M*: Every Task Deserves Its Own Memory Harness

Wenbo Pan[1] **Shujie Liu**[2] **Xiangyang Zhou**[2] **Xianlong Wang**[1]*
Shiwei Zhang[2] **Bingjing Xu**[2] **Wanlu Shi**[2] **Xiaohua Jia**[1]
[1]City University of Hong Kong [2]Microsoft



Figure 1: **Evolved memory harnesses across tasks.** The radial tree shows how MSTAR evolves shared seed programs into task-specialized memory harnesses for LoCoMo, ALFWorld, HealthBench, and PRBench. The surrounding panels show representative task, memory, and agent behavior for each domain.

Abstract

Large Language Model (LLM) agents rely on a memory harness to write, organize, retrieve, and use past experience. A harness that works for one task often fails on another, because different tasks like conversations, embodied planning, or expert reasoning require different storage and retrieval behavior. To address this limitation, we introduce MSTAR, a method that automatically discovers task-optimized memory programs through reflective code evolution. Specifically, MSTAR represents a memory harness as an executable Python program that defines the *Schema*, *Logic*, and *Instruction* components, which are optimized jointly. We evaluate MSTAR on four tasks covering conversation, embodied planning, and specialized reasoning. Our results demonstrate that MSTAR improves performance over existing static memory harnesses across all evaluated tasks. Furthermore, the evolved memory programs exhibit structurally distinct processing mechanisms for each evaluated domain. These findings suggest that every task benefits from its own memory harness, and that memory-program search provides a concrete way to discover it.

Code: github.com/wbopan/mstar **Live demo:** mstar.wenbo.io

*Correspondence to: xianlong.wang@my.cityu.edu.hk

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

1 Introduction

Agentic systems powered by large language models (LLMs) need a memory harness to reuse past experience across diverse tasks [36, 22, 38, 30]. We use *memory harness* to refer to the components that store, organize, retrieve, and use this experience. As different task domains require knowledge in different structures, they call for different memory harnesses. For example, conversational agents need semantic maps of people, events, and preferences to remember user-specific facts (Figure 1, top right) [16], while embodied AI agents need reusable patterns from past trajectories to choose actions in new environments (Figure 1, bottom left) [26], and specialized reasoning domains may need structured databases (Figure 1, top left) [2]. These requirements encode different assumptions about what to store, how to organize it, and when to retrieve it. A harness built around one set of assumptions can therefore transfer poorly to another [27]. This raises a practical question: *How can we find the most suitable memory harness for a given task?*

To answer this question, we propose MSTAR, an evolutionary method that automatically discovers task-optimized memory harnesses for LLM agents. Specifically, MSTAR represents a memory harness as a *memory program*: a Python module with a fixed interface consisting of three components, *Schema*, *Logic*, and *Instruction*. The *Schema* defines the data formats for storage and retrieval, the *Logic* implements the `read` and `write` operations, and the *Instruction* guides the agent's interaction with the knowledge base. These components are jointly optimized through reflective code evolution. Prior works have also improved agent memory harnesses from experience [37, 4, 35, 1]. Most closely, MemEvolve [34] first showed that treating the memory system itself as an evolvable program can significantly improve its performance on downstream tasks. MSTAR builds on this direction but has a different emphasis. Rather than improving one memory system that works across tasks, it jointly optimizes Schema, Logic, and Instruction to find a specialized harness for each target task. To search this space, MSTAR uses two mechanisms: a reflector agent that debugs memory program failures and generates targeted improvements, and a population-based program search strategy that balances exploration and exploitation while enforcing constraints on program quality.

We apply MSTAR to four tasks covering conversation, embodied planning, and specialized reasoning, evaluating its performance against ten competitive baselines across four memory paradigms [16, 26, 3, 2]. Our results indicate that MSTAR achieves the best overall score and performs robustly on every task; in contrast, the baselines perform well only on specific tasks. Furthermore, we observe that the optimal memory programs for different tasks are distinct in both structure and procedure. By exploring a broad memory-program search space, MSTAR evaluates diverse memory programs to identify the task-optimized memory program. Together, these results support the view that every task benefits from its own memory harness, discovered as an executable memory program rather than chosen as a fixed design.

In summary, our main contributions are as follows:

1. We formulate task-optimized memory harness discovery as *memory-program search*, where an executable Python program jointly specifies Schema, Logic, and Instruction.
2. We propose MSTAR, an evolutionary method that optimizes memory harnesses from target-task feedback through reflective code evolution and achieves the strongest overall performance across four tasks compared to competitive generalist memory harnesses.
3. We perform in-depth analysis over the memory harness space and find that task-optimized memory harnesses are distinct in both structure and procedure across different tasks: conversation, embodied planning, and expert reasoning tasks all require different optimal memory harnesses.

2 Problem Setting

We consider a task $\mathcal{T} = (\mathcal{D}_{\text{e}}, \mathcal{D}_{\text{v}}, \mathcal{D}_{\text{t}})$, where $\mathcal{D}_{\text{e}} = \{e_1, \ldots, e_N\}$ is a set of episode trajectories representing past experiences, $\mathcal{D}_{\text{v}} = \{(q_1, y_1), \ldots, (q_M, y_M)\}$ is a validation set of queries with corresponding ground-truth answers, and $\mathcal{D}_{\text{t}}$ is a held-out test set used only for final reporting. An agent $\mathcal{A}$ must memorize information from these past episodes to answer queries from $\mathcal{D}_{\text{v}}$ and $\mathcal{D}_{\text{t}}$. Specifically, the agent is equipped with a knowledge base $\mathcal{K}$; it observes episodes in $\mathcal{D}_{\text{e}}$ and extracts knowledge items k to populate this knowledge base $\mathcal{K}$. At test time, given a query q_j, the agent accesses $\mathcal{K}$, which returns relevant context $c_j = \mathcal{K}.\texttt{read}(q_j)$. Building on this retrieved information,

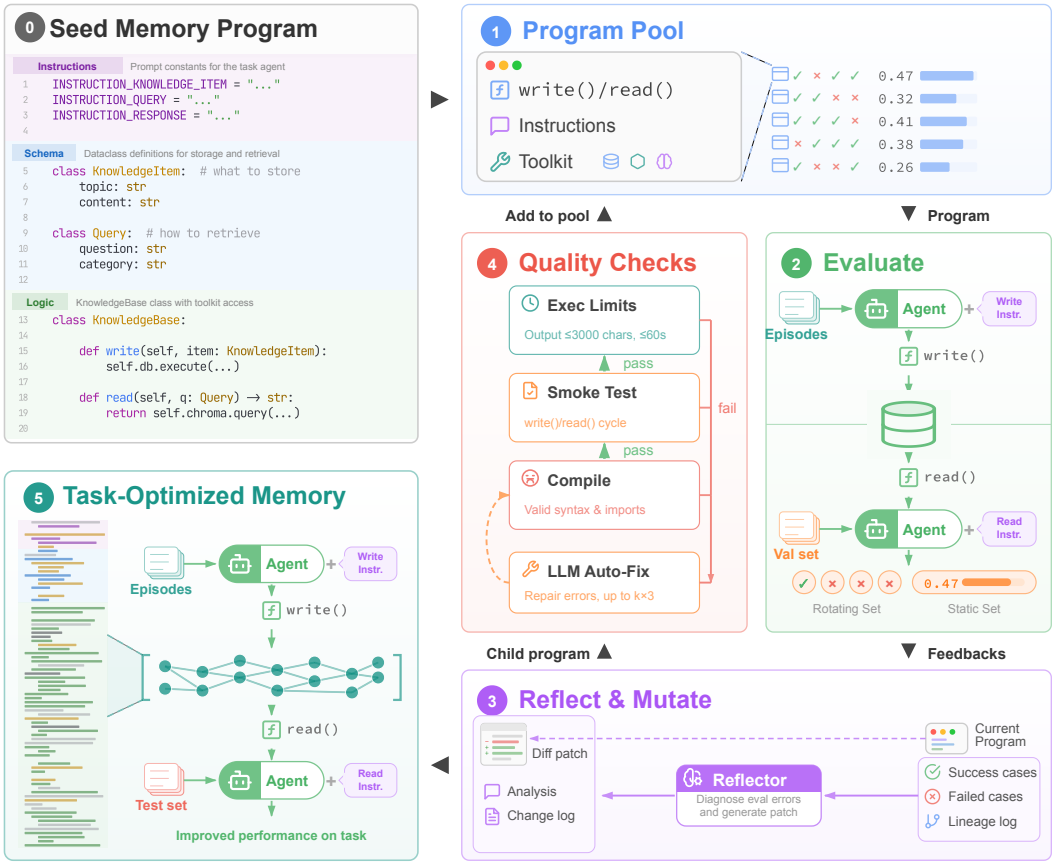

Figure 2: **System overview of MSTAR.** Numbered stages show the workflow from seed memory program (**0**) to final test evaluation (**5**): population pool (**1**), task evaluation (**2**), reflective code evolution (**3**), and compile/runtime quality checks (**4**).

the agent then produces its answer or takes an action conditioned on both the query and the context:

$$\hat{y}_j = \mathcal{A}(q_j, c_j). \tag{1}$$

Problem Definition: Task-Optimized Memory Program. While several prior works have proposed fixed memory methods for agent systems, these generalist structures transfer poorly across task types [27]. To this end, we formalize task-optimized memory harness optimization as search over executable memory programs. Each program P defines the input and output formats of the knowledge base, its underlying organization method such as a vector database or relational tables, and the agent's policy for when and how to access memory. Consequently, P describes an open program space that can express many storage and retrieval procedures. We define $\mathcal{J}(P)$ as the agent's performance on $\mathcal{D}_{\text{v}}$ when equipped with a knowledge base instantiated by P. The objective is to find the optimal memory program P^* that maximizes this performance metric:

$$P^* = \arg\max_P \mathcal{J}(P). \tag{2}$$

In the next section, we address how we can automatically discover the optimal memory program for a given task. Specifically, we propose a framework that uses reflective code evolution to search this program space by iterating and making targeted improvements starting from a seed program.

3 Method

Figure 2 gives an overview of the system MSTAR, a reflective code evolution method that automatically discovers task-optimized memory programs. Specifically, MSTAR consists of three main components: (1) representing the memory-program search space in code (Section 3.1), (2) reflective code evolution (Section 3.2), and (3) population-based program search (Section 3.3).

3.1 Representing the Memory-Program Search Space in Code

We represent each memory program as an executable Python module with a fixed interface. To define a broad memory-program search space, each memory program simultaneously controls three key components of the knowledge base: *Schema*, *Logic*, and *Instruction*. In addition, the program has access to a standardized *Toolkit*. A template for a memory program is shown in Figure 2.

- ***Schema*.**The Schema specifies the data formats that the knowledge base accepts for writing and querying. For example, the input format can range from structured data with timestamps to natural language insights. We implement the Schema using Python dataclasses. As a result, when the agent needs to write or query information, it instantiates these dataclass types and passes them into the memory program for parsing.
- ***Logic*.** The Logic defines the backend operations of the knowledge base when processing and storing Schema-formatted inputs and queries. This component can include operations such as computing embeddings, managing SQL tables, or calling an LLM for secondary data processing. Specifically, the Logic exposes `write` and `read` functions, which are executed whenever the agent performs write or read actions.
- ***Instruction*.** The Instruction defines how the agent interacts with the knowledge base. This includes how to extract information from observed episodes, how to formulate queries, and how to interpret the results returned by the knowledge base. We define the Instruction through constant prompt strings. These instruction constants are inserted into the system prompt when the agent executes corresponding write, query, and response actions.
- **Toolkit.** Within the memory-program search space, we allow the memory program to use various data structures and external tools. We provide a set of whitelisted components, including lists, heaps, relational databases, vector databases, and LLM endpoints. The Toolkit enables memory programs to implement complex storage and retrieval procedures.

3.2 Reflective Code Evolution

Representing a memory program using executable code places memory-program search in an enormous search space. Inspired by recent evolutionary algorithm discovery methods, we design a reflective code evolution process that iteratively refines memory programs using validation feedback [24, 5, 18]. Specifically, this process consists of three stages: sampling feedback from a validation loop, iterative refinement via a coding agent, and constraint checking with automated repair before the candidate is evaluated. An illustration is shown in Figure 2.

Sampling feedback from the validation loop. We use a set of episodes and validation queries to evaluate a memory program. The validation queries are partitioned into a *static* set, which remains constant across all iterations, and a *rotating* set, which changes in each iteration similar to mini-batches in training. During validation, the agent initiates a query to the knowledge base for each sample in both the rotating and static sets and uses the returned information to generate a response or take an action. The aggregated score on the static validation set serves as the performance metric for the current memory program. Meanwhile, the generated trajectories on the rotating set are used as feedback to improve the program later.

