MEMSCOPE: EVOLVING AGENT MEMORY ARCHITECTURES THROUGH LOCALIZED CODE REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent memory architectures define how LLM agents store and reuse experience across tasks, but they are typically fixed after deployment, limiting their adaptability across task types. Recent self-evolving approaches instead adapt these memory architectures using execution feedback, typically represented as evaluation scores or compressed summaries. However, such coarse-grained feedback is insufficient for precise failure localization in current agent systems, where long-horizon task execution involves multiple interacting components. Without a precise failure location, existing approaches lack a clear code-level target for modification, leaving the update scope weakly constrained and often leading to large-scale rewrites of an otherwise capable memory architecture. We propose MemScope, a framework for evidence-grounded, localized evolution of an existing memory architecture. Its tool-using Patch Agent jointly inspects raw execution traces, runtime memory state, and current architecture code to identify repair targets and apply bounded, in-place edits under provider-identity and memory-compatibility constraints. On GAIA, the trace-and-memory inspection variant achieves 86.67% accuracy, compared with 70.00% for trace-only and 63.33% for summary-only feedback. Under xBench tasks, the evolved memory architecture improves over its original version from 49 to 55 correct answers with DeepSeek V3.2 and from 50 to 53 with Kimi K2.7 Code. On disjoint ALFWorld unseen tasks, success rises from 50.75% to 71.64%. Across archived architecture-snapshot transitions, the median code-change ratio is 4.4%. These results support fine-grained execution evidence as a basis for localized, in-place memory-architecture evolution.

1 INTRODUCTION

An LLM agent combines a language model with an agent harness that manages tool use, context construction, execution control, and memory (Weng, 2026; Lee et al., 2026; Zhang et al., 2026b; Lin et al., 2026b). Within this harness, the memory architecture implements reusable decisions about what experience to store, how to retrieve it, and how to present it to the agent (Zhong et al., 2024; Chhikara et al., 2025; Wang et al., 2025; Du et al., 2025). Most existing architectures remain fixed after deployment even as agents encounter open-ended tasks with different memory demands. This motivates adapting not only the contents of memory, but also the executable mechanisms that manage them.

Recent approaches use execution outcomes to evolve agent memory, but differ in what they allow to change. Some update the experience retained by the agent while leaving the executable logic for storage, retrieval, and presentation fixed (Shinn et al., 2023; Liu et al., 2025; Fang et al., 2026; Ouyang et al., 2026b; Zhang et al., 2026c;d). Such content-level updates cannot directly correct failures caused by the logic itself. Architecture-level methods instead make the executable memory architecture mutable, allowing evolution to modify the code that determines what is stored, how it is retrieved, and how retrieved information is presented (Zhang et al., 2026a; Xiong et al., 2026a). This shifts the evolution target from memory contents to the mechanisms that operate on them.

We focus on a complementary problem: identifying where an existing memory architecture should change. Evaluation scores and compressed summaries do not directly connect an observed failure

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

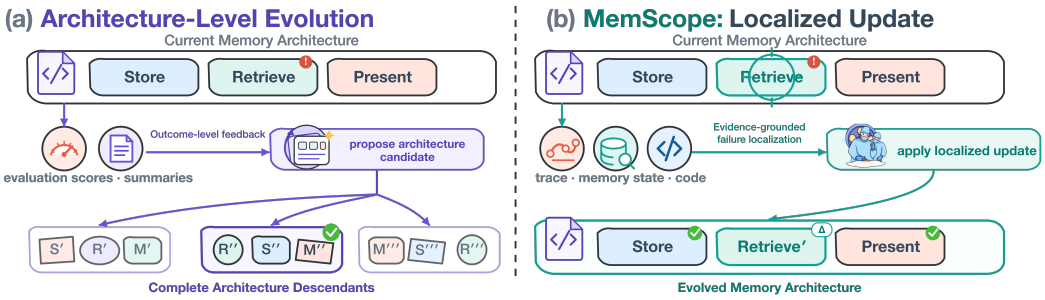

Figure 1: **Feedback and update granularity in memory-architecture evolution.** Prior architecture-level approaches use outcome-level feedback to propose complete executable architectures. MemScope makes the repair target explicit by jointly inspecting execution traces, runtime memory state, and provider code, then applies a localized, in-place edit to the existing provider. In the illustrated update, only the highlighted code region is modified.

to a specific implementation region. In long-horizon execution, this connection is difficult because behavior depends on interactions between stored experience, retrieval decisions, and downstream agent actions. Figure 1 contrasts complete-architecture proposals driven by outcome-level feedback (Xiong et al., 2026a; Zhang et al., 2026a) with evidence-grounded identification of a local repair target.

