system through two primary interfaces: `general_update()` and `general_retrieve()`. After collecting new interactions with the environment, the agent calls `general_update()` to extract useful experience into memory. When faced with a new task, the agent calls `general_retrieve()` to access relevant experience. Internally, each interface can coordinate multiple sub-modules. Each sub-module implements its `update()` and `retrieve()` logic, and optionally maintains its own database when needed. The output of one sub-module can serve as input for subsequent sub-modules, enabling hierarchical and modular memory designs. For example, ReasoningBank (Ouyang et al., 2025) employs a vector-database sub-module to store task descriptions, along with an additional sub-module that extracts experience learned from individual tasks with FMs. Similarly, G-memory (Zhang et al., 2025a) incorporates a graph-database sub-module to model linkages between extracted insights and a vector-database sub-module to store task descriptions. We implement the described abstraction in Python Abstract Classes, and the code is available in Appendix A.1.

3.2 Evaluation of Memory Designs

Based on existing work (Nan et al., 2025; Chhikara et al., 2025; Ouyang et al., 2025; Zhang et al., 2025a; Tang et al., 2025; Suzgun et al., 2025; Hu et al., 2026; Wang et al., 2024b; Zhong et al., 2024; Packer et al., 2024; Fang et al., 2026; Zhao et al., 2024), we summarize that memory components in agentic systems generally operate in two phases.

Memory Collection Phase. The goal of this phase is not task success, but rather collecting knowledge and updating memory for later use in the Deployment Phase. There are two common ways to obtain raw trajectories for updating memory at this stage: (1) leveraging existing agent trajectories from available datasets (Zhong et al., 2024; Chhikara et al., 2025), or (2) starting from an empty memory state and collecting trajectories by running the agent on tasks (Fang et al., 2026; Tang et al., 2025). This phase typically does not involve retrieving from memory (Zhong et al., 2024; Chhikara et al., 2025; Wang et al., 2024b; Packer et al., 2024; Rasmussen et al., 2025).

Deployment Phase. This is the main stage where the agentic system is deployed to applications and expected to achieve task success. Given the memory collected after the Memory Collection Phase, the agentic system retrieves from memory for each incoming task and attempts to solve it. This stage can operate in two modes: **dynamic**, which updates memory with newly collected trajectories from incoming tasks (Zhang et al., 2025a; Ouyang et al., 2025), or **static**, which keeps memory fixed (Tang et al., 2025; Rasmussen et al., 2025).

We evaluate each memory design by first running the Memory Collection Phase and then recording the success rate during the Deployment Phase, using an identical, fixed agentic system, with the memory design being evaluated. The two modes in the Deployment Phase serve different purposes: static mode assesses how effectively the agent leverages a fixed memory to solve new tasks, while dynamic mode measures how well a memory design adapts to a new task distribution through dynamic updates and retrieval. When learning memory designs, we use only static mode to reduce variance during evaluations. The `general_update` and `general_retrieve` modules contribute to the performance of the memory design through memory collection and knowledge retrieval, respectively. When testing the best-learned memory design, we evaluate under both modes. More details on evaluations are available in Appendix A.2.

3.3 OPEN-ENDED EXPLORATION

Similar to ADAS (Hu et al., 2025), we propose a framework for open-ended exploration of memory designs with a Meta Agent programming novel memory designs in code. The algorithm proceeds as follows: (1) An archive is initialized with Python abstract classes as an empty memory design template. (2) The Meta Agent samples previously discovered designs from the archive, reflects on their performance outcomes, and proposes new memory designs. (3) The newly proposed designs are evaluated on the target domain. If errors occur during evaluation, the Meta Agent performs self-reflection to refine the design, repeating this process up to three times if necessary. (4) Finally, the memory design is added to the archive along with its evaluation results and logs, and the process continues with the updated archive until the maximum number of iterations is reached.

Sampling Memory Designs. We sample memory designs from the archive to serve as stepping stones for the exploration of new memory designs at each step. Similar to DGM (Zhang et al., 2025e), each design in the archive is assigned a sampling probability roughly proportional to its success rate and inversely proportional to the number of times it is sampled. This prioritizes designs that perform well but have been sampled less frequently, enabling a balance between refining successful designs and exploring underexplored candidates. All designs maintain non-zero sampling probabilities, keeping all potential improvements reachable. Details are provided in Appendix A.4.

Proposing New Memory Design by the Meta Agent. After sampling previous memory designs from the archive, the Meta Agent reflects on their evaluation results and proposes new memory designs by implementing them in code. Specifically, the Meta Agent first proposes ideas, then plans for the new designs through analyzing the sampled designs along with their success rate and interaction logs. Based on the plans, the Meta Agent implements the new memory designs in code and performs trial runs. Runtime errors during the trial runs trigger debugging reflections to refine the implementations. Finally, the new memory designs are evaluated. Their evaluation logs are added to the memory design archive for future sampling. Prompts of Meta Agent are in Appendix A.5.

4 EXPERIMENTS

4.1 EXPERIMENT SETUP

We perform ALMA on four sequential decision-making benchmarks (Section 4.2). For each benchmark, the dataset is divided into a learning set and a testing set, which are used for memory design learning and for the testing of the learned memory designs to ensure testing on unseen data. Both sets are further split evenly, with the first half used for the Memory Collection Phase and the second half for the Deployment Phase (Section 3.2). To reduce variance, we execute the Deployment Phase three times for learning and testing, and report the average success rate with the standard error. For clarity, in the following sections, "success rate" refers to the average success rate over three runs.

At each learning step, we sample up to five memory designs from the archive without replacement. We expect the choice between sampling with or without replacement to have minimal impact (Appendix B.3). We run ALMA for 11 learning steps to discover 43 memory designs.

During the learning of memory designs, `GPT-5` is used in the Meta Agent, while `GPT-5-nano` is used in the fixed agentic system for evaluating memory designs. While the Meta Agent can choose to use any model (or train models) as FMs within memory designs, we provide `GPT-4o-mini`, `GPT-4.1`, and `text-embedding-3-small` as tools for the Meta Agent to explore workflows such as extracting insights or computing semantic similarity. More details on the learning process setup are available in Appendix B.3. During the testing of the learned memory designs, we test under two agentic system settings: the first employs `GPT-5-nano`, matching the learning setup. The second replaces it with a more capable FM, `GPT-5-mini`, to assess the transferability of memory designs to stronger agentic systems (Table 1). We use `GPT-4o-mini` and `text-embedding-3-small` as FMs in all memory designs during testing to ensure fair comparison. More details are available in Appendix B.4.

4.2 BENCHMARKS

We adopt four sequential decision-making benchmarks with varying levels of difficulty (Paglieri et al., 2025). In all benchmarks, agentic systems perceive the environment through text and inter-

Table 1: Comparison of success rates between the learned memory designs and baseline memory designs across environments. Top: Learning and testing the memory designs with `GPT-5-nano` as the FM in the agentic system. **Bottom**: Transferring the memory designs to test with `GPT-5-mini` in the agentic system. In both cases, learned memory designs outperform human-designed baselines, demonstrating that they are more effective than human-designed memory and generalize robustly across different FMs. Results are reported as mean ± standard error in percentages, calculated over three runs of the Deployment Phase. The relative gains over the no-memory baseline are color-coded.

Memory Designs	ALFWorld	Textworld	Baba Is AI	MiniHack	Overall Avg.
GPT-5-nano					
No Memory	2.9 ± 0.8	5.4 ± 1.0	9.5 ± 2.4	6.7 ± 1.7	6.1
Manual Memory Designs					
Trajectory Retrieval	5.2 ± 1.9 +2.3	2.7 ± 0.9 -2.7	**19.0** ± 6.3 +9.5	7.5 ± 1.4 +0.8	8.6 +2.5
Reasoning Bank	5.2 ± 1.3 +2.3	5.3 ± 0.8 -0.1	9.5 ± 2.4 +0.0	9.8 ± 1.7 +3.7	7.5 +1.4
Dynamic Cheatsheet	5.7 ± 0.8 +2.8	4.3 ± 1.0 -1.1	9.5 ± 4.8 +0.0	9.2 ± 0.8 +2.5	7.2 +1.1
G-Memory	7.6 ± 0.5 +4.7	2.1 ± 0.8 -3.3	14.3 ± 4.1 +4.8	6.8 ± 2.3 +0.1	7.7 +1.6
Learned Memory Designs					
ALMA (Ours)	**12.4** ± 0.5 +9.5	**6.2** ± 1.7 +0.8	**19.0** ± 2.4 +9.5	**11.7** ± 2.2 +5.0	**12.3** +6.2
GPT-5-mini					
No Memory	67.6 ± 1.0	60.5 ± 4.4	21.4 ± 0.0	15.0 ± 1.4	41.1
Manual Memory Designs					
Trajectory Retrieval	80.0 ± 1.4 +12.4	67.0 ± 1.3 +6.5	30.9 ± 2.4 +9.5	16.7 ± 0.8 +1.7	48.6 +7.5
Reasoning Bank	67.1 ± 2.9 -0.5	56.1 ± 3.4 -4.4	21.4 ± 4.1 +0.0	15.8 ± 3.0 +0.8	40.1 -1.0
Dynamic Cheatsheet	78.6 ± 3.3 +11.0	57.8 ± 7.9 -2.7	**38.0** ± 6.3 +16.6	11.7 ± 3.3 -3.3	46.5 +5.4
G-Memory	74.8 ± 0.5 +7.2	68.8 ± 4.7 +8.3	26.2 ± 6.3 +4.8	14.2 ± 2.2 -0.8	46.0 +4.9
Learned Memory Designs (Transferred)					
ALMA (Ours)	**87.1** ± 1.4 +19.5	**75.0** ± 2.3 +14.5	33.3 ± 2.4 +11.9	**20.0** ± 2.9 +5.0	**53.9** +12.8

act with it by generating actions in natural language. The benchmarks are well-suited for evaluating memory designs because they require acquiring and utilizing experience that does not exist in the pre-trained knowledge of FMs (Paglieri et al., 2025). We test ALMA on four benchmarks: (1) ALFWorld (Shridhar et al., 2021), a text-based simulation of embodied household tasks that challenges agents to ground language instructions to action sequences in kitchen environments; (2) TextWorld (Côté et al., 2019), text adventure games that require systematic exploration and reasoning to navigate partially observable environments; (3) Baba Is AI (Cloos et al., 2025), a strategic puzzle game where agents must manipulate the rules of the game, requiring complex reasoning and adaptation to changing game mechanics; (4) MiniHack (Samvelyan et al., 2021), a simplified version of NetHack (Küttler et al., 2020) that challenges agents with long-horizon decision-making in procedurally generated dungeon environments.

For ALFWorld, we are using the default setting (Shridhar et al., 2021). For all other benchmarks, we use BALROG (Paglieri et al., 2025), following its settings and prompt for interactions during learning and final testing. More details on benchmarks are available in Appendix B.1.

4.3 Baselines

We select four representative state-of-the-art human-designed memory systems as baselines. These baselines are entirely handcrafted, with all aspects of the design (e.g., what to store, how to update and retrieve) manually specified and inserted retrieved knowledge into the context of agents, representing various mainstream approaches in prior work. More details are available in Appendix B.2.

Trajectory Retrieval. A straightforward but strong baseline that retrieves similar trajectories (Park et al., 2023; Xu et al., 2024). Task descriptions and trajectories are stored during memory update. During memory retrieval, the most relevant past trajectory is retrieved as knowledge based on the similarity of the task description embedding.

ReasoningBank. The memory designs (Ouyang et al., 2025) organize and extract experience on per-task basis, enabling the retrieval of experience relevant to each task. During memory update, experiences are extracted from each trajectory and stored together with the corresponding task description. The experiences corresponding to the most similar task description are retrieved as knowledge.

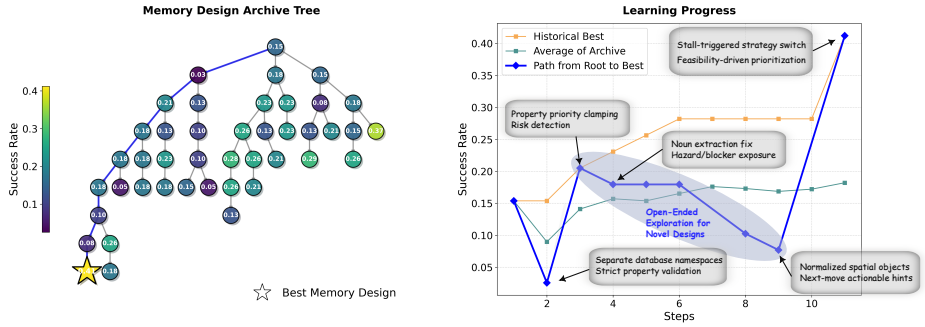

Figure 2: The learning process of ALMA on Baba Is AI, using `GPT-5-nano` as the FM in an agentic system. The learning processes of other benchmarks are shown in Appendix C.2. **Left:** The memory design archive tree, where each node represents a memory design produced during the open-ended exploration for ever-better memory designs. Node colors indicate the success rate, and edges indicate that each child node is derived from its parent. The memory design with the highest success rate is used as the final learned memory design. **Right:** The step-wise learning progress. ALMA progressively discovers memory designs by building on an ever-growing archive of previous discoveries. The path from the root memory design to the best memory design highlights the importance of open-ended exploration, where designs with moderate success rates serve as stepping stones toward optimal solutions.

Dynamic Cheatsheet (Cumulative). A global semantic memory incrementally accumulates experience by FMs across all past trajectories (Suzgun et al., 2025). The trajectories are sequentially used to update a global cheatsheet, which works as knowledge across all tasks.

G-memory. A hierarchical graph-based memory design (Zhang et al., 2025a). During the update of memory, it builds a hierarchical memory where insights and key steps in trajectories are connected in a graph based on task descriptions. When facing new tasks, it traverses the graph to retrieve relevant insights and key steps as knowledge.

4.4 Results

For each benchmark, we select the memory design achieving the highest success rate during the learning of memory designs as the best-learned memory design. We then evaluate both human-designed baselines and the best-learned memory designs using the testing process described in Section 4.1 with static mode in the Deployment Phase (Section 3.2).

As shown in table 1 Top, with the agentic system powered by `GPT-5-nano`, the learned memory designs achieve an overall improvement of 6.2% compared to the no-memory baseline, and outperform all state-of-the-art human-designed memory baselines. These improvements are consistent across benchmarks, demonstrating that learned memory designs can more effectively store and reuse past experience to continually learn than manual designs.

We further evaluate learned memory designs by changing the FM in the agentic system from `GPT-5-nano` to `GPT-5-mini` (Table 1 Bottom). The learned memory designs improve the overall average success rate by 12.8% over the no-memory baseline and outperform all human-designed baselines. These results demonstrate that the discovered memory designs generalize across different FMs, indicating that the improvements are robust and not tied to a specific model. The larger improvement observed with more capable FMs (12.8% vs 6.2%) yields a delta of 6.6%. This delta exceeds those of all human-designed memory baselines, suggesting that our learned memory designs provide stronger support to more capable agentic systems.

To provide a closer look at the learning process, we show the learning process on Baba Is AI and the tree visualization of the resulting memory design archive (fig. 2). The Meta Agent explores diverse new designs on the path toward the best-learned memory design, building upon designs with moderate success rates that nevertheless have the potential to evolve into the optimal memory design. As shown in fig. 2, during open-ended exploration, the Meta Agent incrementally introduces mechanisms like property validation and spatial object normalization. While these mechanisms may not yield immediate performance gains, they serve as stepping stones that contribute to the best-learned memory design when key mechanisms (e.g., strategy switching) are introduced. Visualizations of the learning processes for the other benchmarks are available in the Appendix C.2.

To evaluate the effectiveness of the open-ended exploration in ALMA, we conduct an ablation study comparing against greedy search, which always samples the design with the highest success rate to propose new designs. More details on greedy search are available in Appendix B.3. We perform the learning process with greedy search on ALFWorld, resulting in success rates of 11.9% and 77.1% when testing the best-learned memory design with the agentic system powered by `GPT-5-nano` and `GPT-5-mini`, respectively (Appendix C.3). In both cases (for `GPT-5-nano`, 11.9% vs 12.4%; for `GPT-5-mini`, 77.1% vs 87.1%), success rates are lower than those achieved by designs learned with open-ended exploration, demonstrating the benefits of an exploration-driven learning process over greedy search.

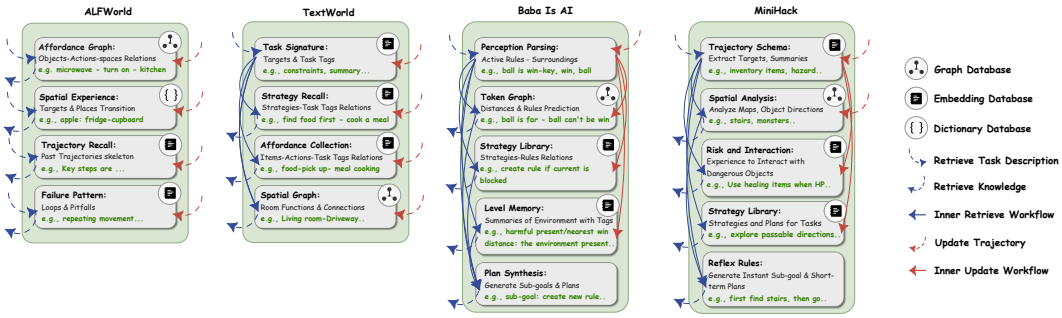

Figure 3: The visualization of the best-learned memory designs across different benchmarks. Each sub-module in memory designs may have a dedicated database or none, depending on its function, and arrows show the retrieval and update workflows in memory designs. The name and explanation of each sub-module are generated by Meta Agent and manually summarized, respectively. Example code and output for a learned memory design are provided.

We visualize the learned memory designs for each domain (fig. 3), showing that ALMA discovers effective memory structures adapted to diverse task requirements. For games with explicit object interaction goals (e.g., ALFWorld and TextWorld), the learned memory designs tend to store fine-grained knowledge, such as spatial relationships between objects and room layouts. In contrast, for tasks that require more complex reasoning (e.g., Baba Is AI and MiniHack), the memory designs favor domain-specific abstract strategy, including strategy libraries and plan synthesis. This pattern indicates that ALMA automatically specializes memory designs to the demands of each domain.

To study how performance scales with experience, we evaluate memory designs using different sizes of tasks during the Memory Collection Phase and using static mode during the Deployment Phase. As shown in fig. 4 left, compared to manual memory designs, the learned memory design achieves higher performance faster with limited data and scales better as more trajectories are provided, demonstrating both higher sample efficiency and stronger scalability. We observe that increasing the trajectory size does not necessarily lead to a monotonic improvement in success rate, which is expected since trajectories may vary in quality (Ouyang et al., 2025).

We evaluate the adaptability of learned memory designs to the distribution shift of tasks. To evaluate adaptation under distribution shift, memory is collected from 70 tasks in the *valid_seen* during the Memory Collection Phase and evaluated on the *valid_unseen* using dynamic mode during the Deployment Phase (Appendix B.4). This tests whether memory designs can adapt to tasks with distribution shift in *valid_unseen* by dynamically updating memory. As shown in fig. 4 right, the learned memory design achieves a success rate of 84.1% on ALFWorld, outperforming all human-designed baselines and demonstrating more effective adaptation under task distribution shift.

We measure the cost efficiency of learned memory designs by studying their end-to-end memory cost, defined as the total cost of FMs required to transform raw interaction logs into retrieved content inserted into the agentic system's context across the Memory Collection and Deployment Phases. We also include the token size of the retrieved content. The details are available in Appendix A.3. As shown in Appendix C.1, compared to all manual designs, our learned memory designs achieve an average success rate of 53.9%, with lower overall end-to-end memory cost across all benchmarks ($0.09) and appropriate token sizes of retrieved content. The results show ALMA effectively balances memory efficiency and task performance. Since ALMA does not explicitly optimize for cost efficiency, we anticipate that incorporating techniques such as multi-objective optimization in future work could enable even more effective exploration of the efficiency-performance Pareto front.

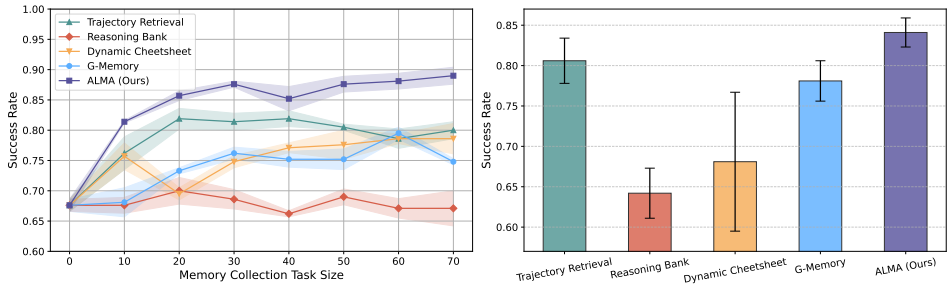

Figure 4: Comparison of memory designs in ALFWorld. Left: Success rates as the task size during the Memory Collection Phase increases. Evaluations are performed using static mode during the Deployment Phase to study how performance scales with collected static memory. Shaded areas indicate standard errors. Learned memory design achieves higher performance faster with limited data and scales better than human-designed baselines. **Right: Success rates under task distribution shift in the Deployment Phase.** Memory is collected from tasks in *valid_seen* and evaluated on the *valid_unseen* dataset with dynamic mode during the Deployment Phase. The error bars indicate the standard errors. The learned memory design adapts more effectively than human-designed baselines under task distribution shift. Both experiments use `GPT-5-mini` as the FM in the agentic system and calculate standard errors over three runs of the Deployment Phase.