Coding agent iteration. We employ a coding agent to iteratively update the current memory program to improve performance and fix potential bugs. We provide the coding agent the execution trajectories from the read and write processes, underperforming samples from the rotating validation set, and the change logs from prior iterations along with their corresponding scores. Based on this information, the coding agent analyzes whether the read and write operations function as expected, identifies the root causes of underperforming cases, and determines which past improvements were effective. Consequently, the coding agent generates a code patch to apply to the current program and produces a new change log to guide future iterations. The complete prompt template is provided in Appendix G.

Constraint checks and automated repair. Before applying the updated memory program, the code evolution process executes a set of runtime checks to ensure its quality. If any check fails, the memory program and its corresponding error messages are resubmitted to the coding agent for repair. These checks prevent low-quality programs from consuming excessive computational resources, and any program that still fails after a fixed number of repair attempts is discarded.

Table 1: **Main results across four task families.** Columns report LoCoMo token F1 and LLM-judge score (L-J), ALFWorld success rate, HealthBench rubric score, and PRBench rubric score. Baselines are grouped by memory paradigm. Test sets contain 100 items for LoCoMo and each HealthBench split and 50 items for each ALFWorld and PRBench split. Best per column in **bold**, second best in **green**. † marks a significant lead in a paired bootstrap test ($p < 0.05$, Appendix J.1). L-J is not evaluated for MemEvolve.

	LoCoMo		ALFWorld		HealthBench		PRBench	
Method	**F1**	**L-J**	**Unseen**	**Seen**	**Data**	**Emerg.**	**Legal**	**Finance**
No Memory	0.036	0.030	0.738	0.640	0.242	0.429	0.431	0.269
Retrieval-based Systems *(Memorizing Episodes)*								
Vector Search	0.256	0.400	0.643	0.720	0.264	0.400	0.466	0.308
G-Memory	0.224	0.380	0.690	0.560	0.309	0.447	0.499	0.318
Mem0	**0.373**	**0.540**	0.738	0.640	0.216	0.413	0.450	0.321
Self-evolution Systems *(Memorizing Experiences)*								
Trajectory Retrieval	0.276	0.420	0.714	**0.780**	0.265	0.442	0.480	0.304
ReasoningBank	0.194	0.380	0.738	0.580	0.315	0.470	0.508	0.330
Dynamic Cheatsheet	0.124	0.190	0.619	0.480	0.286	**0.487**	0.474	0.327
Prompt-optimizing Systems *(Learned Prompts)*								
GEPA	0.132	0.190	**0.857**	0.720	0.304	0.466	**0.568**	**0.449**
GEPA + Vector Search	0.300	0.390	0.810	**0.820**	**0.327**	0.484	0.554	0.411
Program-evolving Systems *(Evolved Memory Code)*								
MemEvolve	0.204	–	0.760	0.620	0.267	0.456	0.484	0.317
MSTAR	**0.459**†	**0.610**	**0.881**†	0.700	**0.390**†	**0.493**	**0.660**†	**0.586**†

3.3 Population-Based Program Search

Unlike function search or prompt optimization, evaluating a memory program in MSTAR can be highly expensive. For any program change, the agent must regenerate episode knowledge and evaluate the program on validation samples. Therefore, the search procedure should explore the program space efficiently under a limited evaluation budget.

Instead of continuously updating a single program, which can cause the evolution algorithm to fall into local optima, we maintain a program pool. We initialize the search with a small set of simple seed programs (Appendix E). In each iteration, we sample a parent program from the pool with probability biased toward higher validation scores. Specifically, the probability $\Pr(P_i)$ of selecting program P_i from the pool is computed as:

$$\Pr(P_i) = \frac{\exp(\mathcal{J}(P_i)/\tau)}{\sum_j \exp(\mathcal{J}(P_j)/\tau)}, \tag{3}$$

where $\mathcal{J}(\cdot)$ is the static validation score (Eq. 2) and τ is the softmax temperature. The coding agent then applies a reflective mutation step to the selected parent, and the resulting child program is added back to the pool. This design balances exploration and exploitation, as top-performing programs have a greater chance of being selected for improvement, yet lower-scoring programs retain a non-zero probability of being explored. We use additional subset-selection procedures to reduce evaluation cost, and describe these implementation details in Appendix D.3.

Overall, MSTAR provides an efficient reflective code evolution procedure for memory-program search. In the following section, we provide an empirical analysis to validate our method.

4 Experimental Setup

We evaluate MSTAR on four benchmark families with seven domain configurations, covering long-term conversational question answering, embodied task completion, medical question answering, and professional legal and financial reasoning. For each configuration, we run 20 evolution iterations, select the best memory program on validation, and report performance on a held-out test split. We use

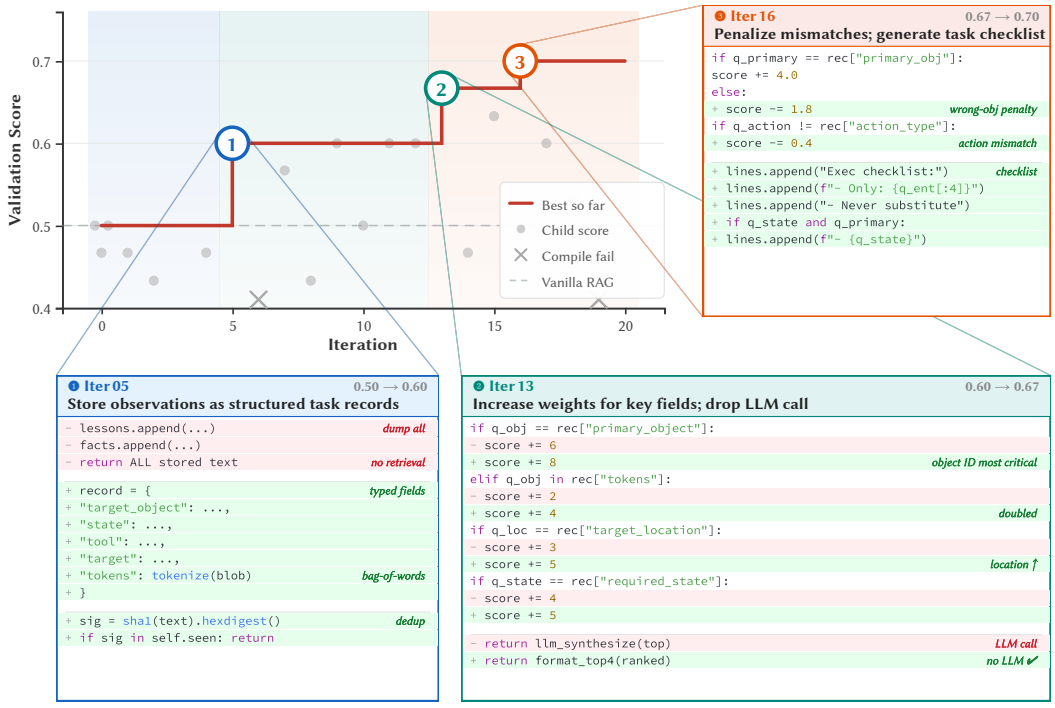

Figure 3: **ALFWorld evolution trajectory.** The main panel plots each child program's validation score and the best score so far across 20 evolution iterations, with crosses marking compile failures and the dashed line marking the Vanilla RAG baseline. Numbered panels highlight representative code changes that structure task records, reweight retrieval, and add mismatch penalties and execution checklists for selecting the next action.

GPT-5.4 Mini as the task agent and GPT-5.3-Codex as the coding agent in reflective code evolution [20, 19]. Additional dataset splits, hyperparameters, evaluation protocols, computational costs, and pseudocode are provided in Appendix B, Appendix D.2, Appendix H, Appendix D.1, and Appendix A.

Benchmarks. We use LoCoMo [16] to test long-term conversational memory, ALFWorld [26] to test memory use in embodied planning, HealthBench [3] to test rubric-graded health reasoning, and PRBench [2] to test professional legal and financial reasoning. Together, these benchmarks cover retrieval-heavy question answering, interactive decision making, and specialized reasoning under structured evaluation. Dataset construction details are provided in Appendix B, and metric details are provided in Appendix H for all seven benchmark configurations.

Baselines. We compare MSTAR with ten baselines, including a no-memory control and nine memory-based methods spanning four paradigms, as summarized in Table 1. These baselines cover direct answering without external memory, retrieval-based memory over stored observations, self-evolving memories that distill reusable experience, prompt-optimization methods with and without vector retrieval, and MemEvolve, which also evolves memory systems as code [13, 33, 21, 1, 34]. Except for Vector Search and No Memory, which serve as generic references, all baselines are published, human-designed memory systems or optimizers. All evolutionary baselines use the same task agent, coding model, data splits, and number of evolution rounds as MSTAR. Detailed baseline implementations are provided in Appendix C, with implementation notes for every baseline.

5 Results

We organize the analysis around three questions: whether memory-program search improves task performance, whether the resulting programs differ structurally across tasks, and which evolved components drive the gains.

Reflective code evolution progressively discovers better memory programs. Table 1 shows that MSTAR achieves the highest score on seven of eight reported metrics. Compared with the best baseline

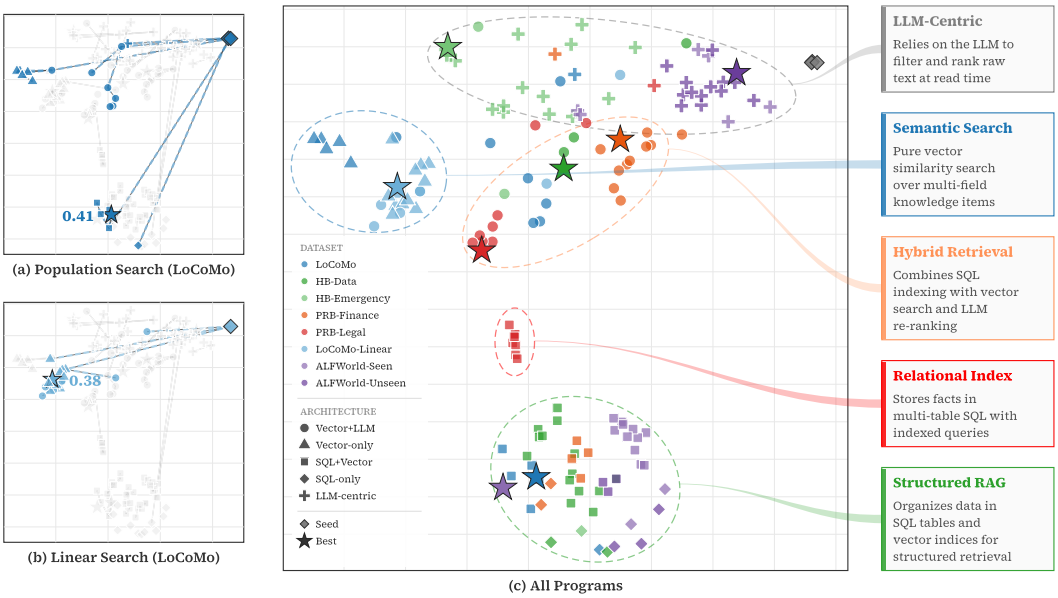

Figure 4: **Program embedding landscape.** Each evolved program is embedded with a code embedding model and projected to 2D via t-SNE [28]. **(a, b)** Population-based search (MSTAR) and linear search, with colored edges tracing parent-child lineage. **(c)** All programs from the evaluated configurations and the LoCoMo linear-search ablation, colored by configuration. Marker shapes denote the storage architecture discovered by evolution: circles for vector + LLM, triangles for vector-only, squares for SQL + vector, rotated squares for SQL-only, and crosses for LLM-centric designs; diamonds mark seed programs and stars mark the best program for each configuration.

in each winning metric, the relative improvement is up to 31%, with the largest gains on LoCoMo and PRBench. The strongest baseline varies by task, indicating that existing memory methods have task-dependent strengths. GEPA is competitive on ALFWorld and PRBench, while Mem0, GEPA+Vector Search, and Dynamic Cheatsheet lead the baselines on LoCoMo and HealthBench. In comparison, MSTAR achieves the best overall performance. MemEvolve, which also evolves memory code with the same coding model, scores below MSTAR on all seven configurations and is closest on HealthBench and PRBench. On five of the six configurations where MSTAR leads, its lead is statistically significant (Appendix J.1). Appendix J further shows that the gain comes mainly from search rather than from the coding model's prior, that MSTAR reaches the final validation score of GEPA with a fraction of its budget, and that the lead holds under an independent judge and other task models. This pattern suggests that executable code search reaches useful memory designs beyond the strongest fixed baselines. Figure 3 shows the ALFWorld evolution trajectory. The evolved memory program improves steadily over iterations, with the reflector producing task-specific changes at different stages of search. Early edits build a usable task-record schema that captures objects, locations, and state changes from demonstrations. Later edits refine retrieval weights and add action-selection checks, allowing the agent to retrieve more relevant trajectory patterns and choose actions more reliably.