Consider an agent that fails to receive useful memory guidance. The relevant experience may be absent from memory, excluded by retrieval filtering, or bypassed by an early-return code path. These explanations suggest different interventions; a failure may also arise after suitable guidance has already been delivered and require no provider change. Inspecting raw traces, runtime memory state, and current code can help distinguish these cases. We therefore investigate whether fine-grained execution evidence can support targeted, in-place updates to an existing memory architecture.

To study this question, we introduce MemScope, a framework for evidence-grounded, localized evolution of agent memory architectures. An Agent Executor performs tasks with the active architecture, while a tool-using Patch Agent inspects execution evidence between task segments. Each architecture is packaged as a reusable memory skill with a natural-language `SKILL.md` file and an executable provider, separate from its runtime memory bank (Anthropic, 2025). When repair is triggered, the Patch Agent examines successful and failed raw traces, stored memories and retrieval decisions, and current provider code to identify a candidate repair region. It then proposes exact old-to-new replacements under a local-update contract that preserves provider identity and requires compatibility with existing memory records. A patch that passes local checks is installed for subsequent execution, retaining the runtime memory bank; insufficient evidence can instead lead to a no-patch decision.

Under DeepSeek V3.2, MemScope + Flash-Searcher reaches 83.64% accuracy on GAIA validation and 78% on complete xBench, compared with 67.88% and 70% for MemEvolve + Flash-Searcher; MemScope's complete xBench score with Kimi K2.7 Code is 74%, and a separate matched-start comparison favors the evolved DynamicCheatsheet over the original with both DeepSeek (55/70 versus 49/70) and Kimi (53/70 versus 50/70). On ALFWorld, unseen-task success after the same online-memory collection is 71.64% for MemScope versus 50.75% for the original DynamicCheatsheet, while the GAIA diagnostic ablation yields 86.67% for trace-and-memory inspection versus 63.33% for summary-only feedback. Across 14 archived provider transitions, the median code-change ratio is 4.4%.

Our contributions are:

- We introduce an evidence-grounded repair interface that jointly exposes raw execution traces, runtime memory state, and provider code, making failure localization an explicit step in the evolution of an existing memory architecture.
- We develop MemScope to turn the resulting diagnosis into bounded, in-place code edits under provider-identity and memory-compatibility constraints. Across 14 archived provider-snapshot transitions, the median code-change ratio is 4.4%.

- In the GAIA diagnostic ablation, trace-and-memory inspection yields 86.67% accuracy versus 63.33% for summary-only feedback. Matched-start xBench comparisons favor the evolved provider under both backbones, and ALFWorld unseen-task success improves from 50.75% to 71.64% after a common memory-collection sequence.

2 Related Work

2.1 Agent Memory

Agent memory supports personalization and experience reuse (Du et al., 2025; Zhang et al., 2025c; Packer et al., 2023; Rasmussen et al., 2025). Its development spans trace storage, reflection, and experience abstraction (Luo et al., 2026), with memory behavior evaluated within and across episodes by EvoMemBench (Wang et al., 2026b). MemoryBank (Zhong et al., 2024), Mem0 (Chhikara et al., 2025), and temporal semantic memory (Su et al., 2026) retain preferences, temporal states, and evolving personas for long-term coherence. Experience-based systems instead reuse execution traces: Agent Workflow Memory induces web-navigation workflows (Wang et al., 2025), while Memp distills procedural memories (Fang et al., 2026; Hu et al., 2025a). Most retain a fixed memory architecture after deployment.

Yet retrieved traces can propagate errors or encourage misaligned replay (Xiong et al., 2026b; Chen et al., 2025). MemScope adapts the architecture itself: provider code defines task-family storage and retrieval mechanisms, while a skill document specifies their use for the current task stream.