5 Conclusion, Safety, and Future Work

We propose ALMA, a paradigm that discovers new memory designs, enabling agentic systems to learn from past experiences. We evaluate ALMA on sequential decision-making benchmarks. The learned designs consistently outperform manually designed baselines, demonstrating the effectiveness of ALMA. Furthermore, subsequent experiments show that learned memory designs exhibit strong scalability, transferability, and cost efficiency. Overall, ALMA provides a principled approach to learn memory design, taking a step toward continual learning in agentic systems.

ALMA represents a step toward continual learning in AI, contributing to the broader goal of developing AI-generating algorithms beyond current manual approaches (Clune, 2020). However, such AI-generating algorithms introduce unique safety concerns, as components of AI systems are learned rather than designed by humans. The learned components may inadvertently introduce behaviors that deviate from human intentions, ignoring optimization guided by a predefined target metric (Bostrom, 2017; Rokon et al., 2020; Ecoffet et al., 2020). Recognizing these challenges, we impose explicit constraints and restrictions in our experimental setup during the learning process. Specifically, each memory design is generated as code by the Meta Agent, they are validated and executed within isolated sandbox environments, with access confined to the sandbox to prevent any interference with the external system, mitigating the risk of unintended actions. We also perform human oversight of learned memory designs to ensure that no potentially harmful actions are included in memory designs. In future work, it will be essential to establish a systematic inspection mechanism as the system scales and is deployed, which may include AI and human inspection.

While ALMA demonstrates strong effectiveness in learning memory designs, it has several limitations that open up future research directions. One limitation is that the memory designs are learned using a pre-defined learning set, instead of dynamically learning memory designs when facing new tasks. Ideally, an adaptive continual learner would be able to learn memory designs online from dynamic data, without separating learning and testing phases. While ALMA has the potential to support online learning, we do not pursue this approach due to computational budget constraints, as evaluating each explored memory design would require a large number of rollouts. Therefore, a possible future direction is to enable online learning of memory design that continually adapts to arbitrary domains. Another limitation stems from the fact that ALMA learns memory designs in code space. While effective, their capabilities may be limited by the underlying FMs. Future work could explore automated designing and training novel FM architectures with native memory support.

Overall, ALMA represents a step toward automated continual-learning AI. While current limitations prevent us from demonstrating the learning of both memory and agentic system, our results highlight the potential to surpass manually designed memory, showing a way towards "learning to continually learn". When developed with safety measures, ALMA holds promise for a general-purpose AI capable of learning to continually learn across domains and autonomously adapting to new ones.

ACKNOWLEDGMENTS

This research was supported by the Vector Institute, the Canada CIFAR AI Chairs program, a grant from Schmidt Futures, an NSERC Discovery Grant, and a generous donation from Rafael Cosman. Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute (https://vectorinstitute.ai/partnerships/current-partners/). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

Fuma Aki, Riku Ikeda, Takumi Saito, Ciaran Regan, and Mizuki Oka. Llm-poet: Evolving complex environments using large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '24 Companion, pp. 243–246, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704956. doi: 10.1145/3638530.3654115. URL https://doi.org/10.1145/3638530.3654115.

Marcin Andrychowicz, Misha Denil, Sergio Gómez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/fb87582825f9d28a8d42c5e5e5e8b23d-Paper.pdf.

Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O. Stanley, Jeff Clune, and Nick Cheney. Learning to continually learn. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang (eds.), *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pp. 992–1001. IOS Press, 2020. doi: 10.3233/FAIA200193. URL https://doi.org/10.3233/FAIA200193.

N. Bostrom. Ethical Issues in Advanced Artificial Intelligence. 2017. ISSN 9781003074991. URL https://doi.org/10.4324/9781003074991-7. Edition: 1.

Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 11079–11091. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/47e288629a6996a17ce50b90a056a0e1-Paper-Conference.pdf.

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

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019. URL https://arxiv.org/abs/1904.10509.

Nathan Cloos, Meagan Jens, Michelangelo Naim, Yen-Ling Kuo, Ignacio Cases, Andrei Barbu, and Christopher J. Cueva. Baba is ai: Break the rules to beat the benchmark, 2025. URL https://arxiv.org/abs/2407.13729.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence, 2020. URL https://arxiv.org/abs/1905.10985.

Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31.

Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/b1301141feffabac455e1f90a7de2054-Paper.pdf.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. In Tristan Cazenave, Abdallah Saffidine, and Nathan Sturtevant (eds.), *Computer Games*, pp. 41–75, Cham, 2019. Springer International Publishing. ISBN 978-3-030-24337-1.

Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning, 2016. URL https://arxiv.org/abs/1611.02779.

Adrien Ecoffet, Jeff Clune, and Joel Lehman. Open questions in creating safe open-ended ai: Tensions between control and creativity. volume ALIFE 2020: The 2020 Conference on Artificial Life of *ALIFE 2022: The 2022 Conference on Artificial Life*, pp. 27–35, 07 2020. doi: 10.1162/isal_a_00323. URL https://doi.org/10.1162/isal_a_00323.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, February 2021. ISSN 1476-4687. doi: 10.1038/s41586-020-03157-9. URL https://doi.org/10.1038/s41586-020-03157-9. TLDR: Go-Explore, a family of algorithms that explicitly remembers promising states and returns to them as a basis for further exploration solves all as-yet-unsolved Atari games and out-performs previous algorithms on Montezuma's Revenge and Pitfall.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL http://jmlr.org/papers/v20/18-598.html.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Representation Learning*, volume 2025, pp. 85260–85385, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/d40d7cbe7210f8a13ea0149eeae9c6de-Paper-Conference.pdf.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory, 2026. URL https://arxiv.org/abs/2508.06433.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. Genenv: Difficulty-aligned co-evolution between llm agents and environment simulators, 2025. URL https://arxiv.org/abs/2512.19682.

Shengran Hu, Ran Cheng, Cheng He, Zhichao Lu, Jing Wang, and Miao Zhang. Accelerating multi-objective neural architecture search by random-weight evaluation. *Complex & Intelligent Systems*, 9(2):1183–1192, April 2023. ISSN 2198-6053. doi: 10.1007/s40747-021-00594-5. URL https://doi.org/10.1007/s40747-021-00594-5. TLDR: A new performance estimation metric, named random-weight evaluation (RWE) is introduced to quantify the quality of CNNs in a cost-efficient manner and reveals the effectiveness of the proposed RWE in estimating the performance compared to existing methods.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Representation Learning*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, Zhenrong Cheng, Xuanbo Fan, Jiaxin Guo, Xinlei Yu, Zhenhong Zhou, Zewen Hu, Jiahao Huo, Junhao Wang, Yuwei Niu, Yu Wang, Zhenfei Yin, Xiaobin Hu, Yue Liao, Qiankun Li, Kun Wang, Wangchunshu Zhou, Yixin Liu, Dawei Cheng, Qi Zhang, Tao Gui, Shirui Pan, Yan Zhang, Philip Torr, Zhicheng Dou, Ji-Rong Wen, Xuanjing Huang, Yu-Gang Jiang, and Shuicheng Yan. Memory in the age of ai agents, 2026. URL https://arxiv.org/abs/2512.13564.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (eds.). *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 7671–7684. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/569ff987c643b4bedf504efda8f786c2-Paper.pdf.

Joel Lehman and Kenneth O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In Seth Bullock, Jason Noble, Richard A. Watson, and Mark A. Bedau (eds.), *Proceedings of the Eleventh International Conference on the Synthesis and Simulation of Living Systems, ALIFE 2008, Winchester, United Kingdom, August 5-8, 2008*, pp. 329–336. MIT Press, 2008. URL http://mitpress2.mit.edu/books/chapters/0262287196chap43.pdf.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011a. doi: 10.1162/EVCO_a_00025.

Joel Lehman and Kenneth O. Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pp. 211–218, New York, NY, USA, 2011b. Association for Computing Machinery. ISBN 9781450305570. doi: 10.1145/2001576.2001606. URL https://doi.org/10.1145/2001576.2001606.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. *Evolution Through Large Models*, pp. 331–366. Springer Nature Singapore, Singapore, 2024. ISBN 978-981-99-3814-8. doi: 10.1007/978-981-99-3814-8_11. URL https://doi.org/10.1007/978-981-99-3814-8_11.

Sirui Liang, Pengfei Cao, Jian Zhao, Wenhao Teng, Xiangwen Liao, Jun Zhao, and Kang Liu. Learning how to remember: A meta-cognitive management method for structured and transferable agent memory, 2026. URL https://arxiv.org/abs/2601.07470.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, Mingxuan Yuan, and Qingfu Zhang. A systematic survey on large language models for algorithm design. *ACM Comput. Surv.*, January 2026. ISSN 0360-0300. doi: 10.1145/3787585. URL https://doi.org/10.1145/3787585. Just Accepted.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex J. Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. Discovering preference optimization algorithms with and for large language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 86528–86573. Curran Associates, Inc., 2024a. doi: 10.52202/079017-2748. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/9d88b87b31986f8293bb0067a841579e-Paper-Conference.pdf.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery, 2024b. URL https://arxiv.org/abs/2408.06292.

Cong Lu, Shengran Hu, and Jeff Clune. Intelligent go-explore: Standing on the shoulders of giant foundation models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Representation Learning*, volume 2025, pp. 21276–21301, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/369a30aac2765950865efd318cef7f76-Paper-Conference.pdf.

Cong Lu, Shengran Hu, and Jeff Clune. Automated capability discovery via foundation model self-exploration, 2025b. URL https://arxiv.org/abs/2502.07577.

Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pp. 419–427, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361118. doi: 10.1145/3321707.3321729. URL https://doi.org/10.1145/3321707.3321729.

Jiayan Nan, Wenquan Ma, Wenlong Wu, and Yize Chen. Nemori: Self-organizing agent memory inspired by cognitive science, 2025. URL https://arxiv.org/abs/2508.03341.

Ben Norman and Jeff Clune. First-explore, then exploit: Meta-learning to solve hard exploration-exploitation trade-offs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 27490–27528. Curran Associates, Inc., 2024. doi: 10.52202/079017-0864. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/30754e5f4cd69d64b5527cdd87d3cf62-Paper-Conference.pdf.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory, 2025. URL https://arxiv.org/abs/2509.25140.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Davide Paglieri, Bartł omiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Ł ukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Foerster, Jack Parker-Holder, and Tim Rocktaeschel. Balrog: Benchmarking agentic llm and vlm reasoning on games. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Representation Learning*, volume 2025, pp. 96666–96702, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f0b1515be276f6ba82b4f2b25e50bef0-Paper-Conference.pdf.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701320. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. Various lengths, constant speed: Efficient language modeling with lightning attention. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 41517–41535. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/qin24c.html.

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

Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos. SourceFinder: Finding malware Source-Code from publicly available repositories in GitHub. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pp. 149–163, San Sebastian, October 2020. USENIX Association. ISBN 978-1-939133-18-2. URL https://www.usenix.org/conference/raid2020/presentation/omar.

J Rosser and Jakob Foerster. Agentbreeder: Mitigating the ai safety impact of multi-agent scaffolds via self-improvement. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. NeurIPS 2025 Spotlight.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper-round1.pdf.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Representation Learning*, volume 2025, pp. 3841–3865, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/0ae94013da7cd459402fd77874e09ee3-Paper-Conference.pdf.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. {ALFW}orld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Kenneth O. Stanley. Why open-endedness matters. *Artificial Life*, 25(3):232–235, 08 2019. ISSN 1064-5462. doi: 10.1162/artl_a_00294. URL https://doi.org/10.1162/artl_a_00294.

Rich Sutton. The bitter lesson, Mar 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html. Accessed: 2026-01-22.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory, 2025. URL https://arxiv.org/abs/2504.07952.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, and Wangchunshu Zhou. AGENT KB: A hierarchical memory framework for cross-domain agentic problem solving. In *ICML 2025 Workshop on Collaborative and Federated Agentic Workflows*, 2025. URL https://openreview.net/forum?id=ohXoWHlrn8.

S.K. Thompson. *Sampling*. CourseSmart Series. Wiley, 2012. ISBN 9780470402313. URL https://books.google.ca/books?id=9MYjqz4ppXkC.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn, 2017. URL https://arxiv.org/abs/1611.05763.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Poet: open-ended coevolution of environments and their optimized solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pp. 142–151, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361118. doi: 10.1145/3321707.3321799. URL https://doi.org/10.1145/3321707.3321799.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth Stanley. Enhanced POET: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 9940–9951. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/wang20l.html.

Yu Wang, Yifan Gao, Xiusi Chen, Haoming Jiang, Shiyang Li, Jingfeng Yang, Qingyu Yin, Zheng Li, Xian Li, Bing Yin, Jingbo Shang, and Julian Mcauley. MEMORYLLM: Towards self-updatable large language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50453–50466. PMLR, 21–27 Jul 2024a. URL https://proceedings.mlr.press/v235/wang24s.html.

Yu Wang, Ryuichi Takanobu, Zhiqi Liang, Yuzhen Mao, Yuanzhe Hu, Julian McAuley, and Xiaojian Wu. Mem-α: Learning memory construction via reinforcement learning, 2025. URL https://arxiv.org/abs/2509.25911.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory, 2024b. URL https://arxiv.org/abs/2409.07429.

Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. Retrieval meets long context large language models. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Representation Learning*, volume 2024, pp. 49569–49584, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/d75f29006df67df084e6586f1cb8458c-Paper-Conference.pdf.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search, 2025. URL https://arxiv.org/abs/2504.08066.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning, 2026. URL https://arxiv.org/abs/2508.19828.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. MAS-GPT: Training LLMs to build LLM-based multi-agent systems. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=3CiSpY3QdZ.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1354. URL https://aclanthology.org/2025.acl-long.1354/.

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

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

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet, 2025c. URL https://arxiv.org/abs/2502.04180.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems, 2025d. URL https://arxiv.org/abs/2512.18746.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. Omni: Open-endedness via models of human notions of interestingness. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Representation Learning*, volume 2024, pp. 6074–6120, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/18b40f124256aa0dcbb3e2832cce252e-Paper-Conference.pdf.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025e. URL https://arxiv.org/abs/2505.22954.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Representation Learning*, volume 2025, pp. 34040–34077, 2025f. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, Jian Xie, Yuxuan Sun, Boyu Gou, Qi Qi, Zihang Meng, Jianwei Yang, Ning Zhang, Xian Li, Ashish Shah, Dat Huynh, Hengduo Li, Zi Yang, Sara Cao, Lawrence Jang, Shuyan Zhou, Jiacheng Zhu, Huan Sun, Jason Weston, Yu Su, and Yifan Wu. Agent learning via early experience, 2025g. URL https://arxiv.org/abs/2510.08558.

Yao Zhang, Chenyang Lin, Shijie Tang, Haokun Chen, Shijie Zhou, Yunpu Ma, and Volker Tresp. Swarmagentic: Towards fully automated agentic system generation via swarm intelligence, 2025h. URL https://arxiv.org/abs/2506.15672.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents, 2024b. URL https://arxiv.org/abs/2404.13501.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Trans. Inf. Syst.*, 43(6), September 2025i. ISSN 1046-8188. doi: 10.1145/3748302. URL https://doi.org/10.1145/3748302.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, Mar. 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19724–19731, Mar. 2024. doi: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

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

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. Symbolic learning enables self-evolving agents, 2024. URL https://arxiv.org/abs/2406.18532.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=r1Ue8Hcxg.

Jiaru Zou, Xiyuan Yang, Ruizhong Qiu, Gaotang Li, Katherine Tieu, Pan Lu, Ke Shen, Hanghang Tong, Yejin Choi, Jingrui He, James Zou, Mengdi Wang, and Ling Yang. Latent collaboration in multi-agent systems, 2025. URL https://arxiv.org/abs/2511.20639.

A Algorithmic Details

A.1 Search Space

Listing 1: The Python abstract class used in the learning process. In the code, sub-modules are implemented as layers, with each layer encapsulating the functionality of a module. The overall execution order of these update and retrieval operations across sub-modules is orchestrated by the general interfaces: `general_retrieve` and `general_update`. The example of learned memory design is available at Appendix C.4

```python
from abc import ABC, abstractmethod
from typing import Dict, Optional, Any
from dataclasses import dataclass
from eval_envs.base_envs import Basic_Recorder

@dataclass
class Sub_memo_layer(ABC):
    """
    Abstract class for retrieve/update sub-function
    """
    layer_intro: str = "Introduction of the structure of current
        defined Database(if any), corresponding Update and
        Retrieve method."
    database: Optional[Any] = None

    @abstractmethod
    async def retrieve(self, **kwargs):
        """
        The retrieve function of current layer.
        """
        pass

    @abstractmethod
    async def update(self, **kwargs):
        """
        The update function of current layer.
        """
        pass

class MemoStructure(ABC):

    def __init__(self):
        self.database: Optional[Any] = None

    # -------- Pipeline Runner --------
    @abstractmethod
    async def general_retrieve(self, recorder: Basic_Recorder) ->
        Dict:
        """
        The general retrieve method, use the retrieve function in
            each layer, determine order and input-output usage by
            yourself.
        """
        pass

    @abstractmethod
    async def general_update(self, recorder: Basic_Recorder) ->
        None:
        """
        The general update method, use the update function in each
             layer, determine order and input-output usage by
            yourself.
        """
        pass
```

A.2 Evaluation of Memory design

We first formally define a memory design as a triplet $\mathcal{M} = (U, D, R)$, where U and R denote the update and retrieval processes that interface with the agentic system, and D represents the internal storage structure used for memory retention. To evaluate a memory design on a dataset $\mathcal{G}$, we split $\mathcal{G}$ evenly into two subsets: a set $\mathcal{G}_{\text{collection}}$ for memory collection and a set $\mathcal{G}_{\text{deployment}}$ for evaluation during the Deployment Phase. During the Memory Collection Phase, the agentic system interacts with the environment to finish tasks in $\mathcal{G}_{\text{collection}}$ without any memory access, producing interaction logs that are used to perform memory updates and construct a static memory. During the Deployment Phase, the agentic system retrieves relevant contexts from the static memory, which are then integrated into the agentic system to facilitate task execution within $\mathcal{G}_{\text{deployment}}$. The performance on $\mathcal{G}_{\text{deployment}}$ reflects the ability of a memory design to collect and utilize knowledge.

Furthermore, when evaluated on a dataset with task distribution shift during the Deployment Phase, the agentic system first retrieves existing knowledge to construct the knowledge context for the new task, and subsequently updates the memory with the resulting interaction log.

Memory Collection Phase. $\mathcal{G}_{\text{collection}}$ is used to collect a set of interaction trajectories $\mathcal{T}_{\text{collection}} = \{\tau^{(j)}_{\text{collection}}\}_{j=1}^{N}$ without any memory access. Each trajectory $\tau^{(j)}_{\text{collection}}$ takes the form

$$\tau^{(j)}_{\text{collection}} = \left(s^{(j)}_1, a^{(j)}_1, s^{(j)}_2, a^{(j)}_2, \ldots, s^{(j)}_{n_j}, a^{(j)}_{n_j}\right), \quad a^{(j)}_t \sim \pi(\cdot \mid s^{(j)}_{\leq t}, a^{(j)}_{<t})$$

where s denotes the environment state, and a denotes the action produced by the fixed agentic policy π. For each trajectory, we further obtain a task-level feedback

$$f^{(j)}_{\text{collection}} = F\left(\tau^{(j)}_{\text{collection}}\right)$$

where $F(\cdot)$ denotes the benchmark-specific evaluation function that assigns a scalar score to each trajectory. We then sequentially update the memory database using the collected interaction logs

$$D_j = U\left(\tau^{(j)}_{\text{collection}}, f^{(j)}_{\text{collection}}, D_{j-1}\right)$$

where the memory state is updated after each collected trajectory. The final collected memory is denoted as D_N.

Deployment Phase. During the Deployment Phase, we evaluate the performance of the agentic system, with knowledge retrieved from D_N, on the deployment set $\mathcal{G}_{\text{deployment}} = \{g^{(i)}_{\text{deployment}}\}_{i=1}^{K}$.

For each task $g^{(i)}_{\text{deployment}}$, we first retrieve relevant knowledge from the memory state D_N

$$e^{(i)} = R(D_N, s^{(i)}_1)$$

where $s^{(i)}_1$ denotes the initial state, including observation and task objective of the current task. The agent then interacts with the environment, resulting in a trajectory $\tau^{(i)}_{\text{deployment}}$, where each action is sampled according to

$$a^{(i)}_t \sim \pi(\cdot \mid s^{(i)}_{\leq t}, a^{(i)}_{<t}, e^{(i)})$$

The feedback is calculated as $f^{(i)}_{\text{deployment}} = F(\tau^{(i)}_{\text{deployment}})$. The overall performance of the memory design $\mathcal{M}$ is then computed as the average over all deployment tasks

$$f_{\mathcal{M}} = \frac{1}{K}\sum_{i=1}^{K} f^{(i)}_{\text{deployment}}$$

When performing Deployment Phase with distribution shift tasks, memory state is further updated sequentially using the collected trajectories $\tau^{(i)}_{\text{deployment}}$ and feedback signals $f^{(i)}_{\text{deployment}}$, producing D_{N+i}. The updated memory state is then used for retrieval when performing the $(i+1)$-th task.