Different tasks produce structurally distinct optimal memory programs. To understand how MSTAR adapts memory program structure to different tasks, we visualize the program embedding landscape accumulated during evolution (Figure 4). For each task and iteration, we first sanitize the program by replacing strings and variable names with placeholders so that the embedding emphasizes structural differences. We then embed the sanitized programs with a code embedding model and project them with t-SNE [28]. Because positions in this projection depend on the embedding model and t-SNE, we use the figure only to

Table 2: **Ablation study on LoCoMo.** Rows report the full system and variants that remove or alter one design choice; metric is token F1. Δ reports absolute change from the full system.

Variant	F1	Δ
MSTAR	0.459	—
− *Instruction*	0.353	−0.106
− *Code*	0.256	−0.203
Max parent selection	0.381	−0.078
− *Diversity*	0.318	−0.141

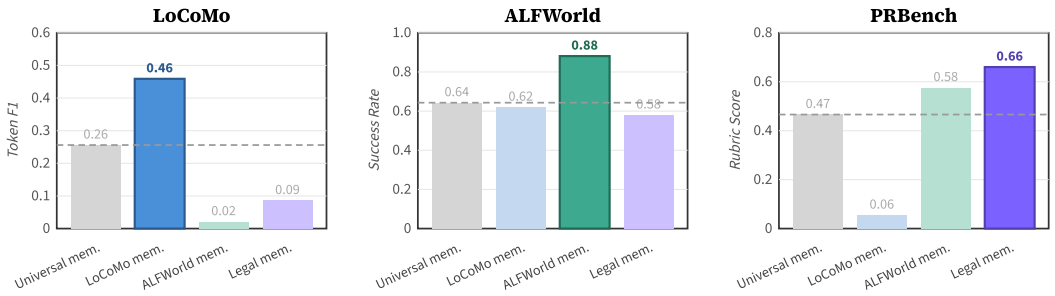

Figure 5: **Cross-task transfer of evolved memory programs.** Each panel fixes one target benchmark and compares memory programs evolved on different source benchmarks. Highlighted bars mark native-task programs, and the dashed line marks the universal seed baseline.

group programs by task and storage architecture, not
to trace how programs evolve. The projection places the best programs for different benchmark configurations in separate regions, with markers indicating different storage architectures. For example, the best ALFWorld program uses a compact SQLite action cache with read-time LLM synthesis. The best LoCoMo program uses ChromaDB semantic retrieval together with SQLite lexical and metadata scoring. Appendix F.4 reports structural summaries, and the supplementary material includes full code for all best programs.

To further test whether these structures are task-specific, we evaluate the best evolved program from each source task on other target tasks (Figure 5). In all three targets, the native evolved program outperforms transferred programs from other benchmarks. Five of six transferred programs perform worse than the universal seed baseline. This pattern suggests that memory program structure should be optimized together with the target task.

Joint evolution of structure and policy enables task-specific adaptation. Table 2 isolates key design choices on LoCoMo. Each variant removes one component of MSTAR: $-$ *Instruction* lets only the code evolve and keeps the prompt instructions fixed; $-$ *Code* does the opposite, evolving only the prompt instructions while keeping the code at the Vector Search seed; *Max parent selection* always picks the highest-scoring parent instead of sampling by softmax; and $-$ *Diversity* drops the population pool and mutates only the current best program at each iteration. $-$ *Code* gives the largest drop, reducing F1 by 0.203. $-$ *Instruction* reduces F1 by 0.106. The full system also scores above *Max parent selection* and $-$ *Diversity*. These results indicate that joint optimization of program Logic and Instruction gives the strongest LoCoMo result.

Evolution is stable across random seeds. A practical concern for evolutionary methods is sensitivity to random seeds, especially under a 20-iteration budget in which discovery quality can depend on early mutations. To evaluate this effect, Table 3 reports test scores from five independent runs on three benchmarks. Across all benchmarks, the coefficient of variation remains at or below 10%, which indicates stable outcomes with moderate variance. Across the 15 runs, 14 outperform the strongest baseline, supporting consistent gains across initializations. Appendix I.2 provides full per-seed scores and validation trajectories.

Table 3: **Stability across evolution seeds.** Mean and std. are computed over five independent stability runs with random seeds; metrics match the primary columns in Table 1. CV is the coefficient of variation in percent, Best Baseline is the strongest fixed baseline, and Win is the number of seeds above that baseline.

Benchmark	Mean$\pm$Std	CV	Best Baseline	Win
LoCoMo	$0.421_{\pm 0.042}$	10.0	0.373 (Mem0)	4/5
HB Data	$0.391_{\pm 0.023}$	5.9	0.327 (GEPA+VS)	5/5
PR Finance	$0.561_{\pm 0.032}$	5.7	0.449 (GEPA)	5/5

6 Discussion

In this section, we examine two properties of reflective code evolution: search diversity and category effects.

Table 4: **Per-category performance breakdown. (a)** ALFWorld Unseen success rate by task type. **(b)** LoCoMo token F1 by query type. n is the number of test episodes or test queries per category. Min reports the minimum score across shown categories. Best per column in **bold**.

	(a) ALFWorld Unseen						(b) LoCoMo				
	Simple (n=8)	**Clean** (n=11)	**Cool** (n=7)	**Heat** (n=6)	**2-Obj** (n=8)	**Min**	**Single** (n=18)	**Temp.** (n=18)	**Causal** (n=9)	**Open** (n=55)	**Min**
No Memory	0.88	0.82	0.57	0.83	0.62	0.57	0.02	0.00	0.11	0.04	0.00
Mem0	0.88	0.64	0.57	**1.00**	0.75	0.57	**0.33**	0.26	0.37	0.43	0.26
Trajectory Retrieval	0.75	0.73	0.71	0.67	0.75	0.67	0.18	0.33	0.34	0.28	0.18
GEPA + Vector Search	0.88	**1.00**	0.57	**1.00**	0.62	0.57	0.15	**0.38**	**0.39**	0.31	0.15
MSTAR	0.88	0.91	**1.00**	0.83	0.75	**0.75**	**0.33**	0.19	0.35	**0.51**	0.19

How does evolution explore the program space? Figure 4(c) provides another view of the search process. Evolved programs form several regions with different storage designs, including LLM-centric, vector-only, SQL-only, SQL+vector, and vector+LLM programs. Programs from different benchmarks often appear in the same regions. This pattern suggests that evolution can try common storage designs before adapting them to a task. Figure 4(a,b) further compares population-based search with a linear variant. The linear variant starts from one empty seed and repeatedly mutates only the current best program, which corresponds to the $-$ *Diversity* ablation in Table 2. Population search keeps multiple candidates active, so later mutations can draw from more than one candidate lineage. The LoCoMo ablation matches this interpretation: $-$ *Diversity* reduces test F1 from 0.459 to 0.318, and *Max parent selection* reaches 0.381 (Table 2). Together, these results indicate that the memory-program space is diverse and multi-modal. Population-based search explores this space effectively by maintaining multiple candidate lineages, improving the chance of finding a strong task-specific program under a small iteration budget.

Does evolution improve performance uniformly? Table 4 disaggregates the results in Table 1 by category (full per-category tables are in Appendix I.1). On ALFWorld Unseen, MSTAR raises the worst category score to 0.75, above the best baseline floor among the methods shown (0.67). This pattern suggests that the evolved action cache addresses failure modes that fixed designs leave exposed across task types. On LoCoMo, the gains are less uniform: MSTAR is strongest on open-domain questions and tied on single-hop recall, while Mem0 and GEPA+Vector Search remain stronger on temporal and causal questions. This split suggests that an aggregate objective can improve the overall score while leaving lower-frequency or harder categories underserved.

7 Related Work

Memory for LLM Agents. Prior work on agent memory builds external stores from interaction histories [36]: typical pipelines extract salient information, retrieve relevant entries for a new query, and update via consolidation or pruning [22, 38, 30]. Recent systems learn richer policies; MemSkill [35] selects natural-language skills via an RL controller, and AWM [31] induces reusable workflows from successful trajectories. These systems commit to a fixed memory representation at design time. MemEvolve [34] is the closest prior work to ours. It treats the memory system as an evolvable program, decomposes it into encode, store, retrieve, and manage modules, and lets a coding agent rewrite these modules, so its search space also covers a broad class of memory designs. MSTAR shares this view and also evolves executable memory code with a coding agent and constraint checks. However, MemEvolve focuses on memory designs that generalize across tasks and evaluates them on deep-research benchmarks, while MSTAR searches for a separate harness for each target task and evaluates it on conversation, embodied planning, and expert reasoning tasks.

Self-Evolving Agents. Self-evolving LLM agents improve from interaction experience without gradient-based updates [11]. ExpeL [37] distills trajectories into editable natural-language insights, FLEX [4] builds an experience library via continual reflection, and a complementary line automates agent design through meta-optimization (ADAS [12]), skill synthesis (Voyager [29]), and verbal self-critique (Reflexion [25]). These systems evolve text-level artifacts that the LLM must re-interpret at runtime; MSTAR instead evolves deterministic code, reducing interpretation variance and enabling explicit data structures and retrieval logic in each memory program.

Program Search with Language Models. Using LLMs as mutation operators has yielded solutions surpassing human heuristics in combinatorial optimization [24], neural architecture search [5], and

algorithmic discovery [18]. Among prompt-level methods, GEPA [1] mutates every prompt in a compound AI system via reflective natural-language updates with Pareto-front candidate selection. MSTAR applies the same paradigm to executable memory programs; because generated code can fail to compile or violate runtime constraints, we add a compile-fix loop and runtime violation recovery that are unnecessary in prompt evolution (Section 3.2).

8 Limitations and Future Work

Limitations. MSTAR has several limitations. First, the search signal can reward behavior that raises the metric without producing a strictly better harness. For example, the LLM judges of HealthBench and PRBench tend to prefer long answers, and the evolved harnesses on these benchmarks also instruct the agent to give long and detailed answers. Similarly, although the reflector prompt discourages hard-coded word lists, evolution still proposes string literals copied from the dataset text on every benchmark, most often on PRBench-Legal (Appendix J.7). Such literals can improve in-task performance, but they also limit transfer to other tasks (Figure 5). Second, MSTAR does not help on every configuration. On ALFWorld-Seen it trails GEPA + Vector Search, although its 95% interval contains the score of GEPA + Vector Search (Appendix J.1). On LoCoMo, systems designed specifically for long-term conversation, such as LightMem [10] and SimpleMem [14], report higher scores than the evolved harness, while our baselines were chosen to cover different memory paradigms rather than to be the strongest LoCoMo systems. Third, all evolution runs use one task agent and reflector pair (GPT-5.4 Mini and GPT-5.3-Codex). The evolved harnesses keep their lead under two other task models (Appendix J.4), but we have not evolved harnesses with a different or open-source coding model. Finally, each run re-ingests episodes and re-scores the validation set for every candidate, so one 20-iteration run costs up to $90 on rubric-graded benchmarks and up to 100 wall-clock hours on ALFWorld (Appendix D.1).

Future work. Because a previous harness can serve as the seed program, a natural next step is to warm-start evolution from a harness evolved on a related task and measure how much this reduces search cost. We also plan to stress-test evolved harnesses with noisy, stale, or contradictory memories, and to study search procedures and objectives that prefer reusable structure over benchmark-specific heuristics.

9 Conclusion

We introduced MSTAR, a method that discovers task-optimized memory programs for large language model agents through reflective code evolution. By representing a memory harness as executable Python code with Schema, Logic, and Instruction, MSTAR turns memory design into a memory-program search problem. Across four benchmarks, the evolved programs outperform strong memory baselines and develop distinct storage, retrieval, and action workflows for different domains. These results support the view that agent memory should be optimized for the target task, and that memory-program search provides a practical path toward this goal.

Acknowledgments and Disclosure of Funding

This work was supported by HK RGC Research Grants No. RFS2425-1S01 and C1043-24GF.

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=RQm2KQTM5r`. Oral. arXiv:2507.19457.

[2] Afra Feyza Akyürek, Advait Gosai, Chen Bo Calvin Zhang, Vipul Gupta, Jaehwan Jeong, Anisha Gunjal, Tahseen Rabbani, Maria Mazzone, David Randolph, Mohammad Mahmoudi Meymand, Gurshaan Chattha, Paula Rodriguez, Diego Mares, Pavit Singh, Michael Liu, Subodh Chawla,

Pete Cline, Lucy Ogaz, Ernesto Hernandez, Zihao Wang, Pavi Bhatter, Marcos Ayestaran, Bing Liu, and Yunzhong He. PRBench: Large-scale expert rubrics for evaluating high-stakes professional reasoning. *arXiv preprint arXiv:2511.11562*, 2025. doi: 10.48550/arXiv.2511.11562. URL https://arxiv.org/abs/2511.11562.