2.2 Self-Evolving Agents

Self-evolving agents use execution feedback to improve persistent components, including solution-generating programs, agent implementations, and execution harnesses (Zelikman et al., 2024; Zhang et al., 2025b; Lee et al., 2026; Zhang et al., 2026b; Lin et al., 2026b). Many memory-related approaches accumulate or refine non-parametric experience and natural-language memory skills (Shinn et al., 2023; Liu et al., 2025; Fang et al., 2026; Zhang et al., 2026c; Ouyang et al., 2026b; Zhang et al., 2026d; 2025a; Lin et al., 2026a; Ouyang et al., 2026a; Yang et al., 2026; Li & Hu, 2026), changing retained content or memory-construction instructions while keeping the executable architecture fixed. Evo-Memory studies memory revision over sequential task streams (Wei et al., 2025), while FORGE turns failure traces into textual memories shared among agents (Bogdanov et al., 2026). TMEM combines explicit memory with online parametric adaptation (Ren et al., 2026); AutoAgent evolves cognition and orchestrates elastic memory (Wang et al., 2026a).

ALMA searches complete memory architectures expressed as code, and MemEvolve jointly evolves experiential knowledge and modular memory architectures (Xiong et al., 2026a; Zhang et al., 2026a). MemScope's local-evolution path takes an existing provider as its starting point and uses trace- and memory-grounded replacements, making the repair target and edit scope explicit.

LLM-based automated program repair studies how models diagnose software failures and generate code changes (Yang et al., 2024), with repository-level repair evaluated on benchmarks such as SWE-bench (Jimenez et al., 2024). MemScope applies localized repair to the agent's own memory architecture, grounding updates in task execution traces, runtime memory, and provider code.

3 Method

Figure 2 shows the evolution target and loop. For an existing skill, MemScope uses task evidence to repair the active provider in place when the trigger fires. Provider code changes locally while the separate runtime memory continues to update during task use. Our provider-evolution runs start from the existing DynamicCheatsheet memory architecture (Suzgun et al., 2026) and distinguish diagnostic-evidence variants from original-versus-evolved provider comparisons.

3.1 Evolution Target

Each memory skill s_t combines natural-language guidance in `SKILL.md` with an executable provider p_t implementing `memory_retrieval` and `memory_storage`, following the Agent