A.3 Evaluation Cost

End-to-End Memory Cost. This metric quantifies the cumulative computational overhead incurred by the FMs to generate the content integrated into the agentic system with static mode in Deployment Phase. The cost includes: (1) the expense during the Memory Collection Phase to produce the memory state D_N; and (2) the cumulative cost of generating knowledge $\{e^{(i)}\}_{i=1}^{K}$ for all tasks during the Deployment Phase. We suggest that this provides a fair and unbiased way to measure the cost of a memory design, independent of its effectiveness. During the Memory Collection Phase, the agentic system interacts with environments without memory access, ensuring that all trajectories are unaffected by the memory design's performance. Since these trajectories are the only input used to produce memory, the costs of producing the memory state D_N and generating relevant knowledge during static deployment depend solely on these trajectories and are therefore independent of the memory design's effectiveness. We present a summary of the costs associated with testing a memory design across all benchmarks, including `GPT-4o-mini` and `text-embedding-3-small`.

Token Size of Retrieved Knowledge. We measure the token count of the retrieved context $e^{(i)}$ for each task during the static Deployment Phase. This metric quantifies the informational overhead injected into the agent's prompt, serving as a key proxy for measuring the inference-time efficiency of the memory design $\mathcal{M}$. The token size is also independent of effectiveness of a memory design, since it is dependent solely on trajectories collected in Memory Collection Phase. We utilize the `GPT-5-mini` tokenizer to calculate the token sizes.

A.4 Memory Design Archive and Sampling Process

Memory Design Archive. To iteratively explore new memory designs, we maintain a memory design archive $\mathcal{A}_l$ that stores all designs discovered up to learning step l. For each candidate design $\mathcal{M}$, we perform a stratified sampling process (Thompson, 2012) after its evaluation on a large deployment set. Specifically, instead of storing the entire interaction logs, we extract a small subset of size K_f as a proxy for the design's performance. This subset is denoted as

$$\mathcal{S}_{\mathcal{M}} = \{(e^{(i)}, \tau^{(i)}_{\text{deployment}}, f^{(i)}_{\text{deployment}})\}_{i=1}^{K_f}$$

where each tuple contains the retrieved context $e^{(i)}$, the resulting trajectory $\tau^{(i)}_{\text{deployment}}$, and the task-level feedback $f^{(i)}_{\text{deployment}}$.

The archive further records the overall performance $f_{\mathcal{M}}$ and the times a memory design is being sampled $t_{\mathcal{M}}$, which tracks the number of times $\mathcal{M}$ has been sampled and evaluated across l learning steps. Formally, at step l, the memory design archive is defined as the collection of these records

$$\mathcal{A}_l = \left\{ (\mathcal{M}, f_{\mathcal{M}}, \mathcal{S}_{\mathcal{M}}, t_{\mathcal{M}}) \;\middle|\; \mathcal{M} \text{ discovered before step } l \right\}.$$

Sampling Process. Similar to prior exploration approaches (Ecoffet et al., 2021; Zhang et al., 2025e), given the current memory design archive, we perform a sampling process to sample memory designs that will be used in the next learning step. For each memory design $\mathcal{M}_i$, we first compute the normalized performance relative to a baseline f_0

$$\hat{f}_{\mathcal{M}_i} = \sigma(f_{\mathcal{M}_i} - f_0) = \frac{1}{1 + \exp(-\lambda(f_{\mathcal{M}_i} - f_0))}$$

where f_0 denotes the performance of the agentic system on $\mathcal{G}_{\text{deployment}}$ without memory access. Subtracting f_0 ensures that the normalized score reflects the performance gain provided by the memory design. The final sampling score of each memory design is then given by

$$J_i \equiv J_{\mathcal{M}_i} = \hat{f}_{\mathcal{M}_i} - \alpha \log(1 + t_{\mathcal{M}_i})$$

where $t_{\mathcal{M}_i}$ is the number of times $\mathcal{M}_i$ has been sampled, and α controls the strength of the penalty. We then compute the sampling probability using a softmax over the final sampling scores

$$p_i = \frac{\exp\left(J_i/T - \max_j J_j/T\right)}{\sum_{j=1}^{|\mathcal{A}_l|} \exp\left(J_j/T - \max_j J_j/T\right)}, \quad i = 1, \ldots, |\mathcal{A}_l|$$

where T is a temperature parameter controlling the smoothness of the distribution. The sampling mechanism prioritizes high-performing yet under-sampled memory designs to balance exploration and exploitation. Meanwhile, designs with moderate performances retain a non-zero sampling probability to maintain diversity throughout the learning process. Finally, we sample memory designs according to $\{p_i\}$:

$$\mathcal{M}_{\text{sampled}} \subset \{\mathcal{M}_1, \ldots, \mathcal{M}_{|\mathcal{A}_l|}\}, \quad \mathcal{M}_{\text{sampled}} \sim \text{Categorical}(p_1, \ldots, p_{|\mathcal{A}_l|})$$

A.5 Prompts of Meta Agent

Prompt for Ideate & Planning

```
You are a **Senior Agent Construction Engineer** responsible for
    provide suggestions for a memory structure written by a entry
    level engineer, to make the memory structure better for downstream
     agent to finish tasks.
### Memo Information Overview
1. **source_code**
   - Each layer(which inherits `Sub_memo_layer`) contains:
     - **Retrieve**: fetches relevant memory elements from the
         database.
     - **Update**: writes or modifies entries in the database.
   - Final MemoStructure(which inherits `MemoStructure`) contains:
       - **general_retrieve**: general retrieve method, that contain
           the order and input-output usage of retrieve function of
           each layer.
       - **general_update**: general update method, that contain the
           order and input-output usage of update function of each
           layer.
   - code usage: Your memory structure will be used in the agent
       workflow:
       - `general_retrieve(recorder)`: used **before** start executing
           the task, to retreive task relevant information. Your output
            json will be directly send to agent, so please make sure
           your output is well organized, include all useful
           information and avoid redundency, in a agent understandable
           way.
       - `general_update(recorder)`: used **after** task is finished,
           to update the trajectory, reward, or other information.
2. **examples**
   - **examples**: some trajectries, including the memory retrieved(
       memory retrived should only happened before starting the task),
        and the steps of actions took by agent.
3. **benchmark_eval_score**
   - performance(success rate) of current memory stucture + general
       agent system. Need to use the score to analyze the performance
       and bottleneck of current memory structure.

### Your Task:
You will analyze past suggestion examples(including past source code,
    suggestions, and the improve score it led to) and the current
    retrieved trajectories and memory source code, then produce
    concrete, prioritized suggestions to improve the memory structure.
Follow the numbered procedure below and produce the requested
    structured outputs.
Step 1 - Learn from past suggestions & the improve score
   1. Look at the provided improve_score (positive as improvement,
       negative as degradation) and the single suggestion_example that
        produced that score.
   2. Explain why that suggestion led to improvement or degradation:
       - What pattern in the change made it succeed or fail?
       - Which behaviors, assumptions, or shortcuts in that suggestion
           were helpful? Which were harmful?
```

```
      - From these concrete cases, extract 2 to 5 general principles
          to adopt and 2 to 5 pitfalls to avoid when creating future
          suggestions.

Step 2 - Inspect sampled trajectories and benchmark performance and
    decide which memories are useful
   1. Review the current trajectory_examples (they include episodes
       with varying rewards).
   2. For each retrieved memory item (or memory group) returned for
       the trajectory, label it as one of:
      - Useful & Relevant - clearly applies to the current situation
          and can guide action;
      - Potentially Useful - has value but needs reformatting,
          summarization, or indexing to be helpful;
      - Irrelevant / Confusing - not related to this trajectory or
          misleading;
      - Empty / Badly Formatted - blank, placeholder, or not parseable
          .
   3. For each memory you mark Useful/Potentially Useful, say how it
       would help (e.g., provides a repeated subgoal, highlights a
       trap, identifies key object interactions).
   4. For Irrelevant/Empty items, explain why they failed retrieval
       combine with the memory source code (e.g., wrong keying, over-
       specific content, missing summarization).

Step 3 - Inspect memory source and produce concrete suggestions
   1. Review the memory source code (retrieval keys, indexing, storage
        format, layers). Using Step 1 principles and Step 2 labels,
       propose specific changes to the memory system that address the
       observed issues.
   2. Combined the memory source code with your analysis in step 2,
       giving suggestions. For each suggested change, include:
      - What to change (code-level or pipeline change, e.g., add
          summarization layer, change indexing key, normalize objects
          to noun-phrases).
      - Why it will help (link back to a principle or a concrete
          failing you observed).
   3. Prioritize suggestions: label them High / Medium / Low priority
       and give an implementation order.
   4. Link Analysis to Benchmark Performance
      - Use benchmark_eval_score to identify which structural
          weaknesses correlate with poor performance.

Extra checks (quality & coherence)
   1. Flag obvious content issues: duplicates, empty entries, raw
       dumps, mis-typed fields, or numeric types that break JSON
       serialization.
   2. Check layer interaction: do layers pass structured outputs to
       each other, or only dump free-form text?
   3. If retrieval returns empty lists or dicts, emphasize structural
       fixes (keying, ensure type consistency, avoid over-relying on
       try/except fallbacks).

Goal: Combine reflection on past improvement signals with current
    system diagnosis to produce actionable, high-level suggestions
    that strengthen memory structure quality.

### Benchmark Information:
{TASK_DESCRIPTION[task_type]}

### Required Output:
Return a JSON object followed below json schema:
```

```
{json.dumps(MEMO_ANALYSIS_OUTPUT_FORMAT, indent = 2, ensure_ascii =
    False)}
```

Prompt for Implementation

```
You are a senior AI software engineer. Your task is to build an agent
    memory system composed of multiple specialized memory layers and a
     coordinating memory structure. The agent will be used in {
    task_type}. Yout memory structure aims to help down stream agent
    to have more relevant advice, experience to help it further finish
     current task.

{TASK_DESCRIPTION[task_type]}

You are given the following two base classes:
<BACKBONE_CODE>
{basic_classes}
</BACKBONE_CODE>
Inherit these two base classes and recorder by writing:
```python
from agents.memo_structure import Sub_memo_layer, MemoStructure
from eval_envs.base_envs import Basic_Recorder
from utils.hire_agent import Agent, Embedding
from langchain_chroma import Chroma
```

<CODE_INPUT>
{interaction_prompt}
</CODE_INPUT>

<CODE_USAGE>
Your memory structure will be used in the agent workflow:
   - `general_retrieve(recorder)`: used **before** start executing the
        task, to retreive task relevant information. Your output json
       will be directly send to agent, so please make sure your output
        is well organized, include all useful information and avoid
       redundency, in a agent understandable way.
   - `general_update(recorder)`: used **after** task is finished, to
       update the trajectory, reward, or other information.
<CODE_USAGE>

Here is the basic tools provided:
<GRAPH_DATABASE_INTERACTION>
{NXGRAPH_CHEETSHEET}
</GRAPH_DATABASE_INTERACTION>

<CHROMA_DATABASE_INTERACTION>
{CHROMA_CHEETSHEET}
</CHROMA_DATABASE_INTERACTION>

<OTHER_TOOLS>
{TOOL_CHEETSHEET}
</OTHER_TOOLS>

Your Task:
Modify the code above so that it fully satisfies the following design
    goals:

1. **Multiple Memory Layers:**
  - Create multiple subclasses of `Sub_memo_layer`.
```

```
  - Each layer must have a **clear responsibility** and you might need
      to maintain its own database(based on nx.graph or chroma).
  - Think carefully about **what type of data** belongs in each layer'
     s database.

2. **General Retrieve/Update Orchestration:**
  - Create a subclass of `MemoStructure` that orchestrates all layers.
  - `general_retrieve()` should intelligently chain the results:
    - Output from layer 1 can become input to layer 2.
    - You should design a reasonable order of retrieval.
  - `general_update()` should propagate updates to relevant layers in
     a sensible order.
  - You are free to reorder calls or preprocess inputs to achieve
     coherent memory behavior.

3. **Out-of-the-Box Reasoning:**
  - Do not just mechanically call each layer one by one ---think about
      the **semantic flow of information**.
  - Consider cases like:
    - what type of memory layer can be used according to the analysis
        result or task description, with the aim to better assist the
        agent to finish it's task?
    - what order and input output should be best suitable for the
        analysis result or task description?
    - Think about high-level stategy: what can be a good memory
        structure, and has good ability to transfer to other area?
  - Make sure each layer plays a meaningful role in the system.
  - Directly perform simple plans or content writing based on if/else
     patterns should be avoided, since this will hurt the transfer
     ability.
  - Keep the retrieved memory clean and useful, aviod cutting off
     meaningful texts, repeat same patterns in the retrieved memory.

4. **Integration with Utilities:**
  - Feel free to use any provided utility functions (e.g., similarity
     calculation, interaction with databases, hire new agent) if
     relevant. The tools available will be listed in `TOOLS` section.
  - You can also create your own tools if neccessary, think out of the
      box.

5. **Code Quality:**
  - Output clean, runnable Python code following PEP8.
  - Ensure `general_retrieve()` and `general_update()` accept `
     Basic_Recorder` and orchestrate the pipeline end-to-end.
  - Initialize all layers in `MemoStructure.__init__`.
  - Do not overuse defensive programming; raise appropriate exceptions
      when unexpected conditions occur to facilitate debugging.

6. **Coherent Policy Logic:**
   - Avoid placeholders like pass or # TODO.
   - Avoid hard-coded if/else branches or enumerated case handling;
       instead, express the logic through modular policy functions,
       scoring mechanisms, or composable decision rules.
   - Instead of enumerating case-specific rules, express generalizable
       principles that could apply across different families or new
       unseen tasks.
   - The logic should be adaptable and compositional, not dependent on
       predefined constants or string names.
   - Use abstractions instead of specific family identifiers.

The goal is to ensure the memory policy behaves consistently across
    tasks and supports generalization, not to hard-code specific task
    behaviors.
```

```
### Important:
- Think creatively about data flow ---outputs of one layer can feed
    into the next.
- Each layer's functionality and stored data should be clearly
    designed.
- Provide **only the final rewritten code**, no explanations.
```

Prompt for Debugging

```
You are a senior AI software engineer and code repair expert.
Your role is to carefully analyze the provided code and the error
    information, identify potential errors or design flaws, and
    directly rewrite or edit the code to fix those issues - while
    keeping the main design goals and intentions exactly the same.

You are given the following context and base classes:
<BACKBONE_CODE>
{basic_classes}
</BACKBONE_CODE>

<CODE_INPUT>
{interaction_prompt}
</CODE_INPUT>

<CODE_USAGE>
Your memory structure will be used in the agent workflow:
   - `general_retrieve(recorder)`: used **before** start executing the
        task, to retreive task relevant information. Your output json
       will be directly send to agent, so please make sure your output
        is well organized, include all useful information and avoid
       redundency, in a agent understandable way.
   - `general_update(recorder)`: used **after** task is finished, to
       update the trajectory, reward, or other information.
<CODE_USAGE>

### Your Task:
Carefully inspect the code, and error information, detect the root
    cause of the errors, structural issues, or missing implementations
    , and **only fix the root cause code**. Here are some cheetsheet
    that could be useful:

<GRAPH_DATABASE_INTERACTION>
{NXGRAPH_CHEETSHEET}
</GRAPH_DATABASE_INTERACTION>

<CHROMA_DATABASE_INTERACTION>
{CHROMA_CHEETSHEET}
</CHROMA_DATABASE_INTERACTION>

<OTHER_TOOLS>
{TOOL_CHEETSHEET}
</OTHER_TOOLS>

### Output:
Return **only the final corrected Python code**, no explanations or
    commentary.
```

A.6 PSEUDOCODE

Algorithm 1 Learning of Memory Designs

Input: initial archive $\mathcal{A}_0 = \emptyset$; meta-learning steps L; collection tasks $\mathcal{G}_{\text{collection}}$; deployment tasks $\mathcal{G}_{\text{deployment}}$.
Output: optimal memory design $\mathcal{M}^*$ and final archive $\mathcal{A}_L$.
for $l = 1$ **to** L **do**
  // Sampling Process.
  Sample a set of base memory designs $\mathcal{M}_{\text{sampled}} \subset \mathcal{A}_{l-1}$ based on historical performance and visit counts.
  Update visit time for each sampled design: $t_{\mathcal{M}_{\text{base}}} \leftarrow t_{\mathcal{M}_{\text{base}}} + 1, \quad \forall \mathcal{M}_{\text{base}} \in \mathcal{M}_{\text{sampled}}$
  // Parallel Refinement and Evaluation.
  for all $\mathcal{M}_{\text{base}} \in \mathcal{M}_{\text{sampled}}$ **in parallel do**
    $\rho \leftarrow \text{PLANNER}(\mathcal{M}_{base}, \mathcal{S}_{\mathcal{M}_{\text{base}}}, f_{\mathcal{M}_{base}})$ *// Planning*
    $\tilde{\mathcal{M}} \leftarrow \text{IMPLEMENTOR}(\mathcal{M}_{\text{base}}, \rho)$ *// Implementation*
    $error_detected \leftarrow \text{TRIALRUN}(\tilde{\mathcal{M}})$
    if $error_detected$ **then**
      for $i = 1, \ldots, N$ **do**
        $\tilde{\mathcal{M}} \leftarrow \text{DEBUGGER}(\tilde{\mathcal{M}}, error_detected)$ *// Retry attempt i*
        $error_detected \leftarrow \text{TRIALRUN}(\tilde{\mathcal{M}})$
        if not $error_detected$ **then**
          break
        end if
      end for
    end if
    // Memory Collection Phase.
    Initialize $D_0 \leftarrow \emptyset$
    for all $g^{(j)} \in \mathcal{G}_{\text{collection}}, j = 1, \ldots, |\mathcal{G}_{\text{collection}}|$ **do**
      $\tau_{g^{(j)}} = \left(s_1, a_1, s_2, a_2, \ldots, s_{n_{g^{(j)}}}, a_{n_{g^{(j)}}}\right), \quad a_t \sim \pi(\cdot \mid s_{\leq t}, a_{<t})$ *// memory-free trajectory collection*
      $f_{g^{(j)}} \leftarrow F(\tau_{g^{(j)}})$ *// Feedback calculation*
      $D_j \leftarrow U_{\tilde{\mathcal{M}}}(\tau_{g^{(j)}}, f_{g^{(j)}}, D_{j-1})$ *// Memory update*
    end for
    // Deployment Evaluation Phase.
    for all $g \in \mathcal{G}_{\text{deployment}}, k = 1, \ldots, |\mathcal{G}_{\text{deployment}}|$ **do**
      $e_{g^{(k)}} \leftarrow R_{\tilde{\mathcal{M}}}(D_{|\mathcal{G}_{\text{collection}}|}, s_{g^{(k)}})$ *// Retrieve experience*
      $\tau_{g^{(k)}} = \left(s_1, a_1, s_2, a_2, \ldots, s_{n_{g^{(k)}}}, a_{n_{g^{(k)}}}\right), \quad a_t \sim \pi(\cdot \mid s_{\leq t}, a_{<t}, e_{g^{(k)}})$ *// Evaluation with memory*
      $f_{g^{(k)}} \leftarrow F(\tau_{g^{(k)}})$ *// Feedback calculation*
    end for
    Aggregate deployment performance $f_{\tilde{\mathcal{M}}} \leftarrow \frac{1}{K} \sum f_{g^{(k)}}$
    Store evaluation samples $\mathcal{S}_{\tilde{\mathcal{M}}} \leftarrow \{(e_g, \tau_g, f_g)\}$
    Insert $(\tilde{\mathcal{M}}, f_{\tilde{\mathcal{M}}}, \mathcal{S}_{\tilde{\mathcal{M}}}, t_{\tilde{\mathcal{M}}})$ into $\mathcal{A}_l$ *// Archive update*
  end for
end for
$\mathcal{M}^* \leftarrow \arg\max_{\mathcal{M} \in \mathcal{A}_L} f_{\mathcal{M}}$ *// Output optimal memory design*

B Experiment Details

B.1 Benchmark Details

We evaluate ALMA on four sequential decision-making benchmarks: ALFWorld (Shridhar et al., 2021), TextWorld (Côté et al., 2019), Baba Is AI (Cloos et al., 2025), and MiniHack (Samvelyan et al., 2021). All benchmarks involve multi-step interactions, providing a suitable testbed for evaluating how agents use memory designs to learn from experience.

ALFWorld. ALFWorld is a text-based embodied environment derived from ALFRED (Shridhar et al., 2020), where agents interact with household environments through natural language instructions (Shridhar et al., 2021). Tasks in ALFWorld require agents to perform multi-step manipulations, such as picking, placing, opening, and toggling objects, often with dependencies between sub-tasks. We follow the standard ALFWorld benchmark configurations, as shown in `https://github.com/alfworld/alfworld/blob/master/configs/base_config.yaml`. It contains a *train* dataset with 3553 tasks, a *valid_seen* with 140 tasks, and a *valid_unseen* dataset with 134 tasks. The *valid_seen* dataset only contains objects and rules that appear in the *train* dataset, while the *valid_unseen* dataset contrains objects and rules that do not appear in *train* dataset. Each episode terminates either when the task is successfully completed or when a maximum step limit is reached, providing a binary reward of 1 for success and 0 for failure. In our experiment, we set a maximum step limit of 100. An example interaction is shown in fig. 5.