[3] Rahul K. Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, Johannes Heidecke, and Karan Singhal. HealthBench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*, 2025. doi: 10.48550/arXiv.2505.08775. URL https://arxiv.org/abs/2505.08775.

[4] Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jinsong Su, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. FLEX: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*, 2025. doi: 10.48550/arXiv.2511.06449. URL https://arxiv.org/abs/2511.06449.

[5] Angelica Chen, David Dohan, and David So. EvoPrompting: Language models for code-level neural architecture search. In *Advances in Neural Information Processing Systems*, volume 36, pages 7787–7817. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/184c1e18d00d7752805324da48ad25be-Paper-Conference.pdf.

[6] Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2318–2335, 2024. doi: 10.18653/v1/2024.findings-acl.137. URL https://aclanthology.org/2024.findings-acl.137/.

[7] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025. doi: 10.48550/arXiv.2504.19413. URL https://arxiv.org/abs/2504.19413.

[8] 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 *Computer Games*, volume 1017 of *Communications in Computer and Information Science*, pages 41–75. Springer, Cham, 2019. doi: 10.1007/978-3-030-24337-1_3. URL https://doi.org/10.1007/978-3-030-24337-1_3.

[9] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006. doi: 10.1016/j.ipl.2005.11.003. URL https://doi.org/10.1016/j.ipl.2005.11.003.

[10] Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. LightMem: Lightweight and efficient memory-augmented generation. In *International Conference on Learning Representations*, 2026.

[11] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. URL https://openreview.net/forum?id=CTr3bovS5F. Survey Certification. arXiv:2507.21046.

[12] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

[14] Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. SimpleMem: Efficient lifelong memory for LLM agents. *arXiv preprint arXiv:2601.02553*, 2026.

[15] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume I: Statistics*, pages 281–297, 1967. URL https://digicoll.lib.berkeley.edu/record/113015.

[16] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13851–13870, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

[17] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming*, 14:265–294, 1978. doi: 10.1007/BF01588971. URL https://doi.org/10.1007/BF01588971.

[18] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. doi: 10.48550/arXiv.2506.13131. URL https://arxiv.org/abs/2506.13131.

[19] OpenAI. GPT-5.3-Codex model. https://developers.openai.com/api/docs/models/gpt-5.3-codex, 2026. Accessed: 2026-05-02.

[20] OpenAI. GPT-5.4 mini model. https://developers.openai.com/api/docs/models/gpt-5.4-mini, 2026. Accessed: 2026-05-02.

[21] 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. In *The Fourteenth International Conference on Learning Representations*, 2026. doi: 10.48550/arXiv.2509.25140. URL https://openreview.net/forum?id=jL7fwchScm. Poster. arXiv:2509.25140.

[22] 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*, pages 1–22. ACM, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

[23] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264/.

[24] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

[25] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

[26] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. doi: 10.48550/arXiv.2010.03768. URL https://openreview.net/forum?id=0IOX0YcCdTn. arXiv:2010.03768.

[27] Alina Shutova, Alexandra Olenina, Ivan Vinogradov, and Anton Sinitsin. Evaluating memory structure in LLM agents. *arXiv preprint arXiv:2602.11243*, 2026. doi: 10.48550/arXiv.2602.11243. URL https://arxiv.org/abs/2602.11243.

[28] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. URL https://www.jmlr.org/papers/v9/vandermaaten08a.html.

[29] 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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

[30] Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. Augmenting language models with long-term memory. In *Advances in Neural Information Processing Systems*, volume 36, pages 74530–74543. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/ebd82705f44793b6f9ade5a669d0f0bf-Paper-Conference.pdf.

[31] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=NTAhi2JEEE.

[32] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025.

[33] Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026. doi: 10.48550/arXiv.2602.07755. URL https://arxiv.org/abs/2602.07755. ICLR 2026 Workshop on MemAgents.

[34] Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. MemEvolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025. doi: 10.48550/arXiv.2512.18746. URL https://arxiv.org/abs/2512.18746.

[35] Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026. doi: 10.48550/arXiv.2602.02474. URL https://arxiv.org/abs/2602.02474.

[36] Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025. doi: 10.1145/3748302. URL https://doi.org/10.1145/3748302.

[37] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://doi.org/10.1609/aaai.v38i17.29936.

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

Appendix

Table of Contents

A Reflective Code Evolution Algorithm 15

A.1 Evolution Loop . . . 15
A.2 Split Evaluation . . . 15

B Benchmarks and Dataset Details 16

B.1 Dataset Splits . . . 16
B.2 LoCoMo . . . 16
B.3 ALFWorld . . . 16
B.4 HealthBench . . . 16
B.5 PRBench . . . 16

C Baseline Methods 17

C.1 No Memory . . . 17
C.2 Retrieval-Based Systems . . . 17
C.3 Self-Evolution Systems . . . 17
C.4 Prompt-Optimizing Systems . . . 17
C.5 Program-Evolving Systems . . . 17

D Experiment Configuration 17

D.1 Computational Cost . . . 17
D.2 Hyperparameters . . . 18
D.3 Representative Subset Selection . . . 19

E Seed Programs 19

E.1 Vector Search . . . 20
E.2 LLM Summarizer . . . 20
E.3 Experience Learner . . . 20

F Evolved Programs 21

F.1 ALFWorld: Deterministic Action Cache . . . 21
F.2 LoCoMo: Multi-Signal Episodic Index . . . 22
F.3 HealthBench and PRBench Program Sketches . . . 23
F.4 Cross-Benchmark Structural Comparison . . . 24

G Prompt Templates and Mutation Interface 24

G.1 Task Agent Prompts . . . 24
G.2 Reflector Prompt . . . 25
G.3 Compile-Fix Prompt . . . 28

H Evaluation Protocols 28

H.1 Token F1 (LoCoMo) . . . 28
H.2 Binary Success (ALFWorld) . . . 28
H.3 Rubric-Based Scoring . . . 28

I Extended Results 29

I.1 Complete Per-Category Results . . . 29
I.2 Multi-Seed Stability Results . . . 30
J Additional Analyses 31
J.1 Significance of the Main Results . . . 31
J.2 Search versus the Coding Model's Prior . . . 31
J.3 Search Efficiency . . . 31
J.4 Transfer across Task Models . . . 32
J.5 Independent Judge . . . 32
J.6 Transfer to LongMemEval . . . 32
J.7 Audit of Hard-Coded Literals and Evaluation Loopholes . . . 32
J.8 Overlap between Splits . . . 33
K Broader Impact 33
L LLM Usage 33

A Reflective Code Evolution Algorithm

Algorithm 1 Reflective code evolution for memory-program search. The left panel gives the population-based evolution loop, and the right panel gives split evaluation for a candidate program.

(a) Evolution Loop

Require: Seed programs $\{S_1, \ldots, S_K\}$, budget B, temp. τ
Require: Episodes D_{e}, static val D_{v}, rotate pool D_{r}
Ensure: Best program P^*

$\mathcal{P} \leftarrow \emptyset$
for $i = 1$ **to** K **do**
  $(J_i, _) \leftarrow \textsc{Eval}(S_i, D_{\text{e}}, D_{\text{v}})$
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(S_i, J_i)\}$
end for
for $t = 1$ **to** B **do**
  $P_{\text{par}} \leftarrow \textsc{SoftmaxSample}(\mathcal{P}, \tau)$
  $\tilde{D} \leftarrow \textsc{Sample}(D_{\text{r}}, n_r)$
  $(_, \mathcal{R}) \leftarrow \textsc{Eval}(P_{\text{par}}, D_{\text{e}}, \tilde{D})$
  $P_{\text{mut}} \leftarrow \textsc{Mutate}(P_{\text{par}}, \mathcal{R})$
  $P' \leftarrow \textsc{CompileFix}^k(P_{\text{mut}})$ {$\perp$ if discarded}
  if $P' \neq \perp$ **then**
    $(J', _) \leftarrow \textsc{Eval}(P', D_{\text{e}}, D_{\text{v}})$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(P', J')\}$
  end if
end for
return $\arg\max_{(P,J)\in\mathcal{P}} J$

(b) Split Evaluation

Require: Program P, task agent $\mathcal{A}$, metric μ, threshold θ
Require: Episodes D_{e}, eval queries D_{eval}
Ensure: Score J, diagnostic cases $\mathcal{R} = (\mathcal{F}, \mathcal{S})$

// Phase 1: Knowledge ingestion
KB $\leftarrow P.\textsc{KnowledgeBase}$(toolkit)
for $d \in D_{\text{e}}$ **do**
  item $\leftarrow \mathcal{A}.\texttt{extract}(d,\ P.\textsc{InstructionKnowledgeItem})$
  KB.write(item, d)
end for
// Phase 2: Retrieval and scoring
$\mathcal{F}, \mathcal{S} \leftarrow \emptyset, \emptyset$
for $(q, y) \in D_{\text{eval}}$ **do**
  query $\leftarrow \mathcal{A}.\texttt{formulate}(q,\ P.\textsc{InstructionQuery})$
  $c \leftarrow$ KB.read(query)
  $\hat{y} \leftarrow \mathcal{A}.\texttt{respond}(q, c,\ P.\textsc{InstructionResponse})$
  $s \leftarrow \mu(\hat{y},\ y)$
  if $s < \theta$ **then**
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{(q, y, \hat{y}, c, s)\}$
  else
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{(q, y, \hat{y}, c, s)\}$
  end if
end for
return $\left(\frac{1}{|D_{\text{eval}}|}\sum s,\ \ (\mathcal{F}, \mathcal{S})\right)$

B Benchmarks and Dataset Details

B.1 Dataset Splits

Table 5 summarizes the data splits for each benchmark configuration.

Table 5: **Dataset split sizes.** Rows report train, validation, and test sizes for every benchmark split used by the evaluation pipeline.

Benchmark	Split	Train	Val	Test
LoCoMo	—	272 sessions	1,440 QA	100 QA
ALFWorld	Unseen	4,652 trajectories	145 ep.	50 ep.
	Seen	4,652 trajectories	150 ep.	50 ep.
HealthBench	Data Tasks	257	120	100
	Emergency	260	122	100
PRBench	Legal	100	100	50
	Finance	120	130	50

For LoCoMo, we exclude category 5 (adversarial and unanswerable questions), which tests refusal capability rather than memory retrieval, retaining 1,540 QA pairs across four categories [16]. We reserve 100 QA pairs for the held-out test split and use the remaining 1,440 QA pairs as validation data. For PRBench, hard items (`finance_hard` and `legal_hard`) are excluded before train, validation, and held-out test construction to avoid distortion of the fitness signal [2].

B.2 LoCoMo

LoCoMo [16] is a multi-session conversational question answering benchmark in which the agent must recover relevant facts from long dialogue histories. The dataset contains extended dialogues with timestamped sessions, and the questions cover single-hop recall, temporal reasoning, causal reasoning, and cross-session multi-hop reasoning. We use the provided conversation sessions as episodes and report token-level F1 together with an LLM-judge score.

B.3 ALFWorld

ALFWorld [26] evaluates embodied task completion in simulated household environments through text interaction. The agent must complete goals such as finding, heating, cleaning, and placing objects. We use expert trajectories from the training split as episodes and report success rate within a 50-step budget on both seen and unseen environment splits.

B.4 HealthBench

HealthBench [3] is a medical question answering benchmark with professional rubric-based evaluation. Each sample includes a multi-turn clinical dialogue, an ideal completion, and structured pass or fail criteria written by medical professionals. We evaluate two categories: health data interpretation and emergency referral recognition. Because HealthBench does not provide a training split, we construct episode pools from a held-out portion of the benchmark data and keep evaluation queries disjoint from episode construction. We filter by category first, then use 54% of the category data for episode construction and reserve 100 examples for held-out testing. We report rubric score, computed as the fraction of criteria marked positive by an LLM judge.

B.5 PRBench

PRBench [2] evaluates professional reasoning in legal analysis and financial valuation. Each sample contains a task prompt, an expert reference, and an importance-weighted rubric. Similar to HealthBench, PRBench does not provide a training split, so we construct episodes from a held-out portion and evaluate on disjoint query sets. We filter out the hard subsets before splitting, use 40% of the remaining data for episode construction, and reserve 50 examples for held-out testing. For

consistency, we apply the same rubric-based scoring protocol used for HealthBench to both PRBench splits in our evaluation.

C Baseline Methods

We compare MSTAR against a no-memory control and nine memory-based baselines spanning retrieval-based systems, self-evolution systems, prompt-optimizing systems, and program-evolving systems.