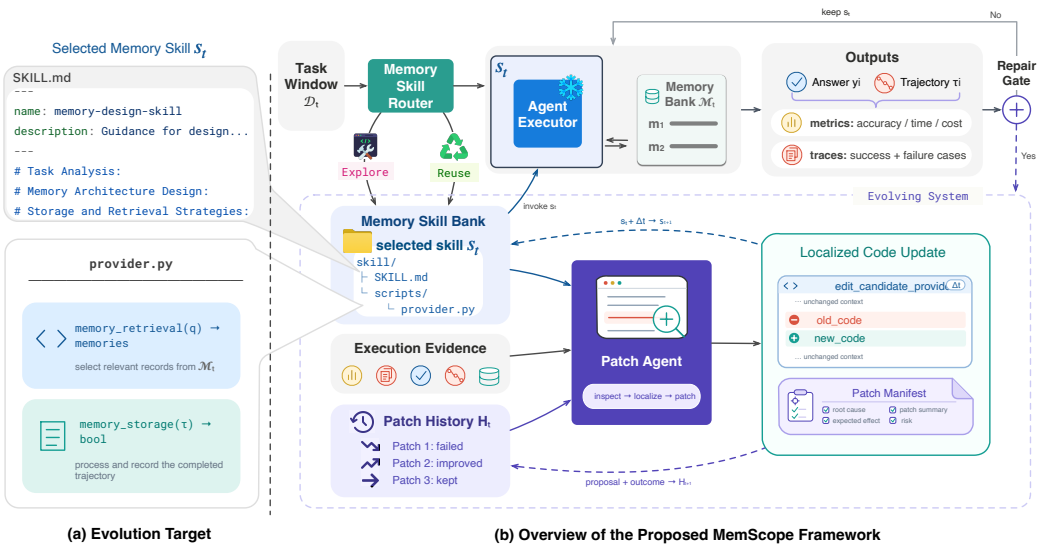

Figure 2: Overview of MemScope. (a) Evolution Target. Following the Agent Skills format, each reusable memory skill contains a required `SKILL.md` file and an executable provider under `scripts/`; the activated provider maintains a separate skill-specific runtime memory bank $\mathcal{M}_t$. (b) MemScope Evolution Loop. Execution produces metrics and raw success and failure traces. When repair is triggered, the Patch Agent inspects these traces, the active memory bank, the current provider, and Patch History. A local edit changes p_t into p_{t+1} without replacing the skill, while the patch and its later outcome are recorded for subsequent evolution.

Skills format (Anthropic, 2025). MemScope locally updates p_t and records the design lesson in `SKILL.md`; runtime memory remains separate from the reusable skill (Figure 2(a)). A recorded diagnosis and the Patch-Agent prompt appear in Appendices B and A.

3.2 MemScope Evolution Loop

As shown in Figure 2(b), an evolution episode is one intended pass over a task window $\mathcal{D}_t$. MemScope first obtains probe evidence, uses a fixed or routed skill, and partitions the remaining tasks into sequential segments executed with the active provider. After each segment, the repair trigger determines whether to invoke the Patch Agent.

Skill selection and task execution. MemScope initializes the Memory Skill Bank with 12 human-designed memory skills from the EvolveLab codebase of MemEvolve (Zhang et al., 2026a). At the start of each episode, MemScope runs probe tasks from $\mathcal{D}_t$. The Memory Skill Router uses the probe tasks and their execution evidence together with the available skill descriptions to select a suitable existing skill s_t and activate its provider p_t. When the Router determines that no existing skill is suitable, MemScope enters global exploration: the Skill Designer derives a new skill specification from the probe evidence, and MemScope generates a complete provider from this specification before registering and evaluating the new skill-provider pair.

After routing, the frozen Agent Executor $\mathcal{A}$ executes the remaining tasks using the selected provider p_t and a skill-specific runtime memory bank $\mathcal{M}_t$, which stores task experience separately from the reusable skill files. For each task x_i, the provider calls `memory_retrieval` to obtain relevant memories, and the Agent Executor produces an answer y_i and trajectory τ_i. MemScope retains τ_i as a raw execution trace for diagnosis, while the provider applies `memory_storage` to distill reusable information from τ_i into a memory item m_i and update the runtime memory bank:

$$r_i = \text{memory_retrieval}_{p_t}(x_i, \mathcal{M}_t), \tag{1}$$

$$(y_i, \tau_i) = \mathcal{A}(x_i, r_i), \tag{2}$$

$$m_i = \text{memory_storage}_{p_t}(\tau_i), \tag{3}$$

$$\mathcal{M}_t \leftarrow \mathcal{M}_t \oplus m_i. \tag{4}$$

Here, r_i denotes retrieved memories, m_i the distilled memory item, and $\oplus$ provider-specific insertion or update. The runtime bank is retained across a localized patch. Sequential execution segments yield accuracy, time, cost, and raw success/failure traces for the repair trigger and Patch Agent.

Repair trigger. MemScope tracks the best execution accuracy from previous episodes on the same task window. Let a_t be current segment accuracy, $a_t^\star$ this historical reference, and f_t the current number of failures. When performance falls below the reference by more than δ and at least $f_{\min}$ tasks fail, MemScope invokes the Patch Agent to inspect the execution evidence: $g_t = \mathbb{I}[a_t^\star - a_t > \delta]\,\mathbb{I}[f_t \geq f_{\min}]$. We use $\delta = 0.05$ and $f_{\min} = 1$. Patches that pass local checks update the provider for subsequent execution; failed checks and `no_patch` retain p_t. Table 7 illustrates a recorded `no_patch` decision.

3.3 Patch Agent: Fine-Grained Diagnosis and Localized Updates

The Patch Agent makes failure localization an explicit step in memory-architecture evolution, turning observed task failures into evidence-grounded, localized updates to the active provider. As shown in Figure 2(b), it takes execution evidence, the selected memory skill s_t, and Patch History $\mathcal{H}_t$ as inputs, and produces a localized code update Δ_t together with a patch manifest that summarizes the proposed root cause, patch, expected effect, and risk.