Figure 5: The example of system prompt, user prompt, and possible response given by the agentic system on ALFWorld.

TextWorld. TextWorld is a text-based interactive environment designed to evaluate sequential decision-making agents through natural language interaction (Côté et al., 2019). In TextWorld, the agentic system navigates and manipulates a partially observable world using textual commands, such as moving between rooms, examining objects, and performing object-centric actions. We evaluate our method on the *Treasure Hunter* and *Cooking* games, using a total of 52 tasks across both games. Each task requires the agentic system to complete a goal specified in natural language, often involving exploration, information gathering, and multi-step reasoning. We perform evaluation using the BALROG benchmark (Paglieri et al., 2025) and follow its standard configuration and prompting scheme from Lu et al. (2025a) (`https://github.com/balrog-ai/BALROG/blob/main/balrog/config/config.yaml`). An episode terminates when the goal condition is satisfied or when the maximum step limit is reached, with a score ranging from 0 to 1 that measures the degree of task completion. In our experiments, we set the maximum number of interaction steps to 80. An example interaction is shown in fig. 6.

Figure 6: The example of system prompt, user prompt, and possible response given by the agentic system on *Cooking* game in TextWorld.

Baba Is AI. Baba Is AI is a grid-based puzzle environment designed to evaluate sequential decision-making agents in worlds governed by explicitly manipulable symbolic rules (Cloos et al., 2025). In Baba Is AI, an agentic system operates in a discrete grid world and interacts with objects by moving and pushing word tiles that define the rules of the environment. The transition rules and goal conditions in Baba Is AI are specified as compositional natural language-like statements (e.g.,

BABA IS YOU, FLAG IS WIN) that can be modified through interaction. Each task requires the agentic system to explore the grid, reason about the current rule set, and strategically alter rules to achieve the goal, often involving long-horizon planning and non-local dependencies. We evaluate on Baba Is AI using BALROG (Paglieri et al., 2025), following its standard Baba Is AI benchmark configuration and prompting scheme (https://github.com/balrog-ai/BALROG/blob/main/balrog/config/config.yaml), using a total of 52 tasks. An episode terminates when the win condition is satisfied under the current rule configuration or when the maximum step limit is reached, yielding a binary reward of 1 for success and 0 otherwise. In our experiments, we set the maximum number of interaction steps to 20. An example interaction is shown in fig. 7.

Figure 7: The example of system prompt, user prompt, and possible response given by the agentic system in Baba Is AI.

MiniHack. MiniHack is a grid-based sandbox environment built on top of the NetHack Learning Environment (NLE) (Samvelyan et al., 2021; Küttler et al., 2020), designed to evaluate sequential decision-making agents in procedurally generated dungeon worlds. We utilize *Navigation* games, which require exploration, planning, and strategic resource management. We evaluate on MiniHack using BALROG (Paglieri et al., 2025), following its standard MiniHack benchmark configuration and prompting scheme (https://github.com/balrog-ai/BALROG/blob/main/balrog/config/config.yaml), using a total of 114 tasks. An episode terminates when the goal condition is satisfied or when the maximum step limit is reached, yielding a binary reward of 1 for

success and 0 otherwise. In our experiments, we set the maximum number of interaction steps to 100. An example interaction is shown in fig. 8.

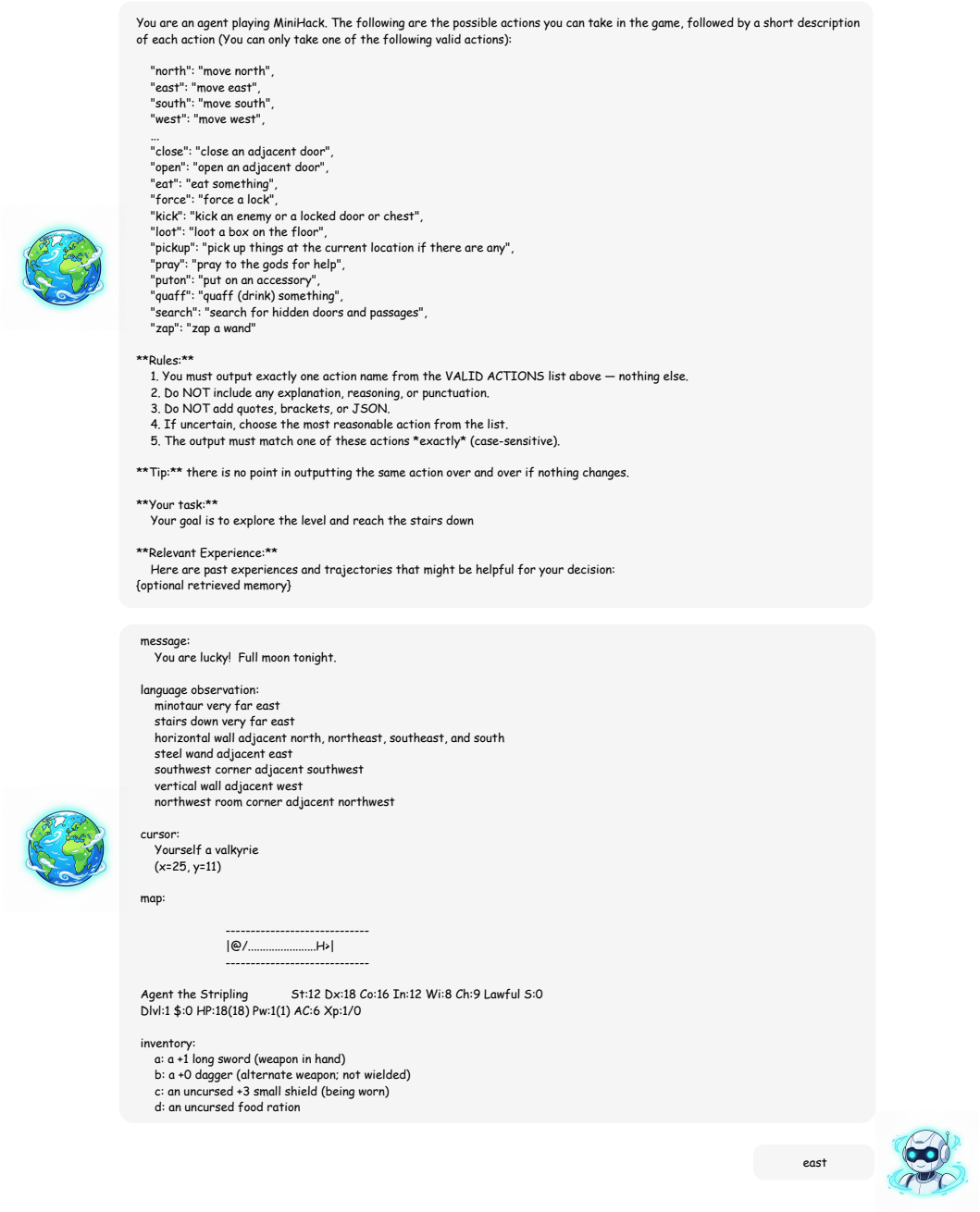

Figure 8: The example of system prompt, user prompt, and possible response given by the agentic system in MiniHack.

B.2 Baseline Details

Trajectory Retrieval. Trajectory Retrieval is a straightforward but strong baseline (Park et al., 2023; Xu et al., 2024). Given a task description, we encode it into a vector representation and retrieve the most similar past trajectory from the trajectory database based on cosine similarity. Due to context length constraints, only the Top-1 retrieved raw trajectory is provided as memory to the agentic system.

Dynamic Cheatsheet. Dynamic Cheatsheet (DC) maintains an external memory that tracks the successes and failures of the agentic system (Suzgun et al., 2025). We adopt the cumulative Dynamic Cheatsheet (DC-Cu), which maintains a single global cheatsheet throughout the evaluation of a memory design. We follow the prompts and workflows in `https://github.com/suzgunmirac/dynamic-cheatsheet`. During the Memory Collection Phase, trajectories are updated sequentially to incrementally update the global cheatsheet. In the Deployment Phase, the fixed global cheatsheet is provided to the agentic system to evaluate the effectiveness of a shared global memory across tasks.

ReasoningBank. ReasoningBank maintains a structured memory schema in which all memory items are stored (Ouyang et al., 2025). Each memory item consists of three components: (1) a title, which serves as an identifier summarizing the core strategy; (2) a description, providing a one-sentence summary of the memory item; and (3) the content, which records concise reasoning steps and insights. During the Memory Collection Phase, memory items are constructed for each trajectory by extracting either pitfalls or successful experiences, depending on the outcome of the trajectory. We follow the workflows and prompting strategies described in Ouyang et al. (2025), while using the ground-truth feedback of each trajectory. During the Deployment Phase, we first retrieve the Top-1 trajectory corresponding to the most similar task description from ReasoningBank, then use memory items associated with the trajectory. The retrieved memory items are provided as memory for the agentic system.

G-Memory. G-Memory uses a graph-based memory design, managing memory across three tiers: the insight, query, and interaction graphs (Zhang et al., 2025a). We follow the workflows, prompts, and default settings provided in `https://github.com/bingreeky/GMemory`. During the Memory Collection Phase, trajectories with task descriptions are first provided to the query graph. They then traverse upward to the insight graph to extract high-level insights, and downward to the interaction graph to identify core interactions within the trajectories, forming the three-tier graph memory. During the Deployment Phase, a task description is given to the query graph to retrieve the Top-1 most similar trajectory. The retrieved trajectory is then traversed upward and downward to identify the top-3 most similar insights and core interactions. The retrieved insights and core interactions are provided as memory for the agentic system.

B.3 Additional Details of Learning Process

ALMA is initialized with a memory design generated from scratch by the Meta Agent powered by `GPT-5/medium`. The generation of memory design follows Python Abstract Classes (Appendix A.1) and serves as the starting point for subsequent exploration. During generation, the Meta Agent is provided with `Provided Tools` containing minimal usage examples for invoking different agent roles (e.g., reasoning or conversational models), as well as embedding functions and database operations.

Starting from this initial memory design, ALMA performs 11 learning steps. At each step, memory designs are sampled from the archive, and the Meta Agent proposes, implements, and evaluates new memory designs. Each evaluated new design, together with its performance metrics and interaction logs, is then added to the memory design archive. This process repeats with the updated archive until the maximum number of learning iterations is reached.

Sampling. We perform the sampling process (Appendix A.4) with $\alpha = 0.5$ to balance the visit-time penalty, and $T = 0.5$ to trade off between distribution smoothness and the probability of selecting higher-performing memory designs. Sampling is performed without replacement to maintain diversity in the selected subset while still favoring higher-probability designs, ensuring effective exploration and evaluation efficiency. As the memory design archive grows to more memory designs (e.g., 30 to 40), and only a small number of designs (five) are sampled at each step, the difference

between sampling with and without replacement is expected to be negligible. Moreover, the purpose of sampling is to encourage exploration of memory designs that achieve high success rates but have been visited infrequently, while preserving a non-zero probability of sampling any memory design in the archive (Appendix A.4). This is achieved through a weighted sampling scheme that balances performance and visitation frequency. Under this objective, both sampling with and without replacement are suitable, as they preserve the intended probability structure and maintain effective exploration behavior. At each step, we sample up to five memory designs and perform parallel exploration of new memory designs.

Greedy Search. As an alternative to sampling in the learning process, we perform greedy search, always selecting the best memory design from the memory design archive at each learning step instead of sampling across all designs. This process continues until the total number of proposed designs matches that of the open-ended exploration. In our case, we perform 43 steps. Finally, the best memory design in the memory design archive is selected as the learned design produced by greedy search. The results of greedy search are shown in Section C.3.

Ideate & Planning. The ideate & planning procedure utilizes sampled memory designs along with their interaction logs to propose ideas and plans for new memory designs. The system prompt for the procedure remains fixed across all benchmarks and throughout the learning process (Appendix A.5). For each sampled memory design, the relevant input information includes: (1) the source code of the current sampled memory design; (2) stratified-sampled interaction logs produced during the evaluation of the current sampled design; and (3) the overall success rate of the current sampled memory design. Additionally, we include the following components to adapt to different benchmarks and situations for planning: (1) a one-shot example of plan proposed for a previous memory design, together with the associated performance of the previous design; (2) a benchmark description, depending on the benchmark being learned. The full benchmark description prompt is provided in `Benchmark Description`.

Benchmark Description

```
TASK_DESCRIPTION = {
      'alfworld': """The evaluation of downstream agent system is
         based on TextWorld:
- The agent explores within a given space (rooms) and a predefined
    list of actions in order to accomplish a specified task goal.
- Different data may involve **different rooms/spaces**, **different
    object placements**, or **entirely different task objectives**.
- The memory structure should strenthen following abilities of agent:
   - Spatial and State Reasoning: what are possible actions for each
       type of object?
   - Object-Centered Trajectory Learning: what did the agent do for
       previous same object management tasks or same location tasks?
""",
   'minihack':"""The evaluation of downstream agent system is based on
        MiniHack:
   - MiniHack is a collection of small game-like tasks where an agent
       sees a grid map of the environment, text and color
       representations, its own stats like health and position, and
       its inventory.
   - Different games will have **different goals, maps and
       observations**. So goals and initial observations can change
       drastically across episodes, requiring **transferable knowledge
       **.
   - The memory structure should strenthen following abilities of
       agent:
      - Spatial and State Reasoning: Infer the environment from
          partial observations, remember explored areas, and navigate
          the map effectively.
      - Long-Horizon Goal Planning: Form multi-step strategies and
          consistently act toward the task objective despite sparse or
           delayed rewards.
```

```
    - Environmental Interaction and Risk Management: Correctly use
        actions and make safe, context-aware decisions around
        monsters, traps, and terrain.
""",
'textworld':"""The evaluation of downstream agent system is based
    on TextWorld:
- The agent explores within a given space (rooms) and actions in
    order to accomplish a specified task goal. The agent may need
    to explore the space and get more progession.
- Done the task not neccessarily means the agent win the game.
    Whether wining or not depends on final reward(a progression
    percentage).
- Different data may involve **different rooms/spaces**, **
    different object placements**, or **entirely different task
    objectives**.
""",
'babaisai':"""The evaluation of downstream agent system is based on
     BABAisAI:
- In this gridworld game, agent interact with various objects and
    textual rule blocks to achieve specific goals.
- Different data may involve **different rooms/spaces**, **
    different object placements**.
- The agent goal is usually to reach an object defined as 'WIN',
    but this can be changed by changing the rules.
- The rules of the game can be manipulated and rearranged by the
    agent, creating a dynamic environment where agents must
    identify relevant objects and rules and then manipulate them to
     change or create new rules to succeed.
- The memory structure should strenthen following abilities of
    agent:
    - ability to perceive all objects and text blocks in the
        environment, understand which rules are currently active or
        blocked, and evaluate how the environment state affects
        potential winning conditions.
    - ability to analyze and determine whether it is possible to win
         by modifying rule text blocks, and identify which rule
        combinations could achieve the goal.
    - ability to plan multi-step actions, including moving the
        character and pushing text blocks, to gradually construct or
         alter rules that lead to victory.
    - ability to flexibly adapt its strategy in response to dynamic
        rule and environment changes, handling potential obstacles
        or conflicts, and learn when modifying rules is more
        effective than simple movement.
    - ability to possess exploration and strategy transfer
        capabilities, discovering new ways to manipulate rules in
        unseen levels and applying prior experience to recognize
        when rule changes are advantageous.
""",
'babyai':"""The world is a 2D grid with rooms, doors, walls, and
    objects such as keys, balls, and boxes:
- Each episode is procedurally generated, so layouts, room shapes,
    and object placements vary significantly between episodes.
- The agent has a limited field of view (a small square area in
    front of it).
- Observations are egocentric, meaning they depend on:
    * Agent position: moving to a different tile reveals a different
         portion of the map.
    * Agent facing direction: turning left or right changes what is
        visible, even without moving.
- Each episode provides a mission text (goal).
"""
}
```

The output reflection and plans for the new memory design follows below schema:

Planning Schema

```
MEMO_ANALYSIS_OUTPUT_FORMAT = {
      "learned_from_suggestion_example": {
      "type": "string",
      "description": "Findings derived from the provided
          suggestion_example and improve_score. Bullet list of
          concrete factors (patterns) that made the suggestion succeed
           or fail. And principles to adopt when making future
          suggestions."
   },
   "trajectory_score_assessment": {
      "type": "array",
      "description": "analysis each retrieved module information based
           on current trajectories sampled and the benchmark scores.",
      "items": {
         "type": "object",
         "properties": {
            "label": {
               "type": "string",
               "enum": ["Useful", "Potentially Useful", "Irrelevant",
                   "Empty/BadFormat"],
               "description": "Categorization of the memory item's
                   relevance based on whether the retrieved content
                   actually helps the agent."
            },
            "how_it_can_help": {
               "type": "string",
               "description": "If Useful/Potentially Useful: short
                   note how it could guide actions (subgoal, trap,
                   object use...). If Irrelevant/Empty: reason (e.g.,
                   wrong keying, over-specific, missing summary,
                   formatting)."
            }
         },
         "required": ["label", "how_it_can_help"]
      }
   },
   "content_quality_issues": {
      "type": "string",
      "description": "Detected content-level problems (duplicates,
          empty entries, serialization issues...). Why those harms
          retrieval or downstream planning.",
   },
   "structure_and_coherence": {
      "type": "string",
      "description": "Analysis of layer interactions, keying, and task
          -awareness. Which parts generalize, which are overfitted.",
   },
   "suggested_changes": {
      "type": "array",
      "description": "Based on all your analysis above, provide
          concrete change that can be applied on provided current
          memory structure code.",
      "items": {
         "type": "object",
         "properties": {
            "priority": {"type": "string", "enum": ["High", "Medium",
                "Low"], "description": "How urgent/impactful this is
                ."},
```

```
            "what": {"type": "string", "description": "Precise
                description of what to change (code/pipeline/config)
                ."},
            "why": {"type": "string", "description": "Link to
                observations/principles: why this addresses the
                problem."}
        },
        "required": ["priority", "what", "why"]
    }
  }
}
```

Implementation. The Meta Agent implements new memory design using the proposed plan. Specifically, we provide several components to the Meta Agent for implementation: (1) `trajectory_score_assessment` and `suggested_changes` in proposed plan, serve as instructions for implementation; (2) a one-shot trajectory example depending on the benchmarks as a reference format of possible input of the memory design; (3) Python Abstract Classes, which serve as the reference template for memory design implementation (Appendix A.1); and (4) provided tools for efficient implementation of new memory designs, as shown in `Provided Tools`.