C.1 No Memory

The no-memory control answers each task directly without any external memory store or retrieved context during evaluation.

C.2 Retrieval-Based Systems

These methods store raw observations and retrieve relevant context by similarity. *Vector Search* stores each observation in a vector collection and retrieves top-k items by embedding cosine similarity, without task-specific parsing [13]. *G-Memory* [33] augments vector retrieval with a hierarchical graph over related tasks implemented with SQLite. *Mem0* [7] extracts atomic facts from observations and maintains them with LLM-driven add, update, and delete operations as new observations arrive.

C.3 Self-Evolution Systems

These methods distill reusable knowledge from experience instead of storing raw episodes directly. *Trajectory Retrieval* [33] retrieves full episode trajectories by embedding similarity. *Reasoning-Bank* [21] extracts key insights from each episode and retrieves them with similar task descriptions. *Dynamic Cheatsheet* [33] maintains a single global summary that is iteratively rewritten after each new experience.

C.4 Prompt-Optimizing Systems

These methods improve agent behavior by evolving the system prompt. *GEPA* [1] optimizes prompts through reflective mutation with Pareto-based candidate selection. We evaluate two variants: *GEPA* and *GEPA + Vector Search*. The second variant adds a vector database to GEPA because the original GEPA setup does not include a persistent knowledge base, which can limit performance on memory-intensive tasks.

C.5 Program-Evolving Systems

MemEvolve [34] evolves a memory system as a program composed of encode, store, retrieve, and manage modules, which a coding agent rewrites across rounds. We use its official implementation and the hyperparameters reported in the original paper, and we replace its optimization model with GPT-5.3-Codex to match MSTAR. On ALFWorld, MemEvolve stopped early after one (Unseen) or two (Seen) of its three rounds because no new system passed its own validator, and we report the best system found up to that point.

D Experiment Configuration

D.1 Computational Cost

Table 6 reports the computational cost of running MSTAR evolution and baselines across all benchmarks. We break down API calls by role: the *task agent* (knowledge extraction, query generation, and answer generation), the *reflector* (code mutation via LLM reflection), the *judge* (per-criterion rubric scoring for HealthBench and PRBench), and the *toolkit* (any LLM call issued from inside the memory pipeline at write or read time, including evolved `write()`/`read()` methods, baseline retrieval steps, and embedding-driven preprocessing helpers). All experiments use `gpt-5.4-mini` as

the task agent, judge, and toolkit model, and `gpt-5.3-codex` as the reflector model [20, 19]. Cost estimates use Azure OpenAI pricing: \$0.75/\$4.50 per 1M input/output tokens for `gpt-5.4-mini`, and \$1.75/\$14.00 per 1M input/output tokens for `gpt-5.3-codex` [20, 19]. Note that `gpt-5.3-codex` is a reasoning model; its output token counts include internal reasoning tokens, which are billed at the same rate as visible output tokens when computing the estimates below. Token columns report GPT-family LLM tokens. BGE-M3 embedding tokens used for vector search and subset selection are tracked separately from the GPT token and dollar estimates.

Table 6: **Computational cost of evolution runs and representative baselines.** Rows report iteration count, API calls by role, GPT-family token counts, estimated dollar cost, and wall-clock time. Wall-clock time includes environment interaction overhead for ALFWorld runs.

Benchmark	Config	Iter.	Task	Reflect.	Judge	Toolkit	Input	Output	Cost	Time
LoCoMo	MSTAR	20	5802	48	—	135	4.9M	4.2M	\$25.43	5.1h
	Vector Search	0	200	0	—	272	294K	71K	\$0.54	2m
	No Memory	0	200	0	—	272	219K	64K	\$0.45	1m
ALFWorld (Unseen)	MSTAR	20	3355	25	—	443	7.5M	1.4M	\$13.92	100h
	Vector Search	0	80	0	—	497	711K	50K	\$0.76	14m
	No Memory	0	80	0	—	497	708K	50K	\$0.76	11m
ALFWorld (Seen)	MSTAR	20	2420	28	—	494	5.6M	1.1M	\$11.94	101h
	Vector Search	0	80	0	—	497	692K	49K	\$0.74	11m
	No Memory	0	50	0	—	224	264K	25K	\$0.31	0m
HB-Emergency	MSTAR	20	4272	31	21370	2219	21M	6.1M	\$47.20	4.3h
	Vector Search	0	45	0	1102	423	830K	145K	\$1.28	6m
	No Memory	0	54	0	1102	412	761K	137K	\$1.19	1m
HB-DataTasks	MSTAR	20	4204	24	15006	1575	18M	6.3M	\$44.64	2.1h
	Vector Search	0	24	0	1097	433	950K	177K	\$1.51	7m
	No Memory	0	34	0	1105	423	836K	171K	\$1.40	2m
PR-Legal	MSTAR	20	3065	28	19984	1242	48M	11M	\$90.19	15h
	Vector Search	0	15	0	958	185	818K	130K	\$1.20	6m
	No Memory	0	15	0	950	185	712K	117K	\$1.06	1m
PR-Finance	MSTAR	20	1795	31	18602	2497	47M	10M	\$84.81	2.6h
	Vector Search	0	8	0	845	212	740K	153K	\$1.24	6m
	No Memory	0	13	0	895	207	646K	139K	\$1.11	11m

Call counts include recorded API attempts. Token totals sum provider usage fields when available; failed attempts without usage metadata contribute to call counts only.

The total evolution cost ranges from \$12–90 per benchmark over 20 iterations, depending primarily on the evaluation protocol. Rubric-graded benchmarks (HealthBench and PRBench) are substantially more expensive because each validation item requires multiple LLM judge calls for per-criterion scoring, making the judge the dominant cost component (60–80% of total calls). Token-F1 benchmarks (LoCoMo) and binary-success benchmarks (ALFWorld) avoid judge calls entirely, keeping evolution under \$26. ALFWorld's long wall-clock time (100h) is dominated by TextWorld environment interaction; API latency contributes less. Although the reflector makes only 24–48 calls per run, it uses the more expensive `gpt-5.3-codex` model (\$14/1M output tokens vs. \$4.50 for `gpt-5.4-mini`) and produces long code patches, so it accounts for 7–24% of total cost depending on the benchmark.

Compared to baselines, the evolution overhead is a one-time cost: once the best program is found, inference uses the same number of calls as any baseline. The per-query inference cost of an evolved program equals that of the corresponding baseline configuration (No Memory or Vector Search) plus any toolkit calls the evolved program makes, typically 0–2 additional LLM calls per query.

D.2 Hyperparameters

Table 7 lists the key hyperparameters used across all experiments.

Table 7: **Hyperparameters for MSTAR evolution.** Rows list shared settings and benchmark-specific ranges used in the main experiments.

Parameter	Value
Evolution iterations	20
Population seed programs	3
Selection strategy	Softmax ($\tau = 0.15$)
Static validation subset	32–60 items (k-means clustering)
Rotating validation subset	5 items per iteration
Train-to-validation ratio	2 (train episodes = 2× validation items)
Embedding model (subset selection)	BGE-M3 [6]
Reflector model	GPT-5.3-Codex (reasoning: medium)
Task agent / toolkit model	GPT-5.4-mini
Toolkit LLM call budget	1 call per `read`/`write` invocation
Compile-fix attempts (k)	3
Evaluator thread pool	64 workers
LLM retry attempts	3 (exponential backoff)

D.3 Representative Subset Selection

We construct representative subsets to reduce evaluation cost while preserving coverage of the full validation set. For the static validation subset, we embed all validation samples, cluster them into n groups using k-means [15] with a fixed seed, and select the sample closest to each cluster centroid to form the fixed static validation set. This procedure covers diverse regions of the validation distribution more effectively than random sampling under the same evaluation budget. For the rotating validation subset used as reflection context, we re-cluster the remaining validation pool at each iteration with a different k-means seed (the iteration index added to a base seed), and select the sample closest to each new cluster centroid; varying the seed perturbs the cluster boundaries, so the chosen samples differ across iterations even though both clustering and centroid selection are deterministic given a seed. The static subset determines fitness, while the rotating subset supplies diverse examples for mutation.

For episode subset selection, we use a facility-location objective to select M training episodes that are relevant to the validation samples [17]. Given validation samples V and candidate episodes E, we select a subset $S \subseteq E$ of size M to maximize:

$$\max_{S \subseteq E, |S|=M} \sum_{v \in V} \max_{e \in S} \mathrm{sim}(v, e), \tag{4}$$

where $\mathrm{sim}(\cdot, \cdot)$ denotes embedding similarity. We use greedy facility-location selection in the same embedding space [17].

E Seed Programs

The program pool is initialized with three structurally diverse seeds designed to cover different points in the memory-program search space. All three share identical instruction constants but differ in storage strategy and retrieval logic. Table 8 summarizes their key design choices.

Table 8: **Seed program comparison.** Rows summarize each seed program's storage strategy, retrieval strategy, and read-time LLM use.

Seed	Storage	Retrieval	LLM in `read`
Vector Search	ChromaDB chunks	Top-k cosine similarity	No
LLM Summarizer	In-memory list	Concatenate + LLM summarize	Yes (1 call)
Experience Learner	Separate lesson/fact lists	Return all	No

E.1 Vector Search

The Vector Search seed (Listing 1) implements vanilla RAG [13]: raw text is split into 500-character paragraph-aligned chunks and stored in a ChromaDB collection; `read()` retrieves the top-5 most similar documents by embedding cosine similarity. No LLM call is used at retrieval time.

E.2 LLM Summarizer

The LLM Summarizer seed (Listing 2) stores raw text in an in-memory list without any preprocessing. At retrieval time, all stored text is concatenated (up to 30,000 characters) and passed to the toolkit LLM alongside the query for query-focused summarization. This seed tests whether neural synthesis at read time can compensate for the lack of structured indexing.

E.3 Experience Learner

The Experience Learner seed (Listing 3) extracts a general lesson and a specific fact from each observation, storing them in separate lists. At retrieval time, it returns all stored lessons and facts (truncated to 500 characters each), ignoring the query entirely. This seed explores whether dual-track extraction with full recall outperforms selective retrieval.

Listing 1: Seed 1: Vector Search.

```
class KnowledgeBase:
    """Vanilla RAG: store text chunks in ChromaDB, retrieve by semantic similarity."""
    def __init__(self, toolkit):
        self.toolkit = toolkit
        self.collection = toolkit.chroma.get_or_create_collection("knowledge")
        self._doc_id = 0

    def write(self, item: KnowledgeItem, raw_text: str) -> None:
        chunks = self._chunk(raw_text, max_chars=500)
        for chunk in chunks:
            self.collection.add(documents=[chunk], ids=[f"doc_{self._doc_id}"])
            self._doc_id += 1

    def read(self, query: Query) -> str:
        if self._doc_id == 0:
            return "No information stored."
        results = self.collection.query(
            query_texts=[query.query_text], n_results=min(5, self._doc_id))
        docs = results["documents"][0] if results["documents"] else []
        return "\n\n".join(docs)[:3000] if docs else "No relevant information found."
```

Listing 2: Seed 2: LLM Summarizer.

```
class KnowledgeBase:
    """LLM-powered query-focused summarization over stored raw texts."""
    def __init__(self, toolkit):
        self.toolkit = toolkit
        self.raw_texts: list[str] = []

    def write(self, item: KnowledgeItem, raw_text: str) -> None:
        self.raw_texts.append(raw_text)

    def read(self, query: Query) -> str:
        if not self.raw_texts:
            return "No information stored."
        combined = "\n\n".join(self.raw_texts)[:30000]
        messages = [{"role": "user", "content":
            f"Given the following query, summarize ONLY the relevant information "
            f"from the provided texts. Be concise and factual.\n\n"
            f"Query: {query.query_text}\n\nTexts:\n{combined}"}]
        try:
            result = self.toolkit.llm_completion(messages)
        except Exception:
            result = combined
        return result[:3000]
```

Listing 3: Seed 3: Experience Learner.

```
@dataclass
class KnowledgeItem:
    lesson_learned: str = field(
        metadata={"description": "A general lesson or pattern learned from the text"})
    fact_to_remember: str = field(
        metadata={"description": "A specific fact worth remembering from the text"})

class KnowledgeBase:
    """Experience-driven learner that stores lessons and facts, returns all on read."""
    def __init__(self, toolkit):
        self.toolkit = toolkit
        self.lessons: list[str] = []
        self.facts: list[str] = []

    def write(self, item: KnowledgeItem, raw_text: str) -> None:
        self.lessons.append(item.lesson_learned)
        self.facts.append(item.fact_to_remember)

    def read(self, query: Query) -> str:
        if not self.lessons and not self.facts:
            return "No information stored."
        lessons_text = "\n".join(self.lessons)[:500]
        facts_text = "\n".join(self.facts)[:500]
        return f"Lessons:\n{lessons_text}\n\nFacts:\n{facts_text}"[:3000]
```

F Evolved Programs

We present the best evolved programs for two representative benchmarks, selected to illustrate the structural diversity described in Section 5. Full source code for all seven benchmark configurations is included in the released experiment artifacts under each run's `programs/` directory.