Diagnosis and localized update. The Patch Agent inspects execution traces, runtime memory, and provider code through the read-only tools in Table 1 to identify a repair target. A score alone cannot distinguish absent experience, filtering/ranking errors, and ineffective guidance formatting; the tools expose observed behavior, stored experience, retrieval decisions, and implementation for on-demand inspection. The GAIA ablation in Section 4.4 compares summary-only feedback with access to raw traces and memory inspection, using task accuracy as the outcome. The Patch Agent may use the tools repeatedly and in any order to select a proposed edit location. Successful traces provide contrastive evidence for selecting the repair target.

Having selected a repair target, the Patch Agent emits exact old-to-new replacements on an experiment-local provider and a manifest containing its hypothesized root cause, expected effect, and risk. Early GAIA artifacts check syntax, class identity, memory type, and required method names; later xBench/ALFWorld artifacts additionally record import and isolated-retrieval checks. Prompts require schema compatibility and prohibit broad rewrites and task/answer hardcoding. After local validation, the updated provider is used in subsequent task execution, and its observed outcomes are recorded in Patch History.

Patch history. Patch History $\mathcal{H}_t$ records each proposal, its installation decision, and later outcomes on revisited tasks, including recovered failures and regressed successes. The Patch Agent can consult these records when making subsequent repair decisions on the same provider and skill. The xBench and ALFWorld cases in Section 4.4 illustrate this use.

Table 1: Evidence sources for failure localization.

Tool or action	Evidence exposed
Trace Inspection	
`list_traces`; `read_trace_header`; `read_trace_steps`; `grep_traces`	Index available cases; inspect a task result and step map; read selected native steps; search matching events across traces.
Memory Inspection (`memory_store`)	
`stats`	Summarize record counts, fields, schemas, and outcome distribution.
`list`	List compact record summaries, optionally filtered by outcome.
`read`	Return one complete raw memory record.
`query`	Execute a recorded request and return raw matches, selection decisions, and the retrieval result.
Code Grounding	
`read_candidate_provider`; `read_skill_file`	Inspect the current executable code and the skill's intended scope and memory-architecture design.

4 Experiments

4.1 Experimental Setup

Benchmarks. Following prior evaluations of general-purpose and self-evolving agents (Laws, 2024; Qiu et al., 2025; Zhang et al., 2026a), we evaluate MemScope on three benchmarks. GAIA (Mialon et al., 2024) covers real-world tasks requiring multi-step reasoning, web browsing, multimodal understanding, and tool use; its validation set contains 165 tasks across three difficulty levels. xBench DeepSearch-2505 (xbench, 2025) evaluates web research and information synthesis. ALFWorld (Shridhar et al., 2021) evaluates sequential household interaction in a text-based environment. Together, they cover information-seeking and embodied task settings. Dataset-specific evaluation protocols appear below.

Evolution and evaluation. We use Flash-Searcher (Qin et al., 2025) as the Agent Executor for GAIA and xBench, with DeepSeek V3.2 (DeepSeek-AI, 2025) and Kimi K2.7 Code (Kimi Team, 2025) as backbone models. For GAIA tasks with visual inputs, Flash-Searcher uses GPT-4o (OpenAI, 2024a) (`gpt-4o-2024-11-20`) for visual inspection. ALFWorld uses DeepSeek V3.2 with a text-action executor. Providers use `all-MiniLM-L6-v2` (Reimers & Gurevych, 2019; Wang et al., 2020) for semantic retrieval. GAIA and xBench answers are evaluated by GPT-4o-mini (OpenAI, 2024b); ALFWorld uses environment-reported task success.