Provided Tools

```
CHROMA_CHEETSHEET = """## Initialize Chroma DB

Use db = Chroma(embedding_function=Embedding()) to create the database
    . DO NOT use persist_dir.

### Add Memory: Adds new text entries to the database and returns
    their unique IDs.

db.add_texts(
   texts: List[str],
   metadatas: Optional[Union[str, int, float, bool, None]] = None,
   ids: Optional[List[str]] = None
) -> List[str]

- metadatas must be **flat list**: each value must be a single
    primitive type (str, int, float, bool, or None).
- You cannot pass lists, nested dicts, or other complex objects.
- If you need to store structured data, serialize it to a JSON string:

### Retrieve Memory

db.similarity_search(
   query: str,
   k: int = 4
) -> List[Document]

return List[Document]: [
  Document(
    page_content="the agent found a key",
    metadata={"type": "item"}
  )
]

### Get by ID
db.get(
   ids: Optional[List[str]] = None
) -> Dict[str, List]

### Delete Memory
```

```
db.delete(
   ids: Optional[List[str]] = None
) -> None
   """
NXGRAPH_CHEETSHEET = """
NETWORKX GRAPH CHEATSHEET

Context:
   import networkx as nx
   G = nx.Graph()

1. NODE OPERATIONS
- G.add_node(node, **attrs): Add a single node with optional
    attributes.
- G.add_nodes_from([n1, n2], **common_attrs): Add multiple nodes at
    once (shared attributes apply to all).
- G.remove_node(node): Remove a node and all edges connected to it.
- G.remove_nodes_from([n1, n2]): Remove multiple nodes.
- node in G: Check if a node exists.
- G.nodes: Get all nodes (NodeView).
- G.nodes[node]: Access node attributes as a dict.
- nx.set_node_attributes(G, {node: {"attr": value}}): Set attributes
    for nodes.

2. EDGE OPERATIONS
- G.add_edge(u, v, **attrs): Add an edge between two nodes.
- G.add_edges_from([(u, v), (x, y)], **attrs): Add multiple edges at
    once (shared attributes apply to all).
- G.remove_edge(u, v): Remove a single edge.
- G.remove_edges_from([(u, v), (x, y)]): Remove multiple edges.
- G.has_edge(u, v): Check if an edge exists.
- G.edges: Get all edges (EdgeView).
- G.edges[(u, v)]: Access edge attributes as a dict.
- nx.set_edge_attributes(G, {(u, v): {"weight": 1.0}}): Set attributes
     for edges.

3. TRAVERSAL / NEIGHBORHOOD
- G.neighbors(node): Get neighbors of a node.
- G.adj[node]: Get dict of neighbors with edge data.
- nx.shortest_path(G, source, target): Find one shortest path between
    nodes.
- nx.shortest_path_length(G, source, target): Get shortest path length
    .
- nx.all_simple_paths(G, source, target, cutoff): Generate all simple
    paths up to a cutoff length.
- nx.connected_components(G): Get connected components as node sets.
- G.subgraph([n1, n2, n3]): Extract a subgraph induced by given nodes.

4. ANALYSIS / CENTRALITY
- G.degree(node): Get degree (number of edges) for a single node.
- G.degree(): Get degree for all nodes (DegreeView).
- nx.degree_centrality(G): Compute degree centrality (dict of node ->
    score).
- nx.betweenness_centrality(G): Compute betweenness centrality.
- nx.pagerank(G): Compute PageRank scores for nodes.
- nx.clustering(G): Compute local clustering coefficient.
- nx.is_connected(G): Check if graph is connected.
- nx.number_connected_components(G): Count connected components.

5. UTILITIES
- G.copy(): Make a copy of the graph.
- G.clear(): Remove all nodes and edges.
- nx.to_dict_of_dicts(G): Convert graph to adjacency dict.
```

```
- nx.to_numpy_array(G): Get adjacency matrix as a NumPy array.
"""
TOOL_CHEETSHEET = """
TOOLS AVAILABLE:

1. Hire Agent
Class: Agent

- Purpose: Asynchronous wrapper around OpenAI Chat API, allows system/
    user prompts and optional JSON schema validation. Higher insights:
     this could be used to summarise information or gain new insights
    from them.
- Initialization:
   from utils.hire_agent import Agent
   agent = Agent(model: str, system_prompt: str, output_schema:
       Optional[Dict] = None)
- Key Methods:
   - agent.get_agent_config() -> Dict: Returns agent configuration and
        chat history.
   - await agent.ask(user_input: str, with_history: bool = False) ->
       Any:
      Sends a user message asynchronously and returns the model's
          response.
      If an output_schema is provided, response will be validated
          against it and returned as a JSON object.
- Usage Example:
   response = await agent.ask("Hello, how are you?")

-IMPORTANT:
   - output_schema must be a **json schema**. Format example:
{
   "location": {
      "type": "string",
      "description": "The location to get the weather for"
   },
   "unit": {
      "type": ["string", "null"],
      "description": "The unit to return the temperature in",
      "enum": ["F", "C"]
   }
}

   - model could be "gpt-4.1", "gpt-4o-mini", based on whether you
       need reasoning ability(if reasoning ability is needed, gpt-4o-
       mini could be a better choice).

2. Embedding
Class: Embedding

- Purpose: Async embedding manager for computing single or batch
    embeddings, with optional similarity calculation. Higher insights:
     this could be used to find connections between texts.
- Initialization:
   from utils.hire_agent import Embedding
   embedder = Embedding(model: str = "text-embedding-3-small", retries
       : int = 3, retry_delay: float = 1.0)
- Key Methods:
   - await embedder.get_embedding(text: str) -> List[float]:
      Computes embedding for a single text string asynchronously.
   - await embedder.get_batch_embeddings(texts: List[str]) -> List[
       List[float]]:
      Computes embeddings for multiple texts asynchronously.
```

```
    - await Embedding.compute_similarity(emb1: List[float], emb2: List[
        float], metric: str = "cosine") -> float:
       Computes similarity between two embeddings asynchronously.
    - await Embedding.compute_one_to_group_similarity(emb: List[float],
         group_emb: List[List[float]], metric: str = "cosine") -> List[
        float]:
       Computes similarity between one embedding and a group of
           embeddings asynchronously.

  - Notes:
    * All embedding calls automatically update a global token tracker.
    * Similarity functions support cosine similarity and run in
        parallel for efficiency.
  """
```

As shown in `Provided Tools`, the Meta Agent is provided with the option of using `GPT-4o-mini` or `GPT-4.1` as FMs within the memory design, for example, when extracting insights from interaction trajectories. By explicitly including multiple model options (e.g., reasoning model or chat model) in the search space, the Meta Agent can adaptively select the most suitable model based on different reasoning and implementation requirements during memory design construction. During testing, however, we consistently use `GPT-4o-mini` for all memory designs to ensure a fair comparison under matched model capacity.

We then perform a trial run to validate the implementation of the new memory design. The trial run samples 5 tasks from the learning set randomly, using 2 tasks for the Memory Collection Phase and 3 tasks for the Deployment Phase. If a runtime error occurs during the trial run, the Meta Agent engages in a reflection and debugging process, after which, the trial run is repeated on the debugged implementation. This loop of reflection, debugging, and trial runs iterates up to 3 times to ensure that the memory design implementation is free of bugs.

Evaluation. We integrate the new memory design with the agentic system powered by `GPT-5-nano/low`. We perform `general_update` on each trajectory during the Memory Collection Phase. During the Deployment Phase, we use `general_retrieve` to take the task description as input to retrieve relevant memory. The learning set is evenly split between the two phases. All tasks in the Deployment Phase are executed three times to compute the average success rate, reducing bias caused by variance in individual runs. For dataset splits, we use the first 30 tasks from the *train* set in ALFWorld as the learning set. For TextWorld and Baba Is AI, the datasets are evenly split into learning and testing sets. For MiniHack, 30% of the dataset is used as the learning set. The datasets for TextWorld, Baba Is AI, and MiniHack are sourced from BALROG (Paglieri et al., 2025). We perform stratified sampling (Thompson, 2012) to extract interaction logs after the Deployment Phase and store extracted logs in the memory design archive. When a memory design is sampled in future learning steps, the stratified logs allow the Meta Agent to reflect on both successes and failures without needing to examine all interaction logs.

B.4 Additional Details of Testing

After learning memory designs on each benchmark, we perform testing under both static and dynamic modes for ALFWorld. For other benchmarks, we perform testing under static mode. For TextWorld and Baba Is AI, we use the second halves of the datasets for testing, respectively. For MiniHack, we use the remaining 70% of the dataset for testing. All testing sets are split evenly for the Memory Collection and the Deployment Phase.

Static Mode. We first evaluate the best-learned memory designs and manual baselines on each benchmark using the agentic system powered by `GPT-5-nano/low`, the same model used during the learning process, to assess the effectiveness of the learned designs. We then test these memory designs with the agentic system powered by `GPT-5-mini/medium` to evaluate their transferability to a more capable agentic system. Additionally, for the ALFWorld *valid_seen* dataset, we test the learned best memory design alongside manual baselines while varying the number of trajectories used in the Memory Collection Phase from 0 to 70, using `GPT-5-mini/medium` as the foundational model in the agentic system.

Dynamic Mode. We evaluate the learned best memory design alongside manual baselines on ALFWorld using the agentic system powered by `GPT-5-mini/medium`. The first 70 tasks from the *valid_seen* dataset are used to generate trajectories for the Memory Collection Phase, while tasks from the *valid_unseen* dataset are used in the Deployment Phase, creating a gap between collected memory and tasks to be solved. During the Deployment Phase, the agentic system uses `general_retrieve()` to access relevant memory for each new task. After completing a task, the interaction log is incorporated via `general_update()`, producing a dynamic memory that evolves as the agentic system sequentially solves tasks.

C Result Details

C.1 Cost Efficiency of Memory Designs

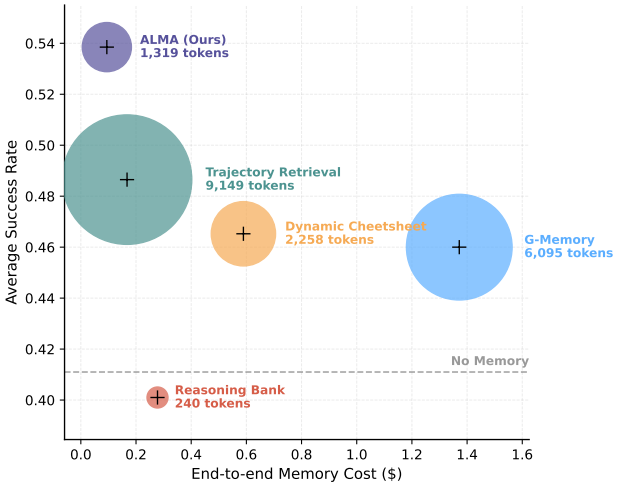

Figure 9: Cost efficiency versus task performance across all benchmarks for different memory designs. The x-axis shows the overall end-to-end memory cost across benchmarks. Bubble size represents the average token size of retrieved content inserted into the agentic system's context per task. The learned memory design outperforms all baselines while being cost-efficient.

C.2 Learning Processes on Other Benchmarks

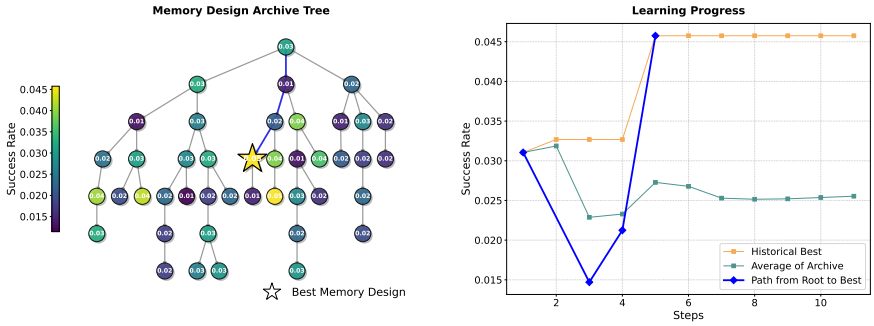

Figure 10: The learning process of ALMA on TextWorld, using `GPT-5-nano` as the FM in an agentic system. **Left:** The memory design archive tree, where each node represents a memory design. Node colors indicate the success rate, and edges indicate that each child node is derived from its parent. The memory design with the highest success rate is used as the final learned memory design. **Right:** The step-wise learning progress. ALMA progressively discovers memory designs by building on an ever-growing archive of previous discoveries. The path from the root memory design to the best memory design highlights the importance of open-ended exploration, where designs with moderate success rates serve as stepping stones toward optimal solutions.

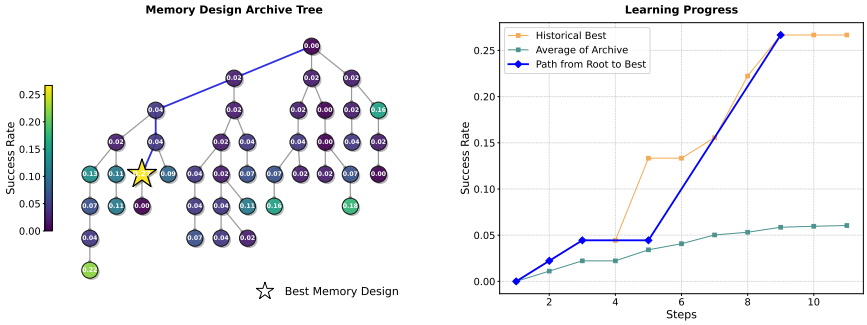

Figure 11: The learning process of ALMA on ALFWorld, using `GPT-5-nano` as the FM in an agentic system. **Left:** The memory design archive tree, where each node represents a memory design. Node colors indicate success rate, and edges indicate that each child node is derived from its parent. The memory design with the highest success rate is used as the final learned memory design. **Right:** The step-wise learning progress. ALMA progressively discovers memory designs by building on an ever-growing archive of previous discoveries. The path from the root memory design to the best memory design highlights the importance of open-ended exploration, where designs with moderate success rates serve as stepping stones toward optimal solutions.

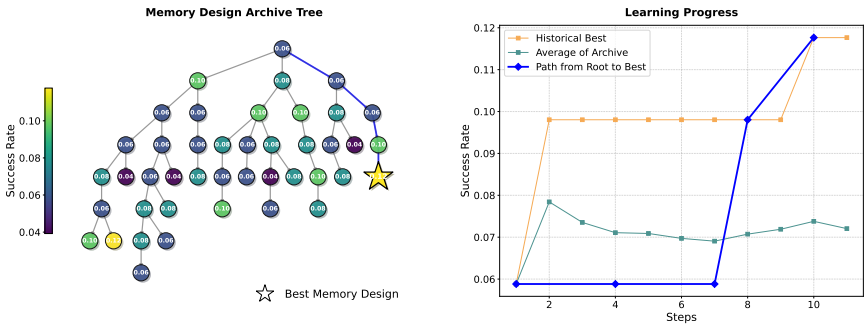

Figure 12: The learning process of ALMA on MiniHack, using `GPT-5-nano` as the FM in an agentic system. **Left:** The memory design archive tree, where each node represents a memory design. Node colors indicate success rate, and edges indicate that each child node is derived from its parent. The memory design with the highest success rate is used as the final learned memory design. **Right:** The step-wise learning progress. ALMA progressively discovers memory designs by building on an ever-growing archive of previous discoveries. The path from the root memory design to the best memory design highlights the importance of open-ended exploration, where designs with moderate success rates serve as stepping stones toward optimal solutions.

C.3 Results of Greedy Exploration

We also conduct a greedy memory design sampling baseline on ALFWorld, where at each learning step we only select the historically best-performing memory design for further exploration (Appendix B.3). To ensure a fair comparison, this greedy strategy is evaluated under the same exploration budget, i.e., it explores the same total number of memory designs as our method. The resulting exploration tree is shown in fig. 13.

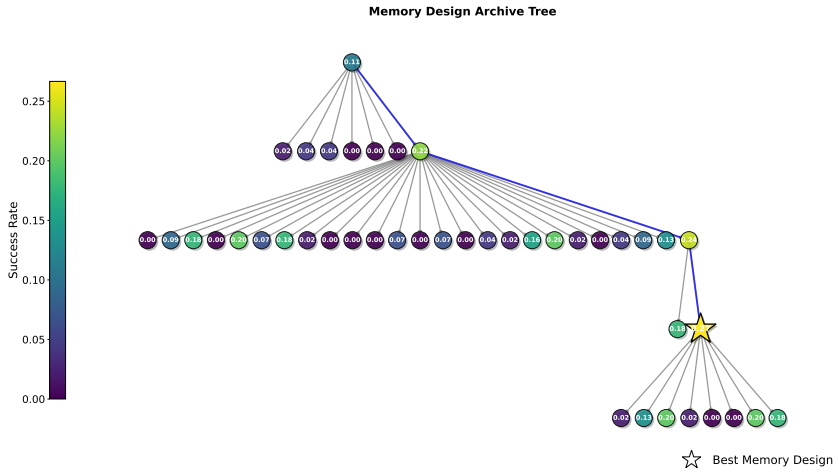

Figure 13: The memory design archive tree of the greedy learning process on ALFWorld, using `GPT-5-nano` as the FM in the agentic system. Each node represents a memory design. Node colors indicate success rate, and edges indicate that each child node is derived from its parent. The memory design with the highest success rate is used as the final learned memory design. In the greedy learning process, each new memory design is developed based on the previous design that achieved the highest success rate.

ALMA produces learned memory designs that achieve better performance and transferability (table 2) compared to Greedy Search, highlighting the importance of open-ended exploration patterns to explore better designs.

Table 2: Comparison of success rate (Mean $\pm$ SE) in percentage using Greedy Search and ALMA on ALFWorld. The standard error is calculated over three runs of the Deployment Phase. Testing is conducted with `GPT-5-nano` and `GPT-5-mini` as foundation models in the agentic system.

Foundation Model	Greedy Search	ALMA (Ours)
GPT-5-nano	11.9 ± 0.5	$\mathbf{12.4 \pm 0.5}$
GPT-5-mini	77.1 ± 0.8	$\mathbf{87.1 \pm 1.4}$