F.1 ALFWorld: Deterministic Action Cache

The best ALFWorld program (Listing 4, Listing 5) constructs a deterministic action cache using SQLite. It stores structured fields extracted from expert demonstrations—target object, destination, required state change (clean/cool/heat/examine/toggle), action hints, and failure modes—and retrieves them via a weighted scoring function that combines token overlap with exact-match bonuses for object, location, and state. The program includes a `_canonical_state()` normalizer that maps diverse surface forms ("rinse", "wash" → "clean"; "chill", "cold" → "cool") to canonical labels, and a fallback parser that extracts task metadata from raw text when the LLM's structured extraction is sparse. A single LLM call at read time synthesizes the top-6 retrieved memories into concise step-by-step guidance for the task agent to follow. This description corresponds to the best ALFWorld Unseen program. The best ALFWorld Seen program uses the same SQLite-backed design family, but its `read()` method returns deterministic ranked guidance without a toolkit LLM call.

The key architectural properties are:

- **SQLite-only storage**, with zero vector retrieval (ChromaDB unused).
- **Canonical state normalization** unifies synonymous state descriptions (6 canonical states).
- **Keyword-based scoring** with exact-match bonuses (+6 for object, +5 for location, +5 for state) and recency tiebreaker for stable ranking.
- **Schema**: 8 structured fields per memory item (task_summary, task_type, target_object, target_location, required_state, action_hint, failure_mode, keywords).

Listing 4: ALFWorld evolved `write()` — canonical state extraction and fallback parsing.

```
def _canonical_state(self, text):
    t = (text or "").lower()
    if any(k in t for k in ["rinse", "wash", "clean"]): return "clean"
    if any(k in t for k in ["cool", "chill", "cold", "refrigerat"]): return "cool"
    if any(k in t for k in ["heat", "warm", "hot"]): return "heat"
    if any(k in t for k in ["examine", "inspect", "look at"]): return "examine"
```

```
    if any(k in t for k in ["toggle", "turn on", "light"]): return "toggle"
    return ""

def write(self, item, raw_text):
    task_summary = self._clean(item.task_summary)
    task_type = self._clean(item.task_type)
    target_object = self._clean(item.target_object)
    target_location = self._clean(item.target_location)
    action_hint = self._clean(item.action_hint)
    failure_mode = self._clean(item.failure_mode)
    required_state = self._canonical_state(item.required_state) or \
        self._canonical_state(f"{task_summary} {task_type} {raw_text}")
    # Fallback: extract from raw task metadata when LLM fields are sparse
    if not target_object:
        m = re.search(r"object_target:\s*([A-Za-z0-9_]+)", raw_text)
        if m: target_object = m.group(1)
    kw = set()
    for part in [task_summary, task_type, target_object, target_location]:
        kw |= self._tokenize(part)
    for k in item.keywords: kw |= self._tokenize(k)
    self.db.execute("INSERT INTO memories (...) VALUES (...)",
        (task_summary, task_type, target_object, target_location,
         required_state, action_hint, failure_mode, json.dumps(sorted(kw)),
         raw_text[:2000]))
```

Listing 5: ALFWorld evolved `read()` — weighted scoring with exact-match bonuses.

```
def read(self, query):
    rows = self.db.execute("SELECT id, task_summary, task_type, "
        "target_object, target_location, required_state, action_hint, "
        "failure_mode, keywords_json FROM memories").fetchall()
    q_obj = self._clean(query.target_object)
    q_loc = self._clean(query.target_location)
    q_state = self._canonical_state(query.required_state)
    q_tokens = self._tokenize(query.request) | self._tokenize(q_obj) | self._tokenize(q_loc)
    scored = []
    for row in rows:
        (row_id, _, _, target_object, target_location,
         required_state, _, _, keywords_json) = row
        mem_kw = set(json.loads(keywords_json or "[]"))
        score = len(q_tokens & mem_kw)                          ①
        if q_obj == (target_object or ""): score += 6           ②
        if q_loc == (target_location or ""): score += 5         ③
        if q_state == (required_state or ""): score += 5        ④
        score += min(row_id, 1000) / 100000.0                   ⑤
        scored.append((score, row))
    selected = sorted(scored, reverse=True, key=lambda x: x[0])[:6]
    # One LLM call: synthesize top memories into step-by-step guidance
    synthesized = self.toolkit.llm_completion([...])            ⑥
    return synthesized[:3000]
```

① Token overlap ② Exact object match ③ Exact location match ④ State match ⑤ Recency tiebreaker ⑥ LLM synthesis

F.2 LoCoMo: Multi-Signal Episodic Index

The best LoCoMo program builds a hybrid memory combining SQLite and ChromaDB (Listing 6, Listing 7). Each observation is parsed into 7 structured metadata fields (participants, organizations, activities, key facts, relation facts, named entities) and stored in both a relational table and a vector collection. At retrieval time, the program fuses semantic similarity scores from ChromaDB with lexical overlap scores from SQLite, applies person-focused boosting when the query targets a specific individual, limits per-source diversity (at most 2 chunks from any single dialogue), and separately ranks extracted facts by query relevance. The final output combines top-ranked candidate facts and relevant text excerpts for answer generation.

The key architectural properties are:

- **425 source lines**, with the largest hybrid retrieval procedure among the best programs.
- **Dual storage**: SQLite for structured metadata + ChromaDB for semantic search.
- **7 metadata fields** per item including relation facts (subject-verb-object triples).

- **Source diversity cap**: maximum 2 chunks per source document, preventing any single conversation from dominating retrieval for a query.
- **Two-tier output**: candidate facts (direct answer candidates) and relevant excerpts (contextual evidence for grounded answer generation in responses).

Listing 6: LoCoMo evolved schema — 7 structured metadata fields.

```
@dataclass
class KnowledgeItem:
    summary: str = field(metadata={"description": "Short summary"})
    participants: list[str] = field(
        metadata={"description": "People explicitly discussed in the text"})
    organizations: list[str] = field(
        metadata={"description": "Organizations, groups, or institutions"})
    activities: list[str] = field(
        metadata={"description": "Activities, hobbies, events mentioned"})
    key_facts: list[str] = field(
        metadata={"description": "Atomic factual statements useful for QA"})
    relation_facts: list[str] = field(
        metadata={"description": "Subject-verb-object facts"})
    named_entities: list[str] = field(
        metadata={"description": "Exact named items: orgs, games, places"})
```

Listing 7: LoCoMo evolved `read()` excerpt — hybrid scoring and source diversity.

```
def read(self, query):
    # Phase 1: Semantic retrieval from ChromaDB
    results = self.collection.query(query_texts=[query_text], n_results=24)
    ids = results["ids"][0]; dists = results["distances"][0]
    for rank, (doc_id, dist) in enumerate(zip(ids, dists)):
        semantic_scores[doc_id] = 1.25 / (rank + 1) + 1.0 / (1 + dist)

    # Phase 2: Lexical scoring from SQLite
    for row in rows:
        lexical = self._overlap_score(query_tokens, doc_tokens)
        person_boost = 1.2 if focus_person in participants else 0.0
        scores[doc_id] += 2.1 * lexical + 1.0 * info_overlap + person_boost

    # Phase 3: Source diversity enforcement
    source_counts = collections.Counter()
    for doc_id, _ in ranked_pairs:
        source_id = row_by_id[doc_id]["source_id"]
        if source_counts[source_id] >= 2: continue     ①
        ranked_ids.append(doc_id)
        source_counts[source_id] += 1

    # Phase 4: Separate fact ranking
    for doc_id in ranked_ids:
        for fact in relation_facts + key_facts:
            fscore = overlap(query_tokens, fact_tokens)
            if focus_person in fact: fscore += 0.5      ②
            fact_candidates.append((fscore, fact))

    # Output: candidate facts + relevant excerpts
    return "Candidate facts:\n..." + "\nExcerpts:\n..."  ③
```

① Max 2 chunks per source ② Person-focused boost ③ Two-tier output format

F.3 HealthBench and PRBench Program Sketches

The HealthBench and PRBench programs evolve toward criterion-aware retrieval rather than broad episodic recall. The HealthBench variants use SQLite tables keyed by clinical topic, urgency signals, patient attributes, and rubric-relevant safety notes; the Data Tasks variant also uses ChromaDB for semantic retrieval. At read time, they prioritize memories that match the current clinical intent and then use one toolkit LLM call to synthesize concise guidance for the task agent. This structure reflects the benchmark's grading protocol: answers must cover medically relevant criteria while avoiding unsafe or unsupported advice.

The PRBench variants emphasize domain-specific decomposition. The legal program stores issue statements, governing rules, factual predicates, counterarguments, and citation-like anchors, then retrieves by matching the requested legal theory and factual pattern. The finance program stores assumptions, valuation drivers, comparable-company cues, risk factors, and calculation notes, then combines structured scoring with vector retrieval for rubric-targeted evidence. These programs illustrate the same pattern as ALFWorld and LoCoMo: each task induces a different schema and retrieval procedure.

F.4 Cross-Benchmark Structural Comparison

Table 9 summarizes the architectural differences across all four benchmark domains, illustrating how reflective code evolution discovers structurally distinct solutions for each task.

Table 9: **Structural comparison of best evolved programs across benchmark configurations.** Rows report program size, storage backends, read-time LLM use, schema size, and test score.

Benchmark	Lines	SQLite	ChromaDB	LLM in `read`	Schema fields	Test
LoCoMo	425	Yes	Yes	No	7	0.459
ALFWorld (Unseen)	276	Yes	No	Yes	8	0.881
ALFWorld (Seen)	441	Yes	No	No	8	0.700
HB-Emergency	393	Yes	No	Yes	7	0.493
HB-Data	769	Yes	Yes	Yes	12	0.390
PR-Legal	1184	Yes	Yes	Yes	22	0.660
PR-Finance	509	Yes	Yes	Yes	10	0.586

Lines count the released best-program source files.

G Prompt Templates and Mutation Interface

This section documents the key LLM prompts used in MSTAR. We categorize prompts by their role in the system: task agent prompts (used during evaluation) and reflector prompts (used during evolution).

G.1 Task Agent Prompts

The task agent interacts with evolved memory programs through three fixed prompt templates. These templates are parameterized by the program's `INSTRUCTION_*` constants, meaning evolution can steer the agent's behavior by modifying these constants without changing the prompt structure.

Knowledge item generation. Given raw text and a `KnowledgeItem` schema, the task agent extracts structured information for a `write` call:

```
{INSTRUCTION_KNOWLEDGE_ITEM}

Text: {raw_text}

The KnowledgeItem must conform to this schema:
{schema}

Output ONLY a valid JSON object matching the schema fields. No explanation.
```

Query generation. Given a question and a `Query` schema, the task agent formulates a retrieval query for a `read` call:

```
{INSTRUCTION_QUERY}

Question: {question}

The query must be a JSON object matching this schema:
{schema}

Respond with the JSON only.
```

Answer generation. Given retrieved memory, the task agent generates an answer. The `ALWAYS_ON_KNOWLEDGE` constant, if non-empty, is prepended to the retrieved content:

```
<retrieved_memory>
{ALWAYS_ON_KNOWLEDGE}

{retrieved}
</retrieved_memory>

{INSTRUCTION_RESPONSE}
```

G.2 Reflector Prompt

The reflector receives the current program, its evaluation score, underperforming cases, and optional context (reference programs, lineage history), and outputs a V4A patch to improve the program. The complete prompt template is shown below. Sections in `{braces}` are populated dynamically at each iteration; optional sections (lineage log, write examples, success cases, reference programs) are included only when available for that iteration.