Across the three benchmarks, the intended evolution schedule comprises five episodes over 30 development tasks, with three probes and 27 execution tasks in segments of 10, 10, and 7. Following prior work (Zhang et al., 2026a; Xiong et al., 2026a), we use sequential online evaluation to measure the resulting memory systems during continued use. Following MemEvolve (Zhang et al., 2026a), we evaluate GAIA on the full validation set. For xBench and ALFWorld, code persists across evolution episodes while memory resets between episodes and accumulates within each; post-evolution comparisons use the last code state and continue updating memory. The separate xBench fixed-code comparison starts original and evolved providers from independent empty stores, as does ALFWorld before its common collection sequence. In these comparisons, the Agent Executor receives memory retrieved by the provider without injecting `SKILL.md` experience into its task context. Our xBench and ALFWorld outcomes count raw `score==1` records, with errors retained in the denominator.

4.2 Main Results on GAIA and xBench

Closed-source agent frameworks. On GAIA, MemScope + Flash-Searcher with DeepSeek V3.2 reaches 83.64% pass@1 accuracy, compared with 75.15% pass@1 for Alita, the highest result in the closed-source group (Table 2).

Table 2: GAIA validation and xBench DeepSearch-2505 accuracy (%).

System	Model Family	GAIA				xBench
		Avg.	L1	L2	L3	
Closed-Source Agent Frameworks						
Alita (pass@1)	Claude Sonnet 4, GPT-4o	75.15	77.36	76.74	65.38	—
Langfun Agent	Claude 3.7 etc.	71.52	83.02	68.60	57.69	—
TraseAgent	Claude etc.	70.30	83.02	69.77	46.15	—
OpenAI Deep Research	o1, o3 etc.	67.36	74.29	69.06	47.60	—
h2oGPTe	Claude 3.5	63.64	67.92	67.44	42.31	—
Desearch	GPT-4o	56.97	71.70	58.14	23.08	—
OMNE	o1-preview	46.06	60.38	44.19	23.08	—
Open-Source Agent Frameworks						
OWL Workforce (pass@3)	GPT-4o+o3-mini	60.61	81.14	58.14	26.92	55.00
OWL Roleplaying (pass@3)	GPT-4o+o3-mini	58.18	81.14	54.65	23.08	—
TapeAgents	Claude 3.7 etc.	55.76	71.70	53.49	30.77	—
AutoAgent	Claude 3.5 etc.	55.15	71.70	53.40	26.92	—
Magentic-One	OpenAI o1 etc.	46.06	56.60	46.51	23.08	—
Cognitive Kernel-Pro (pass@1)	Claude 3.7 etc.	60.00	79.25	56.98	30.77	56.00
Cognitive Kernel-Pro (pass@3)	Claude 3.7 etc.	75.15	84.91	73.26	61.54	—
OAgents	Claude 3.7 etc.	66.67	77.36	66.28	46.15	47.00
Agent KB (pass@1)	GPT-4.1	61.21	79.25	58.14	34.62	48.00
Agent KB (pass@2)	GPT-4.1	67.27	83.02	67.44	34.62	58.00
Agent KB (pass@3)	GPT-4.1	73.94	84.91	73.26	53.85	68.00
Flash-Searcher ⚡ (pass@1)	Kimi K2	52.12	58.49	52.33	34.62	66.00
Flash-Searcher ⚡ (pass@1)	DeepSeek V3.2	60.61	79.25	53.49	46.15	68.00
Self-Evolving Memory Architectures						
MemEvolve + ⚡ (pass@1)	Kimi K2	61.21	67.92	63.95	38.46	68.00
MemEvolve + ⚡ (pass@1)	DeepSeek V3.2	67.88	83.02	63.95	50.00	70.00
MemScope + ⚡ (pass@1)	Kimi K2.7 Code	78.18	88.68	75.58	65.38	74.00
MemScope + ⚡ (pass@1)	DeepSeek V3.2	**83.64**	**90.57**	**81.40**	**76.92**	**78.00**

Open-source agent frameworks. Cognitive Kernel-Pro has the highest GAIA average in this group at 75.15% pass@3; MemScope reaches 83.64% pass@1. Under DeepSeek V3.2 and pass@1, MemScope + Flash-Searcher improves on Flash-Searcher from 60.61% to 83.64% on GAIA and from 68% to 78% on xBench.

Self-evolving memory architectures. Under DeepSeek V3.2, MemEvolve + Flash-Searcher reaches 67.88% on GAIA and 70% on xBench, while MemScope + Flash-Searcher reaches 83.64% and 78%, respectively. The ordering is consistent across both benchmarks.