C.4 EXAMPLE RESULTS OF LEARNED MEMORY DESIGNS

Listing 2: The Best-learned memory design in MiniHack.

```
import asyncio
import json
import re
import uuid
import hashlib
from typing import Any, Dict, List, Optional, Tuple
from dataclasses import dataclass, field

import networkx as nx

from agents.memo_structure import Sub_memo_layer, MemoStructure
from eval_envs.base_envs import Basic_Recorder
from utils.hire_agent import Agent, Embedding
from langchain_chroma import Chroma


# ----------------------------- Utilities
    ---------------------------------


def _normalize_space(text: str) -> str:
    return re.sub(r"\s+", " ", text or "").strip()


def _truncate(text: Optional[str], n: int = 600) -> Optional[str]:
    if text is None:
        return None
    t = str(text)
    return t if len(t) <= n else t[: n - 3] + "..."


def _clean_list_of_dicts(
    items: List[Dict[str, Any]], drop_none: bool = True,
        truncate_keys: Optional[List[str]] = None
) -> List[Dict[str, Any]]:
    truncate_keys = truncate_keys or []
    cleaned = []
    for it in items:
        c = {}
        for k, v in it.items():
            if drop_none and (v is None or v == "" or v == [] or v ==
                 {}):
                continue
            if isinstance(v, str) and k in truncate_keys:
                c[k] = _truncate(v, 600)
            else:
                c[k] = v
        if c:
            cleaned.append(c)
    return cleaned


def _extract_map_area(long_term_context: str) -> Tuple[int, int]:
    if not long_term_context:
        return 0, 0
    lines = long_term_context.splitlines()
    map_started = False
    map_lines: List[str] = []
    for line in lines:
        if map_started:
            if re.search(r"St:\d", line) or "HP:" in line:
```

```
                break
            else:
                map_lines.append(line.rstrip("\n"))
        if line.strip().lower().startswith("map:"):
            map_started = True
    if not map_lines:
        return 0, 0
    height = len(map_lines)
    width = max(len(l) for l in map_lines) if map_lines else 0
    return width, height


def _extract_position(long_term_context: str) -> Optional[Tuple[int,
    int]]:
    if not long_term_context:
        return None
    cursor_match = re.search(r"\(x\s*=\s*(\d+),\s*y\s*=\s*(\d+)\)",
        long_term_context)
    if cursor_match:
        return int(cursor_match.group(1)), int(cursor_match.group(2))
    return None


def _extract_stats_line(long_term_context: str) -> str:
    if not long_term_context:
        return ""
    for line in reversed(long_term_context.splitlines()):
        if "HP:" in line:
            return line.strip()
    return ""


def _extract_action_set(action_dict: Dict[str, str]) -> List[str]:
    if not action_dict:
        return []
    return sorted(set(action_dict.keys()))


def _parse_role_race_alignment(long_term_context: str) -> Dict[str,
    Optional[str]]:
    # Example greeting: "You are a chaotic female human Monk."
    line = ""
    for l in long_term_context.splitlines():
        if "welcome to nethack" in l.lower() or "you are a" in l.
            lower():
            line = l
            break
    ali = None
    race = None
    role = None
    if line:
        # alignment
        m = re.search(r"\b(chaotic|neutral|lawful)\b", line, re.
            IGNORECASE)
        if m:
            ali = m.group(1).lower()
        # race
        m2 = re.search(r"\b(human|elf|dwarf|gnome|orc|gnomish)\b",
            line, re.IGNORECASE)
        if m2:
            race = m2.group(1).lower()
            race = "gnome" if race == "gnomish" else race
        # role: last capitalized word at end or after race
        m3 = re.search(
```

```
            r"\b(human|elf|dwarf|gnome|orc|gnomish)\b\s+([A-Za-z\-]+)
                \.?$",
            line,
            re.IGNORECASE,
        )
        if m3:
            role = m3.group(2).lower()
        else:
            m4 = re.search(r"\b([A-Za-z\-]+)\.?$", line)
            if m4:
                role = m4.group(1).lower()

    # Fallback role from cursor section
    cursor_section = []
    capture_cursor = False
    for l in long_term_context.splitlines():
        if l.strip().lower().startswith("cursor:"):
            capture_cursor = True
            continue
        if capture_cursor:
            cursor_section.append(l.strip())
            if l.strip() == "":
                break
    cursor_text = " ".join(cursor_section).lower()
    mcur = re.search(r"yourself a ([a-z\-]+)", cursor_text)
    if mcur and not role:
        role = mcur.group(1)

    return {"role": role, "race": race, "alignment": ali}


_CANONICAL_ENTITIES = {
    "bars": "wall",
    "wall": "wall",
    "walls": "wall",
    "door": "door",
    "doors": "door",
    "closed door": "door",
    "open door": "door",
    "boulder": "boulder",
    "boulders": "boulder",
    "fountain": "fountain",
    "fountains": "fountain",
    "altar": "altar",
    "altars": "altar",
    "sink": "sink",
    "sinks": "sink",
    "trap": "trap",
    "traps": "trap",
    "stairs up": "stairs_up",
    "stairs down": "stairs_down",
    "staircase up": "stairs_up",
    "staircase down": "stairs_down",
}


def _normalize_dir_token(tok: str) -> Optional[str]:
    w = tok.lower()
    dirs = ["north", "east", "south", "west"]
    has = {d: (d in w) for d in dirs}
    if has["north"] and has["east"]:
        return "northeast"
    if has["north"] and has["west"]:
        return "northwest"
    if has["south"] and has["east"]:
```

```
        return "southeast"
    if has["south"] and has["west"]:
        return "southwest"
    if has["north"]:
        return "north"
    if has["east"]:
        return "east"
    if has["south"]:
        return "south"
    if has["west"]:
        return "west"
    return None


def _extract_local_topology(long_term_context: str) -> Dict[str, List
    [str]]:
    """
    Parse the 'language observation' section to infer local
        directional topology.
    Returns lists of directions for canonical entities.
    """
    if not long_term_context:
        return {
            "walls_at": [],
            "doors_at": [],
            "stairs_up_at": [],
            "stairs_down_at": [],
            "boulder_at": [],
            "fountain_at": [],
            "other": "{}",
        }
    lines = long_term_context.splitlines()
    capturing = False
    section = []
    for line in lines:
        if line.strip().lower().startswith("language observation"):
            capturing = True
            continue
        if line.strip().lower().startswith("cursor:"):
            break
        if capturing:
            section.append(line)

    entity_dirs: Dict[str, set] = {}
    for raw in section:
        low = raw.lower()
        if "stairs up" in low or "staircase up" in low:
            dirs = []
            for t in re.findall(r"([A-Za-z]+)", low):
                norm = _normalize_dir_token(t)
                if norm:
                    dirs.append(norm)
            entity_dirs.setdefault("stairs_up", set()).update(dirs)
        if "stairs down" in low or "staircase down" in low:
            dirs = []
            for t in re.findall(r"([A-Za-z]+)", low):
                norm = _normalize_dir_token(t)
                if norm:
                    dirs.append(norm)
            entity_dirs.setdefault("stairs_down", set()).update(dirs)

        # general entities
        for k, canon in _CANONICAL_ENTITIES.items():
            if k in low:
                dirs = []
```

```
                for t in re.findall(r"([A-Za-z]+)", low):
                    norm = _normalize_dir_token(t)
                    if norm:
                        dirs.append(norm)
                if dirs:
                    entity_dirs.setdefault(canon, set()).update(dirs)

    def to_sorted_list(key: str) -> List[str]:
        return sorted(list(entity_dirs.get(key, set())))

    topology = {
        "walls_at": to_sorted_list("wall"),
        "doors_at": to_sorted_list("door"),
        "stairs_up_at": to_sorted_list("stairs_up"),
        "stairs_down_at": to_sorted_list("stairs_down"),
        "boulder_at": to_sorted_list("boulder"),
        "fountain_at": to_sorted_list("fountain"),
    }
    known = {"wall", "door", "stairs_up", "stairs_down", "boulder", "
        fountain"}
    other = {k: sorted(list(v)) for k, v in entity_dirs.items() if k
        not in known}
    topology["other"] = json.dumps(other, ensure_ascii=False)
    return topology


def _canonicalize_entities(long_term_context: str) -> List[str]:
    if not long_term_context:
        return []
    section = []
    lines = long_term_context.splitlines()
    capturing = False
    for line in lines:
        if "No  Points     Name" in line:
            break
        if line.strip().lower().startswith("language observation"):
            capturing = True
            continue
        if line.strip().lower().startswith("cursor:"):
            break
        if capturing:
            section.append(line.lower())
    text = " ".join(section)
    ents = set()
    for k, v in _CANONICAL_ENTITIES.items():
        if k in text:
            ents.add(v)
    if re.search(r"\bmonster|monsters|you see a [a-z]+", text):
        ents.add("monster")
    return sorted(ents)


def _normalize_item_name(name: str) -> str:
    # common normalizations for higher-quality retrieval keys
    mapping = {
        "fortune cookies": "fortune cookie",
        "food rations": "food ration",
        "tooled horn": "horn",
        "sprig of wolfsbane": "wolfsbane",
        "cloves of garlic": "garlic",
        "pair of gloves": "gloves",
        "leather gloves": "gloves",
        "robe": "robe",
        "potion of healing": "potion of healing",
        "scroll of identify": "scroll of identify",
```

```
        "spellbook of protection": "spellbook of protection",
        "spellbook of identify": "spellbook of identify",
        "fedora": "fedora",
        "fedor": "fedora",
        "potions of extra healing": "potion of extra healing",
        "scrolls of magic mapping": "scroll of magic mapping",
        "hawaiian shirt": "hawaiian shirt",
        "expensive camera": "camera",
        "camera": "camera",
        # melee/ranged common
        "katana": "katana",
        "wakizashi": "wakizashi",
        "yumi": "bow",
        "ya": "arrow",
        "arrows": "arrow",
        "arrow": "arrow",
        "bow": "bow",
    }
    # handle common truncations/typos
    name = name.replace("extrhealing", "extra healing").replace("
        camer", "camera")
    name = re.sub(r"^\+\s*", "", name)  # strip leading + tokens
    if name in mapping:
        return mapping[name]
    if name.endswith("s") and " of " not in name:
        singular = name[:-1]
        return mapping.get(singular, singular)
    return name


def _simplify_inventory_items(short_term_context: str) -> List[str]:
    if not short_term_context:
        return []
    items: List[str] = []
    for line in short_term_context.splitlines():
        m = re.match(r"^[a-zA-Z]:\s+(.*)$", line.strip())
        if not m:
            continue
        raw = m.group(1).lower()

        # Remove annotations and counts
        raw = re.sub(r"\(.*?\)", "", raw)
        raw = re.sub(r"\b(blessed|uncursed|cursed)\b", "", raw)
        raw = re.sub(r"\b\+\d+\b", "", raw)
        raw = re.sub(r"^\s*\+\s*", "", raw)
        raw = re.sub(r"\b\d+\b", "", raw)

        # Standardize determiners and multiword patterns
        raw = raw.replace("pair of ", "")
        raw = raw.replace("an ", "").replace("a ", "").replace("the "
            , "")

        # Normalize known weapon-ammo strings
        if "yumi" in raw:
            items.append("bow")
            continue
        # Nethack 'ya' are arrows; sometimes rendered oddly like '+ y
            '
        if re.search(r"\bya\b", raw) or re.search(r"\b\+\s*y\b", raw)
             or raw.strip() in {"ya", "y", "+ y"}:
            items.append("arrow")
            continue

        raw = _normalize_space(raw)
```

```
        # Prefer item-of-type phrases
        m2 = re.search(r"(wand|ring|potion|scroll|spellbook) of [a-z
            \- ]+", raw)
        if m2:
            item = _normalize_space(m2.group(0))
        else:
            m3 = re.search(r"([a-z]+ of [a-z\- ]+)", raw)
            if m3:
                item = _normalize_space(m3.group(1))
            else:
                toks = [t for t in re.split(r"[^a-z]+", raw) if t]
                if len(toks) >= 2:
                    item = _normalize_space(" ".join(toks[-2:]))
                elif toks:
                    item = toks[-1]
                else:
                    continue

        item = _normalize_item_name(item)
        # Drop stray single letters and plus signs
        if len(item) == 1 and not item.isalpha():
            continue
        if len(item) == 1 and item.isalpha() and item not in {"y"}:
            continue
        if item == "y":
            item = "arrow"

        items.append(item)

    seen = set()
    out = []
    for it in items:
        if it not in seen and it:
            seen.add(it)
            out.append(it)
    return out[:12]


# ----------------- ASCII map parsing and neighbor semantics
    ----------------


_DIRS = {
    "north": (0, -1),
    "northeast": (1, -1),
    "east": (1, 0),
    "southeast": (1, 1),
    "south": (0, 1),
    "southwest": (-1, 1),
    "west": (-1, 0),
    "northwest": (-1, -1),
}

_DIR_CODES = {
    "north": "N",
    "northeast": "NE",
    "east": "E",
    "southeast": "SE",
    "south": "S",
    "southwest": "SW",
    "west": "W",
    "northwest": "NW",
}
```

```python
def _sign(v: int) -> int:
    return 0 if v == 0 else (1 if v > 0 else -1)


def _vector_to_dir(dx: int, dy: int) -> Optional[str]:
    dx = _sign(dx)
    dy = _sign(dy)
    for d, (vx, vy) in _DIRS.items():
        if vx == dx and vy == dy:
            return d
    return None


def _is_passable_char(ch: str) -> bool:
    # Treat '.' and '#' as traversable floors/corridors; '<' and '>'
        as traversable objectives.
    if ch in {".", "#", "<", ">"}:
        return True
    return False


def _is_hazard_char(ch: str) -> bool:
    # Hazard-like tiles we avoid by default
    return ch in {"`", "}", "~", "^", "L"}


def _entity_from_char(ch: str) -> Optional[str]:
    if ch == "{":
        return "fountain"
    if ch == "<":
        return "stairs_up"
    if ch == ">":
        return "stairs_down"
    if ch == "#":
        # in our semantics, '#' is passable corridor; not a wall
            entity
        return None
    return None


def _parse_ascii_map(long_term_context: str) -> Dict[str, Any]:
    if not long_term_context:
        return {"grid": [], "width": 0, "height": 0, "hero_xy": None,
             "neighbors": {}}
    lines = long_term_context.splitlines()
    map_started = False
    map_lines: List[str] = []
    for line in lines:
        if map_started:
            if "HP:" in line or re.search(r"St:\d", line):
                break
            map_lines.append(line.rstrip("\n"))
        if line.strip().lower().startswith("map:"):
            map_started = True
    if not map_lines:
        return {"grid": [], "width": 0, "height": 0, "hero_xy": None,
             "neighbors": {}}
    width = max(len(l) for l in map_lines)
    height = len(map_lines)
    grid = [l.ljust(width) for l in map_lines]
    pos = _extract_position(long_term_context)
    hero_xy = pos if pos else None

    # Robust calibration: check 3x3 around reported pos, then
        fallback scan
```

```
    def find_at_sign(grid_: List[str]) -> List[Tuple[int, int]]:
        coords = []
        for yy, row in enumerate(grid_):
            for xx, ch in enumerate(row):
                if ch == "@":
                    coords.append((xx, yy))
        return coords

    if hero_xy:
        hx, hy = hero_xy
        found = None
        # Local 3x3 search
        for dy in (-1, 0, 1):
            for dx in (-1, 0, 1):
                nx_, ny_ = hx + dx, hy + dy
                if 0 <= ny_ < height and 0 <= nx_ < width and grid[
                    ny_][nx_] == "@":
                    found = (nx_, ny_)
                    break
            if found:
                break
        if found:
            hero_xy = found
        else:
            # Global scan; choose nearest to reported
            at_positions = find_at_sign(grid)
            if at_positions:
                if len(at_positions) == 1:
                    hero_xy = at_positions[0]
                else:
                    hx0, hy0 = hx, hy
                    hero_xy = min(at_positions, key=lambda p: abs(p
                        [0] - hx0) + abs(p[1] - hy0))
            else:
                hero_xy = None
    else:
        # No reported cursor, try global scan
        at_positions = [(xx, yy) for yy, row in enumerate(grid) for
            xx, ch in enumerate(row) if ch == "@"]
        hero_xy = at_positions[0] if at_positions else None

    neighbors: Dict[str, str] = {}
    if hero_xy:
        hx, hy = hero_xy
        for d, (dx, dy) in _DIRS.items():
            nx_ = hx + dx
            ny_ = hy + dy
            if 0 <= ny_ < height and 0 <= nx_ < width:
                ch = grid[ny_][nx_]
            else:
                ch = " "  # out of bounds
            neighbors[d] = ch
    return {"grid": grid, "width": width, "height": height, "hero_xy"
        : hero_xy, "neighbors": neighbors}


def _neighbors_signature(neighbors: Dict[str, str]) -> str:
    if not neighbors:
        return ""
    ordered = ["north", "northeast", "east", "southeast", "south", "
        southwest", "west", "northwest"]
    parts = []
    for d in ordered:
        ch = neighbors.get(d, " ")
        code = _DIR_CODES[d]
```

```
        parts.append(f"{code}:{ch}")
    return "; ".join(parts)


def _safe_json_dumps(data: Any) -> str:
    try:
        return json.dumps(data, ensure_ascii=False)
    except Exception:
        return str(data)


def _get_default_agent(system_prompt: str, schema: Optional[Dict[str,
     Any]] = None) -> Agent:
    return Agent(model="gpt-4o-mini", system_prompt=system_prompt,
        output_schema=schema)


def _suggest_first_moves(
    actions: List[str],
    topology: Dict[str, List[str]],
    map_neighbors: Optional[Dict[str, str]] = None,
    passable_dirs: Optional[List[str]] = None,
    goal_text: str = "",
    hazard_dirs: Optional[List[str]] = None,
    nearest_objective_step: Optional[str] = None,
) -> Dict[str, Any]:
    one_step_dirs = {"north", "northeast", "east", "southeast", "
        south", "southwest", "west", "northwest"}
    allowed = [a for a in (actions or []) if a in one_step_dirs]

    hazard_dirs = hazard_dirs or []
    passable_dirs = passable_dirs or []
    prioritized: List[str] = []
    blocked_map = set()
    passable_map = set()

    # Map-based passability is primary
    if map_neighbors:
        for d in allowed:
            ch = map_neighbors.get(d, " ")
            if _is_passable_char(ch):
                passable_map.add(d)
            else:
                blocked_map.add(d)

    # If no map info, fallback to passable_dirs
    if not passable_map and passable_dirs:
        passable_map = set(passable_dirs)
    # Derive safe vs hazardous
    hazard_set = set(hazard_dirs)
    safe_passable = [d for d in allowed if d in passable_map and d
        not in hazard_set]
    hazardous_passable = [d for d in allowed if d in passable_map and
         d in hazard_set]

    # Objective targeting using neighbors
    objectives_chars = set()
    gt = goal_text.lower()
    if "stairs" in gt:
        objectives_chars.update({"<", ">"})
    if "boulder" in gt or "sokoban" in gt or "boxoban" in gt:
        objectives_chars.update(["{"])  # fountains as targets for
            boxoban-like
    if map_neighbors:
        for d in allowed:
```

```
            ch = map_neighbors.get(d, " ")
            if ch in objectives_chars and d in safe_passable and d
                not in prioritized:
                prioritized.append(d)

    # nearest objective step hint
    if nearest_objective_step and (nearest_objective_step in allowed)
         and nearest_objective_step not in prioritized:
        # Ensure it's considered early if safe; otherwise best-effort
             include
        if nearest_objective_step in safe_passable:
            prioritized.insert(0, nearest_objective_step)
        elif nearest_objective_step in passable_map:
            prioritized.append(nearest_objective_step)

    # Fill with remaining safe passable
    for d in safe_passable:
        if d not in prioritized:
            prioritized.append(d)

    # Only if no safe options exist, consider hazardous passable
    if not prioritized:
        for d in hazardous_passable:
            if d not in prioritized:
                prioritized.append(d)

    blocked_dirs = sorted(list(blocked_map))

    # Fallback non-movement actions if no good moves
    fallback_candidates = []
    for cand in ["search", "wait", "rest"]:
        if cand in (actions or []):
            fallback_candidates.append(cand)

    return {
        "blocked_dirs": blocked_dirs,
        "hazard_dirs": sorted(list(hazard_set)),
        "suggested_first_moves": prioritized[:4],
        "nearby_objectives": {
            "topology": {
                k: v
                for k, v in (topology or {}).items()
                if k in ["fountain_at", "stairs_up_at", "
                    stairs_down_at"] and v
            },
        },
        "fallback_actions": fallback_candidates if not prioritized
            else [],
    }


def _detect_blocked_move_message(text: str) -> bool:
    low = (text or "").lower()
    patterns = [
        "cannot pass through",
        "cannot move through",
        "it's solid stone",
        "you hit the wall",
        "there is a wall in the way",
        "you bump into",
        "blocked",
    ]
    return any(p in low for p in patterns)
```

```python
def _compute_oscillation(actions_seq: List[str]) -> Dict[str, Any]:
    opposites = {
        "north": "south",
        "south": "north",
        "east": "west",
        "west": "east",
        "northeast": "southwest",
        "southwest": "northeast",
        "northwest": "southeast",
        "southeast": "northwest",
    }
    toggles = 0
    pairs: Dict[str, int] = {}
    for i in range(2, len(actions_seq)):
        a0, a1, a2 = actions_seq[i - 2 : i + 1]
        if a0 and a1 and a2:
            if opposites.get(a0) == a1 and opposites.get(a1) == a2:
                toggles += 1
                key = f"{a0}<->{a1}"
                pairs[key] = pairs.get(key, 0) + 1
    score = toggles
    return {"score": score, "pairs": pairs}


def _neighbors_hazard_dirs(neighbors: Dict[str, str]) -> List[str]:
    if not neighbors:
        return []
    dirs = []
    for d, ch in neighbors.items():
        if _is_hazard_char(ch):
            dirs.append(d)
    return sorted(dirs)


def _bfs_first_step_to_targets(
    grid: List[str], width: int, height: int, start: Tuple[int, int],
         targets: set, radius: int = 5
) -> Optional[str]:
    if not start or not grid:
        return None
    from collections import deque

    sx, sy = start
    visited = set()
    q = deque()
    q.append((sx, sy, None))  # (x, y, first_move_dir)
    visited.add((sx, sy))

    def neighbors(x: int, y: int):
        for d, (dx, dy) in _DIRS.items():
            nx_, ny_ = x + dx, y + dy
            if 0 <= nx_ < width and 0 <= ny_ < height:
                yield d, nx_, ny_

    while q:
        x, y, first_dir = q.popleft()
        if abs(x - sx) + abs(y - sy) > radius:
            continue
        ch = grid[y][x]
        if ch in targets and not (x == sx and y == sy):
            return first_dir
        for d, nx_, ny_ in neighbors(x, y):
            if (nx_, ny_) in visited:
                continue
            chn = grid[ny_][nx_]
```

```
            # avoid hazards by default; allow stepping onto target
                even if hazard
            if (nx_, ny_) != (sx, sy) and _is_hazard_char(chn) and
                chn not in targets:
                continue
            if not _is_passable_char(chn) and chn not in targets:
                continue
            visited.add((nx_, ny_))
            q.append((nx_, ny_, first_dir or d))
    return None


# ------------------------- Compose knowledge query
    ------------------------


def _compose_knowledge_query(
    goal: str,
    entities: List[str],
    inventory: List[str],
    role: Optional[str],
    race: Optional[str],
    alignment: Optional[str],
    topology: Dict[str, List[str]],
    neighbor_sig: str,
    passable_dirs: List[str],
    hazard_dirs: List[str],
    nearest_step: Optional[str] = None,
) -> str:
    topo_sig = (
        f"walls:{,'.join(topology.get('walls_at', []))}; "
        f"up:{,'.join(topology.get('stairs_up_at', []))}; "
        f"down:{,'.join(topology.get('stairs_down_at', []))}; "
        f"boulder:{,'.join(topology.get('boulder_at', []))}; "
        f"fountain:{,'.join(topology.get('fountain_at', []))}"
    )
    rr = ",".join([t for t in [role, race, alignment] if t])
    hazard_summary = "hazards:" + ",".join(sorted(hazard_dirs))
    query = (
        _normalize_space(goal)
        + " | role:"
        + rr
        + " | entities:"
        + ",".join(entities[:12])
        + " | inv:"
        + ",".join(inventory[:12])
        + " | topo:"
        + topo_sig
        + " | neighbors:"
        + neighbor_sig
        + " | passable:"
        + ",".join(passable_dirs)
        + " | "
        + hazard_summary
    )
    if nearest_step:
        query += " | objective_step:" + nearest_step
    return query


def _make_task_signature(
    goal: str,
    actions: List[str],
    entities: List[str],
    role: Optional[str],
```

```
    race: Optional[str],
    alignment: Optional[str],
    topology: Dict[str, Any],
    neighbor_sig: str,
) -> str:
    """
    Deterministic compact signature for the task, based on normalized
         goal, action set,
    canonical entities, role/race/alignment, coarse topology and
        neighbor signature.
    """
    topo_keys = ["walls_at", "doors_at", "stairs_up_at", "
        stairs_down_at", "boulder_at", "fountain_at"]
    topo_norm = {k: sorted(topology.get(k, []) or []) for k in
        topo_keys}
    payload = {
        "g": _normalize_space(goal),
        "a": sorted(actions or []),
        "e": sorted(entities or []),
        "r": role or "",
        "ra": race or "",
        "al": alignment or "",
        "t": topo_norm,
        "n": neighbor_sig or "",
    }
    blob = json.dumps(payload, sort_keys=True, ensure_ascii=False)
    sig = hashlib.sha1(blob.encode("utf-8")).hexdigest()  # stable
        and compact
    return f"minihack:{sig}"


# -------------------------- Memory Layers
    -------------------------------


@dataclass
class TaskSchemaLayer(Sub_memo_layer):
    """
    Parses initial context into a reusable, comparable task schema.
    Stores/retrieves prior schemas with outcome summaries. Distills
        similar cases.
    """
    layer_intro: str = "Task schema library: normalized task
        descriptors with goals, action sets, topology, and outcomes."
    database: Optional[Any] = field(default=None)

    def __post_init__(self):
        if self.database is None:
            self.database = Chroma(embedding_function=Embedding())

    async def _build_schema(self, init_dict: Dict[str, Any]) -> Dict[
        str, Any]:
        goal = init_dict.get("goal", "")
        long_term = init_dict.get("long_term_context", "")
        short_term = init_dict.get("short_term_context", "")
        action_dict = init_dict.get("action_dict", {}) or {}

        map_w, map_h = _extract_map_area(long_term)
        pos = _extract_position(long_term)
        stats_line = _extract_stats_line(long_term)
        role_info = _parse_role_race_alignment(long_term)
        entities = _canonicalize_entities(long_term)
        inv_items = _simplify_inventory_items(short_term)
        actions = _extract_action_set(action_dict)
        topology = _extract_local_topology(long_term)
```

```
        # Parse ascii map for neighbors and passability
        parsed_map = _parse_ascii_map(long_term)
        neighbors = parsed_map.get("neighbors", {})
        passable_dirs = [d for d, ch in neighbors.items() if
            _is_passable_char(ch)]
        hazard_dirs = _neighbors_hazard_dirs(neighbors)
        neighbor_sig = _neighbors_signature(neighbors)

        # Heuristic: nearby objectives within small radius
        targets = set()
        gl = goal.lower()
        if "stairs" in gl:
            targets.update({"<", ">"})
        if "boulder" in gl or "sokoban" in gl or "boxoban" in gl:
            targets.update({"{"})
        nstep = None
        if parsed_map.get("grid") and parsed_map.get("hero_xy"):
            nstep = _bfs_first_step_to_targets(
                parsed_map["grid"], parsed_map["width"], parsed_map["
                    height"], parsed_map["hero_xy"], targets, radius
                    =5
            )

        # Override textual topology with map evidence for adjacency-
            critical signals
        if neighbors:
            nearby_up = sorted([d for d, ch in neighbors.items() if
                ch == "<"])
            nearby_down = sorted([d for d, ch in neighbors.items() if
                 ch == ">"])
            if nearby_up:
                topology["stairs_up_at"] = nearby_up
            if nearby_down:
                topology["stairs_down_at"] = nearby_down
            # Fountains near
            fdirs = sorted([d for d, ch in neighbors.items() if ch ==
                 "{"])
            if fdirs:
                topology["fountain_at"] = fdirs

        # Augment canonical entities from neighbor evidence
        if neighbors:
            if any(ch == "<" for ch in neighbors.values()) and "
                stairs_up" not in entities:
                entities.append("stairs_up")
            if any(ch == ">" for ch in neighbors.values()) and "
                stairs_down" not in entities:
                entities.append("stairs_down")
            if any(ch == "{" for ch in neighbors.values()) and "
                fountain" not in entities:
                entities.append("fountain")
        entities = sorted(set(entities))

        signature = _make_task_signature(
            goal, actions, entities, role_info["role"], role_info["
                race"], role_info["alignment"], topology,
                neighbor_sig
        )
        knowledge_query = _compose_knowledge_query(
            goal,
            entities,
            inv_items,
            role_info["role"],
            role_info["race"],
```

```
            role_info["alignment"],
            topology,
            neighbor_sig,
            passable_dirs,
            hazard_dirs,
            nearest_step=nstep,
        )

        schema = {
            "task_id": signature,
            "goal": _normalize_space(goal),
            "actions": actions,
            "initial_entities": entities,
            "inventory_items": inv_items,
            "map_size": {"width": map_w, "height": map_h},
            "initial_position": {"x": pos[0], "y": pos[1]} if pos
                else None,
            "initial_stats_line": stats_line,
            "role": role_info["role"],
            "race": role_info["race"],
            "alignment": role_info["alignment"],
            "local_topology": topology,
            "map_neighbors": neighbors,
            "passable_dirs": passable_dirs,
            "hazard_dirs": hazard_dirs,
            "nearest_objective_step": nstep,
            "knowledge_query": knowledge_query,
        }
        return schema

    async def _distill_case_lessons(
        self, cases_texts: List[Tuple[str, Dict[str, Any]]], schema:
            Dict[str, Any]
    ) -> List[Dict[str, Any]]:
        if not cases_texts:
            return [
                {
                    "lesson": "Prefer passable safe tiles; avoid
                        oscillation; move toward visible objectives
                        when safe.",
                }
            ]
        distilled: List[Dict[str, Any]] = []
        lesson_schema = {"lesson": {"type": "string", "description":
            "One or two concise actionable lines"}}
        agent = _get_default_agent(
            "Distill a short, actionable lesson from a prior episode
                snippet. Focus on transferable insight for grid
                roguelikes.",
            lesson_schema,
        )
        for text, md in cases_texts[:2]:
            try:
                resp = await agent.ask(
                    f"Current goal: {schema.get('goal')}\nEntities:
                        {', '.join(schema.get('initial_entities', [])
                        )}\nTopology: {schema.get('local_topology')}\
                        nSnippet:\n{text}\nProduce one concise lesson
                         (<=180 chars).",
                    with_history=False,
                )
                lesson = _truncate(resp.get("lesson", ""), 200)
            except Exception:
                lesson = _truncate(
```

```
                    "Explore open tiles first; avoid pushing objects
                        into corners; align with objectives.",
                    200,
                )
            distilled.append(
                _clean_list_of_dicts(
                    [
                        {
                            "case_id": md.get("case_id"),
                            "reward": md.get("reward"),
                            "outcome": md.get("outcome"),
                            "lesson": lesson,
                        }
                    ],
                    drop_none=True,
                )[0]
            )
        return distilled

    async def retrieve(self, **kwargs) -> Dict[str, Any]:
        init_dict: Dict[str, Any] = kwargs.get("init", {})
        if not isinstance(init_dict, dict):
            raise ValueError("TaskSchemaLayer.retrieve requires init
                dict")
        schema = await self._build_schema(init_dict)
        docs = self.database.similarity_search(schema["
            knowledge_query"], k=12)
        ranked: List[Tuple[str, Dict[str, Any]]] = []
        seen_ids = set()
        for d in docs:
            md = d.metadata or {}
            # hygiene: drop empty/generic stubs
            es = str(md.get("episode_summary", "") or "").strip()
            if es and len(es) < 30:
                continue
            case_id = md.get("case_id")
            if case_id and case_id in seen_ids:
                continue
            if case_id:
                seen_ids.add(case_id)
            reward = md.get("reward")
            try:
                reward_f = float(reward) if reward is not None else
                    0.0
            except Exception:
                reward_f = 0.0
            ranked.append(
                (
                    d.page_content,
                    {
                        "case_id": md.get("case_id"),
                        "task_id": md.get("task_id"),
                        "reward": reward_f,
                        "outcome": md.get("outcome"),
                    },
                )
            )
        ranked = [r for r in ranked if r[1].get("reward", 0.0) > 0.0]
             or ranked
        ranked.sort(key=lambda x: (x[1].get("reward", 0.0)), reverse=
            True)
        # Keep top-2 to avoid noise
        ranked = ranked[:2]
        distilled = await self._distill_case_lessons(ranked, schema)
        return {"task_schema": schema, "similar_cases": distilled}
```

```
    async def update(self, **kwargs) -> None:
        init_dict: Dict[str, Any] = kwargs.get("init", {})
        reward: float = kwargs.get("reward", 0.0)
        outcome: str = kwargs.get("outcome", "")
        episode_summary: str = kwargs.get("episode_summary", "")
        if not isinstance(init_dict, dict):
            return
        schema = await self._build_schema(init_dict)
        case_id = str(uuid.uuid4())
        meta = {
            "case_id": case_id,
            "task_id": schema["task_id"],
            "reward": float(reward),
            "outcome": outcome,
            "episode_summary": _truncate(episode_summary, 1000),
        }
        text = (
            f"Goal: {schema['goal']}\n"
            f"Role/Race/Align: {schema.get('role')}/{schema.get('race
                ')}/{schema.get('alignment')}\n"