Interface specification. The following specification is provided to the reflector to define the `KnowledgeBase` API:

```
You are designing a Knowledge Base Program that implements three classes:

1. **KnowledgeItem** (dataclass): Defines what information is captured
   as knowledge items when writing to the knowledge base.
   - Must be a @dataclass with typed fields
   - An external LLM will populate instances by generating JSON matching
     your field definitions
   - **Field types MUST be JSON-compatible**: use only str, int, float,
     bool, list[str], Optional[str]
   - Do NOT use datetime, tuple, bytes, or custom objects
   - Use `field(metadata={"description": "..."})` to describe fields

2. **Query** (dataclass): Defines what parameters are used when reading
   from the knowledge base.
   - Same constraints as KnowledgeItem

3. **KnowledgeBase** (class): The core knowledge base system.
   - `__init__(self, toolkit)`: Receives a Toolkit with:
     - `toolkit.db`: sqlite3.Connection (in-memory SQLite)
     - `toolkit.chroma`: chromadb ephemeral client
     - `toolkit.llm_completion(messages, **kwargs) -> str`: LLM for
       reasoning, summarization, and information extraction
       (1 call per write/read invocation)
     - `toolkit.logger.debug(message)`: Debug logging
   - `write(self, item: KnowledgeItem, raw_text: str) -> None`
   - `read(self, query: Query) -> str`

Allowed imports: json, re, math, hashlib, collections, dataclasses,
typing, datetime, textwrap, sqlite3, chromadb

## Runtime Constraints
- `read()` output limit: at most 3000 characters.
- `write()` / `read()` timeout: 60 seconds each.
- `toolkit.llm_completion()` budget: at most 1 LLM call per
  `write()` or `read()` invocation. The budget resets before each call.

## Instruction Constants (required)
Four module-level string constants:
- INSTRUCTION_KNOWLEDGE_ITEM: What to extract and how to structure it.
- INSTRUCTION_QUERY: How to formulate retrieval queries.
- INSTRUCTION_RESPONSE: Answer format, length, and style.
- ALWAYS_ON_KNOWLEDGE: Persistent context injected into every task
  agent prompt. Can be empty.
```

Patch format specification. The reflector is instructed to output changes in V4A patch format:

```
Before the patch, output a commit message summarizing your changes:
```

```
*** Commit Message
Title: <one-line summary of what you changed and why>
- <root cause / diagnosis>
- <what you changed>

Then output your changes as a V4A patch.

IMPORTANT: You MUST output the exact markers '*** Begin Patch' and
'*** End Patch' on their own lines. Do NOT wrap them in code fences.

Format:
*** Begin Patch
*** Update File: program.py
@@ <optional context hint>
 context line (1-2 lines before change)
-removed line
+added line
 context line (1-2 lines after change)
*** End Patch

Rules:
- Lines prefixed with '-' are removed, '+' are added,
  ' ' (space) are unchanged context.
- Include 1-2 context lines before and after each change.
- Multiple hunks are allowed within one '*** Update File' block.
```

Main reflection prompt. The following is the complete prompt template sent to the reflector LLM for each mutation step in evolution.

```
You are an expert Python programmer specializing in knowledge base
system design.

Your task: Given a Knowledge Base Program, its evaluation score, and
underperforming cases, identify the root cause of each low score and
improve the program. Improvements are two-fold:
(A) **Prompt Optimization** -- tune the four instruction constants
    (especially ALWAYS_ON_KNOWLEDGE) to steer the task agent's
    behavior, and
(B) **Memory Design** -- improve the KnowledgeItem/Query schemas and
    KnowledgeBase storage/retrieval logic.
Both dimensions matter and should be considered together.

<interface_spec>
{KB_INTERFACE_SPEC}
</interface_spec>

<rules>
1. Output your diagnosis first, then your changes as a patch.
2. The code must define exactly three classes (KnowledgeItem, Query,
   KnowledgeBase) and four module-level string constants
   (INSTRUCTION_KNOWLEDGE_ITEM, INSTRUCTION_QUERY,
   INSTRUCTION_RESPONSE, ALWAYS_ON_KNOWLEDGE).
3. KnowledgeBase.__init__ must accept 'toolkit'; write takes a
   KnowledgeItem and a raw_text string; read takes a Query and
   returns str.
4. 'read()' must return at most 3000 characters.
5. Keep it simple. Make minimal changes that generalize beyond the
   specific cases shown -- no hardcoded word lists or
   case-specific pattern rules.
6. **Prompt Optimization**: Update INSTRUCTION_* to steer the task
   LLM's output format. Update ALWAYS_ON_KNOWLEDGE with domain
   strategies, heuristics, and behavioral rules the task agent
   should always follow -- this constant is injected into EVERY
   task agent action/decision prompt and is often the
   highest-leverage change. Study the <model_generation> transcripts
   in the underperforming cases to identify agent behavioral
   patterns (e.g., looping, inefficient exploration, wrong object
   selection) that ALWAYS_ON_KNOWLEDGE can fix.
7. **Memory Design**: Improve KnowledgeItem/Query field schemas and
   KnowledgeBase read()/write() logic to store and retrieve more
   useful information for the task agent.
8. Add clear comments explaining WHY each part of the code works
   the way it does -- this helps future iterations understand and
   preserve your design decisions.
</rules>
```

```
<patch_format>
{PATCH_FORMAT_SPEC}
</patch_format>

<current_program iteration="{iteration}">
```python
{code}
```
</current_program>

<evaluation_score>{score}</evaluation_score>

{lineage_section}
{train_section}
{memory_debug_logs}
{success_section}
{reference_section}

The following cases show poor performance on the validation set after
memory has been written. Each case contains the full
retrieval-and-answer conversation trajectory.

<underperforming_cases>
<case id="1">
<question>{question}</question>
<rationale>{expected_answer}</rationale>
<model_generation>{model_output}</model_generation>
<score>{case_score}</score>
<conversation>
  [user]: {query_generation_prompt}
  [assistant]: {query_json}
  [user]: <retrieved_memory>...</retrieved_memory> {instruction}
  [assistant]: {answer}
</conversation>
</case>
...
</underperforming_cases>

<task>
1. Diagnose why these cases scored low -- examine both the retrieval
   conversation AND the <model_generation> transcript for agent
   behavioral issues.
2. Propose improvements along two dimensions:
   (A) **Prompt Optimization**: How should INSTRUCTION_* and
       ALWAYS_ON_KNOWLEDGE change to steer the task agent better?
   (B) **Memory Design**: How should the schemas or
       storage/retrieval logic change to provide more useful
       information?
3. Output your changes as a patch.
</task>
```

Optional sections. The following sections are conditionally included when their data is available:

- **Lineage log**: Evolution history of the current program's lineage (ancestors, children, regression markers), formatted as commit-style entries with delta scores. Regression markers (← `REGRESSION`) flag changes that hurt performance, instructing the reflector not to repeat them.
- **Write examples**: Sample knowledge ingestion trajectories showing how the external LLM generates knowledge items from raw document text and how `write()` is called.
- **Success cases**: Cases where the current program performed well, instructing the reflector to preserve the behavior that makes these work.
- **Reference programs**: Higher- or lower-scoring programs from the population, with instructions to study which design patterns (e.g., use of `llm_completion`, ChromaDB vs. SQLite, schema granularity) correlate with scores within the pool.
- **Memory debug logs**: Outputs of `toolkit.logger.debug()` calls within `write()` and `read()`, providing visibility into program execution.

Underperforming case selection. At each iteration, 2 failed cases are sampled from the rotating validation set using the Efraimidis–Spirakis weighted sampling algorithm with weight $w_i = 1 - \text{score}_i$, biasing selection toward lower-scoring cases while maintaining diversity [9].

G.3 Compile-Fix Prompt

When a mutated program fails to compile or violates runtime constraints, a compile-fix prompt is sent to the reflector with the failing program and error details:

```
You are an expert Python programmer. A Knowledge Base Program failed
to compile or run. Fix the error and output your fix as a patch.

{KB_INTERFACE_SPEC}

## Failing Code
```python
{code}
```

Error
{error_type}: {error_details}

Fix the error and output your fix as a patch.
```

Up to k compile-fix iterations are attempted before the mutation is discarded.

H Evaluation Protocols

Each benchmark uses a task-specific evaluation metric. All metrics return a score in $[0, 1]$; the fitness function $J(P)$ averages these scores across the evaluation set.

H.1 Token F1 (LoCoMo)

Following Rajpurkar et al. [23], we compute token-level F1 between the model output and the expected answer. Both strings are lowercased, articles ("a", "an", "the") are removed, and all punctuation is stripped. Precision and recall are computed over the resulting token multisets using the standard formula below:

$$F_1 = \frac{2 \cdot |\text{out} \cap \text{exp}|}{|\text{out}| + |\text{exp}|}. \tag{5}$$

If both token sets are empty, the score is 1.0; if exactly one is empty, the score is 0.0.

LLM-Judge Score (LoCoMo). The L-J column in Table 1 reports a complementary LLM-as-judge metric that allows for paraphrastic answers that token F1 would penalize. A separate judge call grades each prediction against the gold answer and returns a binary correctness label, which we then average over the test set. The judge uses the same task agent model (GPT-5.4-mini) and the prompt template below; the score is 1.0 if the response parses as the integer 1 and 0.0 otherwise.

```
You are a strict judge. Determine if the output answers the question
correctly based on the expected answer. Reply ONLY with 1 (correct)
or 0 (incorrect).

Expected answer: <<expected>>
Actual output:   <<output>>

Score (0 or 1):
```

H.2 Binary Success (ALFWorld)

The agent interacts with the TextWorld environment for up to 50 steps [8, 26]. A score of 1.0 is assigned if the task is completed within the step budget; 0.0 otherwise.

H.3 Rubric-Based Scoring (HealthBench, PRBench)

Each evaluation item includes a structured rubric with multiple criteria, each assigned a point value (positive for desirable traits, negative for undesirable ones). An LLM judge independently grades

each criterion by examining the conversation and the last assistant response, returning a JSON object with an `explanation` and a boolean `criteria_met` field. For negative criteria (e.g., "Is overly verbose"), a good response should be graded as `false` (the undesirable trait is not present). The score is computed as:

$$\text{score} = \text{clip}\left(\frac{\sum_{i:\,\text{met}_i} p_i}{\sum_{i:\,p_i>0} p_i},\ 0,\ 1\right), \tag{6}$$

where p_i is the point value for criterion i and met_i indicates whether the criterion was judged as met. Negative criteria that are met subtract from the numerator. This protocol follows the official HealthBench and PRBench evaluation methodology [3, 2]. The LLM judge uses the following prompt template for each criterion during grading:

```
Your job is to look at a conversation and a rubric item, and score
the last turn (the last assistant response) on how well it follows
the rubric item.

# Conversation
<<conversation>>

# Rubric item
<<rubric_item>>

# Instructions
Return a json object with fields: "explanation" and "criteria_met".
- "explanation": why the response does or does not meet the criteria.
- "criteria_met": boolean. If a rubric item has multiple criteria,
  all must be met to return true.
- Exception: if criteria says "such as" or "for example", the
  response need not include all listed examples.
- For negative criteria (undesirable traits), return whether the
  criteria IS met (true = bad behavior present), not whether the
  response is good.
```

I Extended Results

I.1 Complete Per-Category Results

Table 4 in the main paper reports per-category results for ALFWorld Unseen and LoCoMo with selected baselines. Here we provide the complete per-category breakdown for all benchmarks and all baselines in the appendix tables below.

Table 10: **Complete per-category results for ALFWorld Unseen.** Columns report success rate by task type. Best per column appears in **bold** for each reported task category.

	Simple (n=8)	**Movable** (n=1)	**Clean** (n=11)	**Light** (n=1)	**Cool** (n=7)	**Heat** (n=6)	**2-Obj** (n=8)
No Memory	0.88	**1.00**	0.82	0.00	0.57	0.83	0.62
Vector Search	0.62	**1.00**	0.64	0.00	0.71	0.67	0.62
G-Memory	**1.00**	**1.00**	0.55	**1.00**	0.29	0.83	0.75
Mem0	0.88	**1.00**	0.64	0.00	0.57	**1.00**	0.75
Trajectory Retrieval	0.75	**1.00**	0.73	0.00	0.71	0.67	0.75
ReasoningBank	**1.00**	**1.00**	0.55	**1.00**	0.57	0.83	0.75
Dynamic Cheatsheet	**1.00**	**1.00**	0.36	**1.00**	0.00	**1.00**	0.75
GEPA	0.88	**1.00**	**1.00**	0.00	**0.86**	**1.00**	0.62
GEPA + Vector Search	0.88	**1.00**	**1.00**	0.00	0.57	**1.00**	0.62
MSTAR	0.88	**1.00**	0.91	**1.00**	**1.00**	0.83	0.75

HealthBench and PRBench each evaluate on single-category splits (emergency referrals, health data interpretation, legal reasoning, and financial reasoning). Their per-category scores equal the overall scores reported in Table 1; we omit separate per-category tables for these benchmarks.

Table 11: **Complete per-category results for ALFWorld Seen.** Columns report success rate by task type. Best per column appears in **bold** for each reported task category.