With Kimi K2.7 Code, MemScope reaches 78.18% on GAIA and 74% on xBench (Table 2).

Following MemEvolve (Zhang et al., 2026a), we evaluate xBench on the full DeepSearch-2505 set using a 40-step budget and GPT-4o-mini judging. To show how often evolution retains a prior success, recovers a failure, or causes a regression, Table 3 pairs original and final evolved DynamicCheatsheet outcomes on the same 70 questions (31–100). Within each backbone, both providers use Flash-Searcher and start with independent empty memory stores that update online from trajectories and post-task correctness feedback, without the judge's reference answer. The evolved versus original providers answer 55/70 versus 49/70 correctly with DeepSeek V3.2 and 53/70 versus 50/70 with Kimi K2.7 Code.

Table 3 decomposes the gains into retained successes, recovered failures, and regressions. The evolved provider retains 95.92% and 92.00% of original successes under DeepSeek and Kimi, respectively, with recovered failures outnumbering regressions in both runs.

4.3 Generalization to Unseen ALFWorld Tasks

Table 3: Paired xBench outcomes on 70 questions. C/I denote correct/incorrect; transitions are from the original to the evolved provider.

Backbone	**C → C**	**I → C**	**C → I**	**I → I**
DeepSeek V3.2	47	8	2	13
Kimi K2.7 Code	46	7	4	13

Table 5: GAIA diagnostic ablation: accuracy (%), mean time, and steps.

Experimental setting		**Diagnostic inputs**			**Task accuracy (%)**				**Resources**	
Setting	**Provider**	**Evidence**	**Memory**	**History**	**Overall**	**L1**	**L2**	**L3**	**Time (s)**	**Steps**
No Memory	×	None	×	×	56.67	75.0	58.82	20.0	517.61	11.30
Original Provider	✓	None	×	×	56.67	62.5	64.71	20.0	771.05	32.53
Summary-only	✓	Summary	×	×	63.33	75.0	70.59	20.0	993.14	40.17
Trace-only	✓	Raw Trace	×	×	70.00	**87.5**	70.59	40.0	921.93	30.37
Trace + Memory Inspection	✓	Raw Trace	✓	×	**86.67**	75.0	**94.12**	**80.0**	804.41	25.73
MemScope	✓	Raw Trace	✓	✓	83.33	**87.5**	82.35	**80.0**	775.29	33.07

Following ALMA (Xiong et al., 2026a), we evaluate ALFWorld on the full `valid_unseen` set, disjoint from the evolution tasks. All methods start with empty memory and follow the same memory-collection sequence before evaluation, with memory updating online. The comparison uses DeepSeek V3.2, temperature zero, 32-token action outputs, 30 outer agent steps, and a 50-step environment cap. Original and evolved DynamicCheatsheet share the runner and task manifests; the ALMA row evaluates an existing design under this task sequence.

Table 4: ALFWorld `valid_unseen` success under DeepSeek V3.2 (correct/67, %).

System	**Success**
DynamicCheatsheet	34/67 (50.75)
ALMA	45/67 (67.16)
MemScope	**48/67 (71.64)**

Under this shared local evaluation, MemScope succeeds on 48/67 ALFWorld unseen tasks (71.64%), versus 34/67 (50.75%) for the original DynamicCheatsheet (Suzgun et al., 2026) and 45/67 (67.16%) for ALMA (Xiong et al., 2026a) (Table 4). The same original-versus-evolved ordering appears in xBench's matched-start comparison, extending the evidence from web research to unseen sequential-action tasks.

4.4 Ablation and Analysis

Following the evolution procedure used in the main experiments, we vary the diagnostic information available to the Patch Agent to examine its effect on localized repair (Table 5). Summary-only receives candidate reports; Trace-only adds raw trace tools; Trace + Memory Inspection additionally exposes the store; MemScope adds Patch History. No Memory and Original Provider are reference controls.