            f"Actions: {', '.join(schema['actions'])}\n"
            f"Entities: {', '.join(schema['initial_entities'])}\n"
            f"Inventory: {', '.join(schema['inventory_items'])}\n"
            f"Topo: {schema['local_topology']}\n"
            f"Neighbors: {_neighbors_signature(schema.get('
                map_neighbors', {}))}\n"
            f"Passable: {', '.join(schema.get('passable_dirs', []))}\
                n"
            f"Outcome: {outcome}, Reward: {reward}\n"
            f"Summary: {episode_summary}\n"
        )
        self.database.add_texts(texts=[text], metadatas=[meta], ids=[
            case_id])


@dataclass
class StrategyLibraryLayer(Sub_memo_layer):
    """
    Stores distilled, transferable strategies and pitfalls.
    Retrieves high-signal guidance for the current task schema and
        compresses to bullets.
    """
    layer_intro: str = "Strategy library: distilled plan outlines,
        heuristics, and pitfalls aggregated across tasks."
    database: Optional[Any] = field(default=None)

    def __post_init__(self):
        if self.database is None:
            self.database = Chroma(embedding_function=Embedding())

    async def _compress_suggestions(self, schema: Dict[str, Any],
        suggestions: List[str]) -> List[str]:
        if not suggestions:
            suggestions = [
                "Survey adjacent passable tiles and favor new
                    exploration paths.",
                "Avoid oscillating moves; if blocked, pivot or search
                    .",
                "Align early moves toward visible objectives when
                    safe.",
            ]
        out_schema = {
            "bullets": {
                "type": "array",
```

```
                "items": {"type": "string"},
                "description": "3-6 concise, task-conditioned tips.",
            }
        }
        agent = _get_default_agent(
            "Condense raw strategy hints into 3-6 concise, actionable
                 bullets for grid roguelike tasks. Max total 500
                chars.",
            out_schema,
        )
        try:
            joined = "\n".join(suggestions)
            resp = await agent.ask(
                f"Goal: {schema.get('goal')}\nEntities: {', '.join(
                    schema.get('initial_entities', []))}\nInventory:
                    {', '.join(schema.get('inventory_items', []))}\
                    nTopology: {schema.get('local_topology')}\
                    nNeighbors: {_neighbors_signature(schema.get('
                    map_neighbors', {}))}\nRaw hints:\n{joined}\
                    nCondense to bullets.",
                with_history=False,
            )
            bullets = resp.get("bullets", [])[:6]
            bullets = [_truncate(b, 160) for b in bullets]
            if not bullets:
                bullets = [
                    "Explore passable directions; avoid repeated wall
                         bumps.",
                    "Keep escape routes; do not corner pushable
                        objects.",
                    "Approach visible objectives only when safe.",
                ]
            return bullets
        except Exception:
            return [
                "Survey open directions; avoid bumping into walls
                    repeatedly.",
                "Prefer reversible steps; do not corner pushable
                    objects.",
                "Re-evaluate plan after each interaction or new
                    discovery.",
            ]

    async def retrieve(self, **kwargs) -> Dict[str, Any]:
        schema: Dict[str, Any] = kwargs.get("schema", {})
        if not schema:
            return {"strategies": [], "plan_outline": {}, "
                bulleted_tips": []}
        query = schema.get("knowledge_query", "") or schema.get("goal
            ", "")
        docs = self.database.similarity_search(query, k=10)
        raw_texts = []
        filtered = []
        for d in docs:
            md = d.metadata or {}
            q = md.get("quality")
            try:
                qf = float(q) if q is not None else 0.0
            except Exception:
                qf = 0.0
            if qf < 0.05:
                continue
            raw_texts.append(d.page_content)
            filtered.append(
                {
```

```
                "strategy_id": md.get("strategy_id"),
                "topics": md.get("topics"),
                "suggestion": _truncate(d.page_content, 300),
            }
        )
    bullets = await self._compress_suggestions(schema, raw_texts)

    plan_schema = {
        "plan_title": {"type": "string", "description": "Short
            title for the plan"},
        "ordered_steps": {
            "type": "array",
            "items": {"type": "string"},
            "description": "Ordered, high-level steps.",
        },
        "pitfalls_to_avoid": {
            "type": "array",
            "items": {"type": "string"},
            "description": "Common mistakes and how to avoid them
                .",
        },
        "key_checks": {
            "type": "array",
            "items": {"type": "string"},
            "description": "State checks to perform during
                execution.",
        },
    }
    agent = _get_default_agent(
        "Generate concise, transferable plans for grid-based
            roguelike tasks. Tailor to goal, entities, inventory,
             and local topology.",
        plan_schema,
    )
    hint_text = "\n".join([s["suggestion"] for s in filtered])
    user_prompt = (
        f"Goal: {schema.get('goal')}\n"
        f"Entities: {', '.join(schema.get('initial_entities', [])
            )}\n"
        f"Actions: {', '.join(schema.get('actions', []))}\n"
        f"Inventory: {', '.join(schema.get('inventory_items', [])
            )}\n"
        f"Topology: {schema.get('local_topology')}\n"
        f"Neighbors: {_neighbors_signature(schema.get('
            map_neighbors', {}))}\n"
        f"Prior strategy hints:\n{hint_text}\n"
        f"Produce an outline."
    )
    try:
        plan_outline = await agent.ask(user_prompt, with_history=
            False)
    except Exception:
        plan_outline = {
            "plan_title": "Generic Plan",
            "ordered_steps": [
                "Survey surroundings and identify objectives and
                    blockers.",
                "Map reachable safe tiles and plan minimal-risk
                    path.",
                "Interact with key objects while maintaining
                    retreat paths.",
            ],
            "pitfalls_to_avoid": [
                "Avoid pushing objects into corners unless goal-
                    aligned.",
```

```
                    "Do not enter dead-ends without escape.",
                ],
                "key_checks": [
                    "Re-evaluate objective proximity",
                    "Monitor HP and threats before risky actions",
                ],
            }
        filtered = _clean_list_of_dicts(filtered, drop_none=True,
            truncate_keys=["suggestion"])
        return {"strategies": filtered[:4], "plan_outline":
            plan_outline, "bulleted_tips": bullets}

    async def update(self, **kwargs) -> None:
        schema: Dict[str, Any] = kwargs.get("schema", {})
        summary: Dict[str, Any] = kwargs.get("summary", {})
        if not schema or not summary:
            return
        strategies: List[str] = summary.get("transferable_strategies"
            , [])
        pitfalls: List[str] = summary.get("pitfalls", [])
        topics: List[str] = summary.get("topics", [])
        score: float = float(summary.get("signals", {}).get("score",
            0.0))
        success = bool(summary.get("signals", {}).get("success",
            False))

        texts: List[str] = []
        metas: List[Dict[str, Any]] = []
        ids: List[str] = []

        for s in strategies:
            sid = str(uuid.uuid4())
            texts.append(s)
            metas.append(
                {
                    "strategy_id": sid,
                    "source_task_id": schema.get("task_id"),
                    "quality": score if score is not None else 0.0,
                    "success_bias": success,
                    "topics": ",".join(topics[:10]),
                }
            )
            ids.append(sid)

        for p in pitfalls:
            sid = str(uuid.uuid4())
            texts.append("Pitfall: " + p)
            metas.append(
                {
                    "strategy_id": sid,
                    "source_task_id": schema.get("task_id"),
                    "quality": (score or 0.0) * 0.9,
                    "success_bias": success,
                    "topics": ",".join(topics[:10]),
                }
            )
            ids.append(sid)

        if texts:
            self.database.add_texts(texts=texts, metadatas=metas, ids
                =ids)


@dataclass
class SpatialPriorLayer(Sub_memo_layer):
```

```
    """
    A knowledge graph of entity/action relations and spatial
        affordances.
    Nodes: entity types or action tokens. Edges: relation with
        weights.
    """
    layer_intro: str = "Spatial knowledge graph: entity/action
        affordances and relations with support counts."
    database: Optional[Any] = field(default=None)

    def __post_init__(self):
        if self.database is None:
            self.database = nx.Graph()
        for node in ["blocks_movement", "pushable", "goal_target", "
            hazard", "slows", "unknown"]:
            if node not in self.database:
                self.database.add_node(node, kind="relation")
        base_pairs = [
            ("wall", "blocks_movement", 3.0),
            ("door", "blocks_movement", 1.5),
            ("boulder", "pushable", 2.5),
            ("stairs_up", "goal_target", 1.0),
            ("stairs_down", "goal_target", 1.0),
            ("trap", "hazard", 2.0),
            ("lava", "hazard", 2.5),
            ("monster", "hazard", 1.5),
            ("fountain", "goal_target", 1.0),
            ("water", "hazard", 1.0),
        ]
        for e, r, w in base_pairs:
            if e not in self.database:
                self.database.add_node(e, kind="entity")
            if not self.database.has_edge(e, r):
                self.database.add_edge(e, r, weight=w)

    def _increment_relation(self, entity: str, relation: str, weight:
         float = 1.0):
        g: nx.Graph = self.database
        entity = entity.lower()
        if entity not in g:
            g.add_node(entity, kind="entity")
        if relation not in g:
            g.add_node(relation, kind="relation")
        if g.has_edge(entity, relation):
            g.edges[(entity, relation)]["weight"] += weight
        else:
            g.add_edge(entity, relation, weight=weight)

    def _parse_relations_from_text(self, text: str):
        low = (text or "").lower()
        if "cannot pass through" in low or "cannot move through" in
            low:
            m = re.search(r"cannot (?:pass|move) through (?:the )?([a
                -z\- ]+)", low)
            if m:
                mention = m.group(1).strip()
                if "bar" in mention:
                    self._increment_relation("wall", "blocks_movement
                        ", 2.0)
                else:
                    self._increment_relation(mention.split()[0], "
                        blocks_movement", 2.0)
        if "it's solid stone" in low or "there is a wall in the way"
            in low or "you hit the wall" in low:
            self._increment_relation("wall", "blocks_movement", 1.5)
```

```
        if "push" in low and "boulder" in low:
            self._increment_relation("boulder", "pushable", 1.2)
        if re.search(r"stairs?\s+up", low):
            self._increment_relation("stairs_up", "goal_target", 0.8)
        if re.search(r"stairs?\s+down", low):
            self._increment_relation("stairs_down", "goal_target",
                0.8)
        for obj in ["boulder", "fountain", "altar", "sink"]:
            if re.search(rf"you see here (?:an?|the)?\s*{obj}", low):
                rel = "goal_target" if obj in ["fountain"] else "
                    unknown"
                self._increment_relation(obj, rel, 0.6)
        for hazard in ["trap", "lava", "acid", "poison", "spike", "
            monster"]:
            if hazard in low:
                self._increment_relation(hazard, "hazard", 1.0)
        if "water" in low or "water floor" in low:
            self._increment_relation("water", "hazard", 1.0)

    async def retrieve(self, **kwargs) -> Dict[str, Any]:
        entities: List[str] = kwargs.get("entities", []) or []
        topology: Dict[str, Any] = kwargs.get("topology", {}) or {}
        neighbors: Dict[str, str] = kwargs.get("map_neighbors", {})
            or {}
        inferred: List[str] = list(entities)

        if not inferred and topology:
            if topology.get("stairs_up_at"):
                inferred.append("stairs_up")
            if topology.get("stairs_down_at"):
                inferred.append("stairs_down")
            if topology.get("fountain_at"):
                inferred.append("fountain")
            if topology.get("walls_at"):
                inferred.append("wall")
        if neighbors:
            if any(ch == "<" for ch in neighbors.values()):
                inferred.append("stairs_up")
            if any(ch == ">" for ch in neighbors.values()):
                inferred.append("stairs_down")
            if any(ch == "{" for ch in neighbors.values()):
                inferred.append("fountain")
            if any(_is_hazard_char(ch) for ch in neighbors.values()):
                inferred.append("water")

        g: nx.Graph = self.database
        priors: List[str] = []
        for e in set(inferred):
            if e in g:
                neighbors_rel = []
                for nbr in g.neighbors(e):
                    w = g.edges[(e, nbr)].get("weight", 1.0)
                    if g.nodes[nbr].get("kind") == "relation":
                        neighbors_rel.append((nbr, w))
                neighbors_rel.sort(key=lambda x: -x[1])
                for rel, w in neighbors_rel[:5]:
                    priors.append(f"{e} -> {rel} (support={round(w,
                        2)})")

        if not priors:
            base = [("wall", "blocks_movement"), ("boulder", "
                pushable"), ("stairs_down", "goal_target")]
            for e, r in base:
                if g.has_edge(e, r):
```

```
                    priors.append(f"{e} -> {r} (support={round(g.
                        edges[(e, r)].get('weight', 1.0), 2)})")
        return {"spatial_priors": priors}

    async def update(self, **kwargs) -> None:
        init_long = (kwargs.get("init") or {}).get("long_term_context
            ", "")
        steps: List[Dict[str, Any]] = kwargs.get("steps", []) or []
        goal_text: str = (kwargs.get("init") or {}).get("goal", "")
        if init_long:
            self._parse_relations_from_text(init_long)
        if goal_text:
            self._parse_relations_from_text("goal: " + goal_text)
        for st in steps:
            ltc = st.get("long_term_context", "")
            if ltc:
                self._parse_relations_from_text(ltc)


@dataclass
class RiskAndInteractionLayer(Sub_memo_layer):
    """
    Stores risk management heuristics and interaction patterns with
        inventory/terrain/monsters.
    Compresses retrieved tips into concise bullets.
    """
    layer_intro: str = "Risk and interaction library: safety
        heuristics and inventory usage rules."
    database: Optional[Any] = field(default=None)

    def __post_init__(self):
        if self.database is None:
            self.database = Chroma(embedding_function=Embedding())

    async def _compress_tips(self, schema: Dict[str, Any], tips_texts
        : List[str]) -> List[str]:
        if not tips_texts:
            tips_texts = [
                "Avoid repeating blocked moves; pivot to passable
                    directions or search.",
                "Maintain escape routes; do not corner yourself while
                     exploring.",
                "Use healing items proactively when HP is low before
                    risky moves.",
            ]
        out_schema = {"tips": {"type": "array", "items": {"type": "
            string"}}}
        agent = _get_default_agent(
            "Condense risk and interaction notes into 3-6 concise
                bullets tailored to goal, topology, and inventory.
                Max total 450 chars.",
            out_schema,
        )
        try:
            joined = "\n".join(tips_texts)
            resp = await agent.ask(
                f"Goal: {schema.get('goal')}\nInventory: {', '.join(
                    schema.get('inventory_items', []))}\nEntities:
                    {', '.join(schema.get('initial_entities', []))}\
                    nTopology: {schema.get('local_topology')}\
                    nNeighbors: {_neighbors_signature(schema.get('
                    map_neighbors', {}))}\nNotes:\n{joined}\nCondense
                    .",
                with_history=False,
            )
```

```
            bullets = resp.get("tips", [])[:6]
            bullets = [_truncate(b, 150) for b in bullets]
            if not bullets:
                bullets = [
                    "Avoid repeating blocked moves; pivot or search."
                        ,
                    "Keep escape routes; avoid tight corners near
                        walls.",
                ]
            return bullets
        except Exception:
            return [
                "Avoid repeated moves into blocked tiles; pivot and
                    explore open directions.",
                "Preserve escape routes when engaging hazards.",
                "Use healing consumables preemptively at low HP.",
            ]

    async def retrieve(self, **kwargs) -> Dict[str, Any]:
        schema: Dict[str, Any] = kwargs.get("schema", {})
        if not schema:
            return {"risk_tips": []}
        inv = schema.get("inventory_items", [])
        ents = schema.get("initial_entities", [])
        query = (
            "risk safety interactions | items: "
            + ", ".join(inv[:10])
            + " | ents: "
            + ", ".join(ents[:10])
            + " | topo:"
            + json.dumps(schema.get("local_topology", {}))
        )
        docs = self.database.similarity_search(query, k=10)
        raw_texts = []
        for d in docs:
            md = d.metadata or {}
            conf = md.get("confidence")
            try:
                cf = float(conf) if conf is not None else 0.0
            except Exception:
                cf = 0.0
            if cf < 0.05:
                continue
            raw_texts.append(d.page_content)
        bullets = await self._compress_tips(schema, raw_texts)
        return {"risk_tips": bullets}

    async def update(self, **kwargs) -> None:
        schema: Dict[str, Any] = kwargs.get("schema", {})
        summary: Dict[str, Any] = kwargs.get("summary", {})
        if not schema or not summary:
            return
        risk_notes: List[str] = summary.get("risk_notes", [])
        score: float = float(summary.get("signals", {}).get("score",
            0.0))
        success = bool(summary.get("signals", {}).get("success",
            False))
        texts, metas, ids = [], [], []
        for r in risk_notes:
            rid = str(uuid.uuid4())
            texts.append(r)
            metas.append(
                {
                    "tip_id": rid,
                    "source_task_id": schema.get("task_id"),
```

```
                    "confidence": score * (1.2 if success else 0.8),
                }
            )
            ids.append(rid)
        if texts:
            self.database.add_texts(texts=texts, metadatas=metas, ids
                =ids)


@dataclass
class ReflexRulesLayer(Sub_memo_layer):
    """
    Reflexive, topology-conditioned micro-advice for immediate next
        moves.
    Stateless generation based on current schema and spatial priors.
    """
    layer_intro: str = "Reflex rules: pattern-to-action guidance tied
         to local topology and common affordances."
    database: Optional[Any] = field(default=None)

    async def retrieve(self, **kwargs) -> Dict[str, Any]:
        schema: Dict[str, Any] = kwargs.get("schema", {})
        spatial_priors: List[str] = kwargs.get("spatial_priors", [])
        hazard_dirs: List[str] = kwargs.get("hazard_dirs", []) or []
        goal = (schema.get("goal", "") or "")
        topo = schema.get("local_topology", {}) or {}
        neighbors: Dict[str, str] = schema.get("map_neighbors", {})
            or {}
        actions: List[str] = schema.get("actions", []) or []
        tips: List[str] = []

        blocked_dirs = topo.get("walls_at", []) or []
        if blocked_dirs:
            tips.append(
                f"Avoid bumping into blocked directions: {', '.join(
                    sorted(blocked_dirs))}. Pivot to open tiles."
            )
        if neighbors:
            blocked_map = [d for d, ch in neighbors.items() if not
                _is_passable_char(ch)]
            if blocked_map:
                tips.append(f"Adjacent blocked tiles: {', '.join(
                    sorted(blocked_map))}. Favor passable neighbors."
                    )
        if hazard_dirs:
            tips.append(
                "Avoid stepping onto hazardous tiles (water/lava/
                    traps) unless necessary; prefer safe passable
                    neighbors."
            )
        sokoban_like = "boulder" in goal.lower() or "sokoban" in goal
            .lower() or "boxoban" in goal.lower()
        if sokoban_like:
            tips.append("Leave space behind boulders before pushing;
                avoid pushing into corners or against walls.")
            if topo.get("boulder_at") and topo.get("fountain_at"):
                tips.append("Align boulder-fountain along open lanes;
                     scout path before any push.")
        for p in spatial_priors:
            if "boulder -> pushable" in p and not any("Boulders are
                pushable" in t for t in tips):
                tips.append("Boulders are pushable; ensure the tile
                    behind is free before pushing.")
            if "wall -> blocks_movement" in p and not any("Walls/bars
                 block" in t for t in tips):
```

```
                tips.append("Walls/bars block movement; navigate
                    around instead of repeating bumps.")
        stairs_adjacent = any(ch in ("<", ">") for ch in neighbors.
            values())
        progress_goal = any(k in goal.lower() for k in ["as far as
            possible", "progress", "deeper", "descend", "ascend"])
        if stairs_adjacent:
            if progress_goal:
                tips.append("Objective is progress-descend/ascend now
                     if safe (HP ok, no threats adjacent).")
            else:
                tips.append("Only approach stairs after scanning the
                    room; avoid premature level changes.")
        if "search" in actions:
            near_walls = [d for d, ch in neighbors.items() if ch == "
                #"]
            if near_walls and not topo.get("doors_at"):
                tips.append("If adjacent to walls/bars with no door
                    visible, try searching along the wall before
                    advancing.")
        tips = [_truncate(t, 180) for t in tips][:6]
        return {"reflex_tips": tips}

    async def update(self, **kwargs) -> None:
        return


# -------------------------- Orchestrator
    ----------------------------------


def _normalize_action_token(action: Optional[str]) -> Optional[str]:
    if not action:
        return None
    a = _normalize_space(action).lower()
    # remove 'move ' prefix
    a = re.sub(r"^move\s+", "", a)
    # reduce 'far ' prefix
    a = re.sub(r"^far\s+", "", a)
    mapping = {
        "n": "north",
        "e": "east",
        "s": "south",
        "w": "west",
        "ne": "northeast",
        "se": "southeast",
        "sw": "southwest",
        "nw": "northwest",
        "north": "north",
        "east": "east",
        "south": "south",
        "west": "west",
        "northeast": "northeast",
        "southeast": "southeast",
        "southwest": "southwest",
        "northwest": "northwest",
    }
    if a in mapping:
        return mapping[a]
    # try to map tokens like "northwest " with extra text
    for k, v in mapping.items():
        if a.startswith(k):
            return v
    return a if a in mapping.values() else None
```

```
class MiniHackMemory(MemoStructure):
    """
    Multi-layer memory: Task schema -> Spatial priors -> Strategy
        retrieval -> Risk tips -> Reflex tips
    Update: Summarize episode (+action failure counts, oscillation)
        -> Update Spatial KG -> Update Strategy & Risk -> Index
        Schema
    """

    def __init__(self):
        super().__init__()
        self.schema_layer = TaskSchemaLayer()
        self.strategy_layer = StrategyLibraryLayer()
        self.spatial_layer = SpatialPriorLayer()
        self.risk_layer = RiskAndInteractionLayer()
        self.reflex_layer = ReflexRulesLayer()

    async def _summarize_episode(
        self, init_dict: Dict[str, Any], steps: List[Dict[str, Any]],
             reward: float
    ) -> Dict[str, Any]:
        schema = {
            "episode_summary": {"type": "string", "description": "
                Concise summary of what happened."},
            "transferable_strategies": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Generic strategies that can transfer
                    to similar tasks.",
            },
            "pitfalls": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Common mistakes made or observed.",
            },
            "risk_notes": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Safety-related notes and interactions
                    .",
            },
            "signals": {
                "type": "object",
                "description": "Outcome signals.",
                "properties": {
                    "success": {"type": "boolean"},
                    "score": {"type": "number"},
                },
                "required": ["success", "score"],
            },
            "topics": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Short topic tags like 'pushing', '
                    navigation', 'hazards'.",
            },
        }
        agent = _get_default_agent(
            "Summarize an episode from MiniHack. Extract transferable
                 strategies, pitfalls, and risk notes. "
            "Use repeated failed actions and oscillation info to
                craft actionable pitfalls. Keep outputs concise and
                generic.",
            schema,
```

```
        )
        init_text = (
            f"Goal: {init_dict.get('goal','')}\n"
            f"Init entities: {', '.join(_canonicalize_entities(
                init_dict.get('long_term_context','')))}\n"
            f"Init inventory: {', '.join(_simplify_inventory_items(
                init_dict.get('short_term_context','')))}\n"
            f"Actions: {', '.join(_extract_action_set(init_dict.get('
                action_dict',{})))}\n"
        )
        sample_steps = steps[-12:] if len(steps) > 12 else steps
        step_summaries = []
        blocked_counts: Dict[str, int] = {}
        action_seq: List[str] = []
        for i, st in enumerate(sample_steps):
            msg = st.get("long_term_context", "")
            info = _extract_stats_line(msg)
            step_summaries.append(f"t{i}: {info} | msg_head: {msg.
                splitlines()[0] if msg else ''}")
        for st in steps:
            acts = st.get("action_took")
            if isinstance(acts, str):
                direction = _normalize_action_token(acts)
            elif isinstance(acts, list) and acts:
                direction = _normalize_action_token(acts[0])
            else:
                direction = None
            if direction:
                action_seq.append(direction)
            if _detect_blocked_move_message(st.get("long_term_context
                ", "")):
                if direction:
                    blocked_counts[direction] = blocked_counts.get(
                        direction, 0) + 1
        blocked_summary = ", ".join([f"{d}:{c}" for d, c in sorted(
            blocked_counts.items(), key=lambda x: -x[1])])
        oscill = _compute_oscillation(action_seq)

        user_prompt = (
            f"Episode overview:\n{init_text}\n"
            f"Reward: {reward}\n"
            f"Repeated blocked moves (direction:count): {
                blocked_summary if blocked_summary else 'none'}\n"
            f"Oscillation score: {oscill['score']}, pairs: {oscill['
                pairs']}\n"
            f"Recent steps (compact):\n" + "\n".join(step_summaries)
        )
        try:
            result = await agent.ask(user_prompt, with_history=False)
        except Exception:
            result = {
                "episode_summary": "Summary unavailable.",
                "transferable_strategies": [
                    "Reassess pathing when encountering blocked tiles
                        ; shift to open directions.",
                    "Plan pushes to keep escape routes and avoid
                        corners.",
                ],
                "pitfalls": [
                    "Repeating moves into known walls; pivot and
                        explore new paths.",
                ],
                "risk_notes": ["Monitor HP; avoid entering unknown
                    areas when low."],
```

```
                "signals": {"success": reward > 0, "score": float(
                    reward)},
                "topics": ["navigation", "planning"],
            }

        if oscill["score"] > 1:
            pair_txt = ", ".join([f"{k}:{v}" for k, v in sorted(
                oscill["pairs"].items(), key=lambda x: -x[1])])
            osc_pitfall = f"Avoid back-and-forth oscillation ({
                pair_txt}); commit to a new passable direction or
                search."
            result.setdefault("pitfalls", [])
            if osc_pitfall not in result["pitfalls"]:
                result["pitfalls"].append(osc_pitfall)
            result.setdefault("topics", [])
            if "oscillation" not in result["topics"]:
                result["topics"].append("oscillation")

        return result

    async def general_retrieve(self, recorder: Basic_Recorder) ->
        Dict:
        init_dict = getattr(recorder, "init", {}) or {}

        # Step 1: Task schema and filtered, distilled similar cases
        schema_ret = await self.schema_layer.retrieve(init=init_dict)
        schema = schema_ret.get("task_schema", {})

        # Step 2: Spatial priors for canonical entities (with
            topology and map neighbors)
        spatial_ret = await self.spatial_layer.retrieve(
            entities=schema.get("initial_entities", []),
            topology=schema.get("local_topology", {}),
            map_neighbors=schema.get("map_neighbors", {}),
        )

        # Step 3: Strategies and condensed bullets tailored to schema
        strat_ret = await self.strategy_layer.retrieve(schema=schema)

        # Step 4: Risk and interaction tips condensed
        risk_ret = await self.risk_layer.retrieve(schema=schema)

        # Step 5: Reflex micro-advice (hazard-aware)
        reflex_ret = await self.reflex_layer.retrieve(
            schema=schema,
            spatial_priors=spatial_ret.get("spatial_priors", []),
            hazard_dirs=schema.get("hazard_dirs", []),
        )

        # Step 6: Initial movement suggestions based on local map
            neighbors and topology with hazard-awareness
        move_suggestions = _suggest_first_moves(
            schema.get("actions", []),
            schema.get("local_topology", {}) or {},
            map_neighbors=schema.get("map_neighbors", {}),
            passable_dirs=schema.get("passable_dirs", []),
            goal_text=schema.get("goal", "") or "",
            hazard_dirs=schema.get("hazard_dirs", []),
            nearest_objective_step=schema.get("nearest_objective_step
                "),
        )

        # Action mask for downstream planners
        allowed_safe = [d for d in schema.get("passable_dirs", []) if
             d not in set(move_suggestions.get("hazard_dirs", []))]
```

```
        action_mask = {
            "allowed": sorted(allowed_safe),
            "blocked": move_suggestions.get("blocked_dirs", []),
            "hazard": move_suggestions.get("hazard_dirs", []),
            "fallback": move_suggestions.get("fallback_actions", []),
        }

        # Build clean memory package
        task_overview = {
            "goal": schema.get("goal"),
            "role": schema.get("role"),
            "race": schema.get("race"),
            "alignment": schema.get("alignment"),
            "actions": schema.get("actions"),
            "map_size": schema.get("map_size"),
            "initial_position": schema.get("initial_position"),
            "initial_entities": schema.get("initial_entities"),
            "inventory_items": schema.get("inventory_items"),
            "local_topology": schema.get("local_topology"),
            "map_neighbors": schema.get("map_neighbors"),
            "passable_dirs": schema.get("passable_dirs"),
            "hazard_dirs": schema.get("hazard_dirs"),
            "nearest_objective_step": schema.get("
                nearest_objective_step"),
            "initial_movement_suggestions": move_suggestions,
            "action_mask": action_mask,
            "similar_cases": schema_ret.get("similar_cases", []),
        }
        task_overview = {k: v for k, v in task_overview.items() if v
            not in [None, {}, [], ""]}

        memory_package = {
            "task_overview": task_overview,
            "suggested_strategies": {
                "plan_outline": strat_ret.get("plan_outline"),
                "bulleted_tips": strat_ret.get("bulleted_tips", []),
            },
            "spatial_priors": spatial_ret.get("spatial_priors", []),
            "risk_and_interactions": risk_ret.get("risk_tips", []),
            "reflex_tips": reflex_ret.get("reflex_tips", []),
            "retrieval_metadata": {
                "task_id": schema.get("task_id"),
                "knowledge_query": schema.get("knowledge_query"),
            },
        }
        if isinstance(memory_package.get("task_overview", {}).get("
            similar_cases"), list):
            memory_package["task_overview"]["similar_cases"] =
                _clean_list_of_dicts(
                memory_package["task_overview"]["similar_cases"],
                    drop_none=True, truncate_keys=["lesson"]
            )
        memory_package["retrieval_metadata"] = {k: v for k, v in
            memory_package["retrieval_metadata"].items() if v}
        return memory_package

    async def general_update(self, recorder: Basic_Recorder) -> None:
        init_dict = getattr(recorder, "init", {}) or {}
        steps: List[Dict[str, Any]] = getattr(recorder, "steps", [])
            or []
        reward: float = float(getattr(recorder, "reward", 0.0) or
            0.0)

        # Build schema once for downstream updates
        schema_ret = await self.schema_layer.retrieve(init=init_dict)
```

```
        schema = schema_ret.get("task_schema", {}) or {}

        # Summarize the episode with repeated failure counts and
            oscillation
        summary = await self._summarize_episode(init_dict, steps,
            reward)

        # Update spatial knowledge first
        await self.spatial_layer.update(init=init_dict, steps=steps)

        # Update strategy and risk libraries with distilled info
        await self.strategy_layer.update(schema=schema, summary=
            summary)
        await self.risk_layer.update(schema=schema, summary=summary)

        # Finally, index the task schema with outcome and full
            summary
        outcome = "success" if summary.get("signals", {}).get("
            success", False) else "failure_or_partial"
        await self.schema_layer.update(
            init=init_dict,
            reward=reward,
            outcome=outcome,
            episode_summary=summary.get("episode_summary", ""),
        )
```