	Simple (n=2)	**Clean** (n=12)	**Light** (n=2)	**Cool** (n=8)	**Heat** (n=7)	**2-Obj** (n=15)	**Overall** (n=50)
No Memory	**1.00**	0.33	**1.00**	0.50	0.43	0.87	0.640
Vector Search	**1.00**	0.42	**1.00**	**1.00**	0.29	0.87	0.720
G-Memory	**1.00**	0.33	**1.00**	0.12	0.29	0.87	0.560
Mem0	**1.00**	0.25	**1.00**	0.62	0.43	0.87	0.640
Trajectory Retrieval	**1.00**	0.58	**1.00**	0.88	0.43	**0.93**	0.780
ReasoningBank	**1.00**	0.33	**1.00**	0.25	0.43	0.80	0.580
Dynamic Cheatsheet	**1.00**	0.17	**1.00**	0.38	0.00	0.73	0.480
GEPA	**1.00**	0.50	**1.00**	0.88	0.43	0.80	0.720
GEPA + Vector Search	**1.00**	**0.92**	**1.00**	**1.00**	0.29	0.80	**0.820**
MSTAR	**1.00**	0.42	**1.00**	0.75	**0.57**	0.80	0.700

Bold marks the best per category; GEPA+VS has the highest overall score (0.820).

Table 12: **Complete per-category results for LoCoMo.** Columns report token F1 by question category: single-hop recall, temporal reasoning, causal reasoning, and open-domain. Best per column appears in **bold** for each reported question category.

	Single-hop (n=18)	**Temporal** (n=18)	**Causal** (n=9)	**Open-domain** (n=55)	**Overall**
No Memory	0.02	0.00	0.11	0.04	0.036
Vector Search	0.19	0.29	0.38	0.25	0.256
G-Memory	0.23	0.12	0.31	0.24	0.224
Mem0	**0.33**	0.26	0.37	0.43	0.373
Trajectory Retrieval	0.18	**0.33**	0.34	0.28	0.276
ReasoningBank	0.25	0.07	0.34	0.19	0.194
Dynamic Cheatsheet	0.14	0.00	0.22	0.14	0.124
GEPA	0.09	0.06	0.24	0.15	0.132
GEPA + Vector Search	0.16	**0.38**	**0.39**	0.31	0.300
MSTAR	**0.33**	0.19	0.35	**0.51**	**0.459**

I.2 Multi-Seed Stability Results

Table 3 in the main paper reports summary statistics (mean, standard deviation, coefficient of variation) across five evolution seeds. Here we provide the full per-seed results.

Table 13: **Per-seed test scores across stability experiments.** Best Baseline denotes the strongest fixed baseline.

	Test Score by Seed								
Benchmark	**0**	**1**	**2**	**3**	**4**	**Mean**	**±Std**	**CV**	**Best Baseline**
LoCoMo	0.459	0.463	0.364	0.394	0.423	0.421	0.042	10.0%	0.373
HB-Data	0.390	0.382	0.357	0.411	0.413	0.391	0.023	5.9%	0.327
PR-Finance	0.586	0.544	0.582	0.582	0.513	0.561	0.032	5.7%	0.449

On LoCoMo, 4 out of 5 seeds exceed the strongest baseline (Mem0, 0.373). The single exception (seed 2, 0.364) falls within 2.4% of the baseline, consistent with the higher variance inherent to LoCoMo's multi-category evaluation. On HealthBench Data and PRBench Finance, all 5 seeds exceed the respective strongest baselines (GEPA+VS at 0.327; GEPA at 0.449) by substantial margins, demonstrating consistent improvement regardless of initialization.

Table 14 reports the best validation score achieved during evolution for each seed, which determines the program selected for final test evaluation.

Table 14: **Best validation score during evolution by seed.** Scores are computed on the static validation subset used to select the program for test evaluation.

	Best Validation Score						
Benchmark	**0**	**1**	**2**	**3**	**4**	**Mean**	**±Std**
LoCoMo	0.333	0.273	0.289	0.312	0.333	0.308	0.027
HB-Data	0.424	0.381	0.378	0.397	0.380	0.392	0.020
PR-Finance	0.521	0.522	0.559	0.554	0.515	0.534	0.020

J Additional Analyses

This section reports analyses that complement the main results. Unless stated otherwise, all runs use the evolved programs and splits of Table 1.

J.1 Significance of the Main Results

To account for the small test sets, we compute 95% percentile bootstrap intervals with 10,000 resamples over the test items. We then pair MSTAR item by item with a strong baseline on each configuration and report the difference and its one-sided p-value (Table 15). Among the six configurations where MSTAR leads, the lead is significant on five. On ALFWorld-Seen, the interval of MSTAR contains the score of the best baseline, GEPA + Vector Search (0.820).

Table 15: **Bootstrap intervals and paired tests on the test sets.**

Benchmark	**MSTAR**	**95% CI**	**Δ vs. baseline [95% CI]**	p
LoCoMo	0.459	[0.386, 0.535]	+0.086 vs. Mem0 [+0.044, +0.130]	0.001
ALFWorld-Unseen	0.881	[0.786, 0.976]	+0.143 vs. No Memory [+0.048, +0.262]	0.002
ALFWorld-Seen	0.700	[0.580, 0.820]	−0.080 vs. Traj. Retrieval [−0.220, +0.060]	0.17
HB-Data	0.390	[0.336, 0.445]	+0.063 vs. GEPA + VS [+0.018, +0.114]	0.003
HB-Emergency	0.493	[0.429, 0.555]	+0.006 vs. Dyn. Cheatsheet [−0.062, +0.069]	0.41
PR-Legal	0.660	[0.599, 0.721]	+0.092 vs. GEPA [+0.051, +0.137]	<0.0001
PR-Finance	0.586	[0.534, 0.638]	+0.137 vs. GEPA [+0.096, +0.179]	<0.0001

J.2 Search versus the Coding Model's Prior

To separate the effect of search from the prior knowledge of the coding model, we ask GPT-5.3-Codex to write a LoCoMo memory program in a single shot from the task description and a few examples, without any evolution. Table 16 shows that the best of five single-shot programs exceeds the Vector Search seed but remains far below the evolved program. Therefore, most of the gain comes from search.

Table 16: **Single-shot programs written by the coding model on LoCoMo (token F1).**

Configuration	**Search budget**	**F1**
Single-shot Codex, mean ± std over 5 samples	0 iterations	0.227 ± 0.061
Single-shot Codex, best of 5 (selected on validation)	0 iterations	0.297
Vector Search (seed)	n/a	0.256
MSTAR	20 iterations	0.459

J.3 Search Efficiency

All evolutionary baselines use the same task agent, coding model, splits, and number of rounds as MSTAR. Table 17 compares the cost at which MSTAR first exceeds the final validation score of GEPA. On five of seven configurations, MSTAR reaches this score with 16–31% of the GEPA budget.

Table 17: **Validation cost for MSTAR to exceed the final validation score of GEPA.**

Benchmark	GEPA final val.	GEPA cost	MSTAR first exceeds	Cost (share)
LoCoMo	0.231	≥$4.57	iteration 1	$1.00 (22%)
ALFWorld-Unseen	0.980	≥$8.08	iteration 2 (tie)	$2.04 (25%)
ALFWorld-Seen	0.875	≥$3.96	never	n/a
HB-Data	0.373	≥$15.65	iteration 2	$4.80 (31%)
HB-Emergency	0.438	≥$12.06	iteration 1	$2.89 (24%)
PR-Legal	0.515	≥$29.74	iteration 1	$4.75 (16%)
PR-Finance	0.409	≥$26.67	iteration 1	$5.42 (20%)

J.4 Transfer across Task Models

We evaluate the frozen evolved programs and the fixed baselines under two other task models, the weaker GPT-5-mini from the same family and DeepSeek-V3.2 from a different family. As Table 18 shows, MSTAR ranks first under every task model.

Table 18: **Frozen programs under different task models.** The strongest baseline is Mem0 on LoCoMo and ReasoningBank on PRBench-Legal.

Benchmark	Task model	MSTAR	Vector Search	Strongest baseline	No Memory
LoCoMo	GPT-5.4 Mini	**0.459**	0.256	0.373	0.036
LoCoMo	GPT-5-mini	**0.430**	0.253	0.344	0.016
LoCoMo	DeepSeek-V3.2	**0.318**	0.259	0.264	0.115
PR-Legal	GPT-5.4 Mini	**0.660**	0.466	0.508	0.431
PR-Legal	GPT-5-mini	**0.833**	0.597	0.650	0.607
PR-Legal	DeepSeek-V3.2	**0.631**	0.348	0.389	0.332

J.5 Independent Judge

HealthBench and PRBench are scored by an LLM judge from the same model family as the task agent. We therefore re-score the frozen answers with DeepSeek-V4-Flash using the same per-criterion rubric prompts. Table 19 shows that the lead of MSTAR over the strongest baseline holds under this judge, although its size changes.

Table 19: **Rubric scores re-judged by DeepSeek-V4-Flash.**

Benchmark	Strongest baseline	MSTAR	Baseline
HB-Data	GEPA + Vector Search	**0.403**	0.321
HB-Emergency	Dynamic Cheatsheet	**0.512**	0.418
PR-Legal	GEPA	**0.612**	0.569
PR-Finance	GEPA	**0.656**	0.558

J.6 Transfer to LongMemEval

To test whether the LoCoMo program captures task-level structure rather than benchmark artifacts, we evaluate it on 20 LongMemEval$_S$ questions sampled with a fixed seed [32]. The program evolved on LoCoMo reaches a token F1 of 0.372, compared with 0.278 for the Vector Search seed.

J.7 Audit of Hard-Coded Literals and Evaluation Loopholes

We statically scan all 157 programs produced by the seven evolution runs for imports outside the whitelist, access to ground truth or evaluator internals, stack-frame introspection, and dynamic execution or file and network I/O, and we find no such case. We also count string literals in the best program of each benchmark that also appear in the dataset text (Table 20). Such literals appear in all seven search trajectories as early as iterations 1–6, and they are most frequent on PRBench-Legal,

where legal terms form the vocabulary of the task. However, they do not always win, since on LoCoMo one candidate contained 81 such literals while the final program contains none.

Table 20: **Dataset-derived string literals in the best evolved program (after deduplication).**

Benchmark	Literals	Per 100 lines
PR-Legal	89	8.2
ALFWorld-Seen	23	6.8
PR-Finance	25	5.3
HB-Data	14	2.3
HB-Emergency	2	1.0
LoCoMo	0	0.0
ALFWorld-Unseen	0	0.0

J.8 Overlap between Splits

For all HealthBench and PRBench configurations, we compute content-word Jaccard similarity across all pairs of episode pool, validation set, and test set items. The best-match Jaccard is at most 0.37, with medians between 0.06 and 0.12, and no rubric criterion appears verbatim in any episode.

K Broader Impact

This work studies agent memory as a research object in controlled benchmark settings. Its main positive impact is to make memory programs more reproducible and inspectable: the discovered program is executable code that can be read, tested, and released with the artifacts. This can help researchers compare memory programs beyond fixed retrieval pipelines. The main risk is that better memory can also make agents more persistent in carrying forward wrong, private, or biased information when the data or objective contains it. For uses outside controlled benchmarks, we believe deployment should include data minimization, task-specific evaluation, logging, and human review, especially in high-stakes domains such as health and law.

L LLM Usage

The authors used large language model agents as research assistants during the preparation of this work. Their assistance covered early literature exploration, identification of related papers, implementation support, experiment execution and debugging, and language polishing during writing. The authors made the scientific decisions, developed the ideas and claims, designed the method and experiments, and determined the paper structure and final content.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction state the method and scope; results in Section 5 support the empirical claims.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: Section 8 discusses metric gaming, hard-coded literals, configurations where the method does not lead, the single model pair, and cost.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A].

Justification: The paper gives a problem formulation but does not claim theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes].

   Justification: Sections 3–4 and Appendices A, B, D.2, and H specify the procedure.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes].

   Justification: The code, experiment scripts, and generated artifacts are released at `https://github.com/wbopan/mstar`.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

   Justification: Section 4 and Appendices B and D.2 give splits, models, and hyperparameters.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes].

   Justification: Section 5 and Appendix I.2 report seed variance, and Appendix J.1 reports bootstrap intervals and paired tests.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.

- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

   Justification: Appendix D.1 reports model calls, tokens, cost, and wall-clock time.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes].

   Justification: The work uses existing benchmarks and simulated environments, and we identify no exception to the Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes].

    Justification: Appendix K discusses both positive impacts and risks.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A].

    Justification: The paper does not release a high-risk model or scraped dataset.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes].

    Justification: Existing datasets and baselines are cited, and the supplementary release documents asset sources, licenses, and terms of use.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes].

    Justification: The supplementary material documents the released code, evolved programs, scripts, and artifacts.

    Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A].

    Justification: The work does not involve new crowdsourcing or human-subject experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A].

    Justification: No new human-subject study is conducted.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes].

    Justification: Appendix L describes the use of LLM agents for literature exploration, related-paper identification, implementation support, experiment execution and debugging, and writing polish. The ideas, claims, paper structure, and final content are the authors' original work.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.

- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.