Listing 3: Example of the retrieved knowledge context during the Deployment Phase in MiniHack.

```
memo_retrieved = {
"task_overview": {
    "goal": "Your goal is to get as far as possible in the game.",
    "role": "rogue",
    "actions": [
    "apply",
    "east",
    "north",
    "northeast",
    "northwest",
    "pickup",
    "south",
    "southeast",
    "southwest",
    "west"
    ],
    "map_size": {
    "width": 80,
    "height": 21
    },
    "initial_position": {
    "x": 41,
    "y": 18
    },
    "initial_entities": [
    "wall"
    ],
    "inventory_items": [
    "short sword",
    "dagger",
    "leather armor",
    "potion of sickness",
    "lock pick",
    "empty sack"
    ],
    "local_topology": {
```

```
    "walls_at": [
        "east",
        "north",
        "northeast",
        "south",
        "southeast",
        "southwest"
    ],
    "doors_at": [],
    "stairs_up_at": [],
    "stairs_down_at": [],
    "boulder_at": [],
    "fountain_at": [],
    "other": "{}"
    },
    "map_neighbors": {
    "north": ".",
    "northeast": ".",
    "east": ".",
    "southeast": ".",
    "south": ".",
    "southwest": ".",
    "west": ".",
    "northwest": "."
    },
    "passable_dirs": [
    "north",
    "northeast",
    "east",
    "southeast",
    "south",
    "southwest",
    "west",
    "northwest"
    ],
    "initial_movement_suggestions": {
    "blocked_dirs": [],
    "hazard_dirs": [],
    "suggested_first_moves": [
        "east",
        "north",
        "northeast",
        "northwest"
    ],
    "nearby_objectives": {
        "topology": {}
    },
    "fallback_actions": []
    },
    "action_mask": {
    "allowed": [
        "east",
        "north",
        "northeast",
        "northwest",
        "south",
        "southeast",
        "southwest",
        "west"
    ],
    "blocked": [],
    "hazard": [],
    "fallback": []
    },
    "similar_cases": [
```

```
    {
        "case_id": "34b90c20-541c-4148-bca7-6df9979f2b3e",
        "reward": 1.0,
        "outcome": "failure_or_partial",
        "lesson": "Always assess the surrounding topology for
            passable routes before acting. Prioritize movement toward
             open paths to maximize progress."
    },
    {
        "case_id": "30541a20-6805-4222-883c-3ea8537a4bd8",
        "reward": 1.0,
        "outcome": "failure_or_partial",
        "lesson": "Always assess your surroundings for walls before
            moving. Avoid repetitive actions that lead to dead ends
            and seek alternative routes to progress."
    }
    ]
},
"suggested_strategies": {
    "plan_outline": {
    "plan_title": "Navigating the Grid to Progress",
    "ordered_steps": [
        "Assess initial position and adjacent walls.",
        "Determine the safest path avoiding walls.",
        "Utilize available inventory for movement and defense.",
        "Plan movement towards open spaces or unexplored areas.",
        "Execute movement step by step, keeping track of the path.",
        "Monitor health and inventory status.",
        "Adapt strategy based on encountered obstacles."
    ],
    "pitfalls_to_avoid": [
        "Rushing movements without assessing surroundings.",
        "Ignoring the potential benefits of inventory items.",
        "Failing to track health status leading to unplanned deaths."
            ,
        "Becoming fixated on one direction despite walls blocking the
             way."
    ],
    "key_checks": [
        "Confirm no walls block planned movement direction.",
        "Reassess health and inventory after each turn.",
        "Ensure there are no dead ends or traps in the path ahead.",
        "Check if any enemies appear during movement."
    ]
    },
    "bulleted_tips": [
    "Survey adjacent passable tiles to uncover new paths.",
    "Avoid repeatedly moving in circles; change direction if blocked.
        ",
    "Focus on exploring directions that lead to visible objectives.",
    "Utilize your lock pick to open potential paths hidden behind
        walls.",
    "Keep your inventory organized; prioritize items based on
        immediate needs."
    ]
},
"spatial_priors": [
    "wall -> blocks_movement (support=280.5)"
],
"risk_and_interactions": [
    "Explore passable directions; avoid blocked paths.",
    "Keep escape routes clear; don't get cornered.",
    "Use the potion of sickness when HP is low for risky moves."
],
"reflex_tips": [
```

```
    "Avoid bumping into blocked directions: east, north, northeast,
        south, southeast, southwest. Pivot to open tiles.",
    "Walls/bars block movement; navigate around instead of repeating
        bumps."
],
"retrieval_metadata": {
    "task_id": "minihack:643759935e128b64511fe4c27717958dc1ebf583",
    "knowledge_query": "Your goal is to get as far as possible in the
         game. | role:rogue | entities:wall | inv:short sword,dagger,
        leather armor,potion of sickness,lock pick,empty sack | topo:
        walls:east,north,northeast,south,southeast,southwest; up:;
        down:; boulder:; fountain: | neighbors:N:.; NE:.; E:.; SE:.;
        S:.; SW:.; W:.; NW:. | passable:north,northeast,east,
        southeast,south,southwest,west,northwest | hazards:"
}
}
```