The Trace + Memory Inspection variant achieves the highest GAIA diagnostic accuracy at 86.67%, versus 70.00% for Trace-only and 63.33% for Summary-only. The Patch History variant reaches 83.33%, versus 86.67% for Trace + Memory Inspection; the one-task difference comprises three tasks answered correctly only without history and two answered correctly only with history. Relative to Summary-only, Trace-only uses 24.4% fewer average task steps and 7.2% less task time. These resource differences describe downstream task execution.

How Patch History informs subsequent repair. The evolution records show how history guides later edits. In the xBench DeepSeek run, a patch raised the retrieval threshold from 0.7 to 0.8 to suppress irrelevant memories. A later repair consulted recovery and regression outcomes, recognized over-filtering, and adjusted the threshold to 0.75 while retaining protection against stale guidance. In ALFWorld, the Patch Agent recalled that lowering the threshold to 0.2 had admitted weakly related

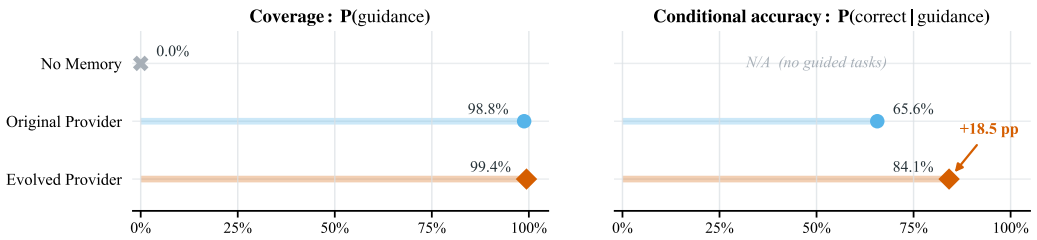

Figure 3: GAIA guidance during online memory updates. Original versus evolved provider coverage: 98.8% versus 99.4%; accuracy among guided tasks: 65.6% versus 84.1%. Code and memory state both vary.

memories; it kept the current 0.3 threshold and changed the no-match fallback to return empty guidance rather than an unrelated old cheatsheet. Both subsequent revisions were installed.

Across 14 archived provider transitions, the median code-change ratio is 4.4% and the median transition touches one function (Appendix Table 8). Appendix B.5 discusses the diagnosis, code-modification, and validation costs of evolution.

Table 6: A recorded GAIA repair case under different diagnostic interfaces.

Aspect	**Summary-only**	**Trace-only**
Evidence	Grouped failure reports	Zero-length traces; empty guidance steps
Proposed diagnosis	Insufficient cross-verification	Early-return guard under `MemoryStatus.IN` may suppress guidance
Edit	Add a verification rule to the storage prompt	Remove the early-return guard

Evidence-grounded repair decisions. Table 6 contrasts two recorded GAIA repair decisions. The trace-based diagnosis identifies a concrete provider-code target, whereas the summary-based diagnosis motivates a generic storage instruction. A complementary xBench case illustrates a decision to leave the provider unchanged: the proposer records `no_patch` after inspecting retrieved memory and noting that failed tasks had already received correct guidance. Its manifest also cites weak outcomes from a previous threshold adjustment. Together, these cases illustrate how execution evidence can inform both the location of a proposed edit and the decision not to modify the provider.

Guidance coverage and conditional accuracy. Figure 3 separates guidance frequency from outcomes when guidance is supplied. On GAIA, coverage is 98.8% for the original provider and 99.4% for the evolved provider, whereas accuracy among guided tasks is 65.6% for the original versus 84.1% for the evolved provider. The larger observed difference is in guided-task outcomes, rather than guidance frequency; both provider code and memory state vary across the compared online trajectories.

5 Conclusion

MemScope turns evolution of an existing memory provider into trace- and memory-grounded code repair. With DeepSeek V3.2, MemScope reaches 83.64% on GAIA and 78% on xBench, versus 67.88% and 70% for MemEvolve + Flash-Searcher. Matched online comparisons favor the evolved over original DynamicCheatsheet on xBench and ALFWorld unseen tasks (71.64% versus 50.75% on ALFWorld). In the GAIA diagnostic ablation, trace-plus-memory accuracy is 86.67% versus 63.33% for summary-only. Together, these results show that localized repair can improve an existing provider during continued use.

AI USE STATEMENT

Generative AI was used to assist with experiment planning and protocol design, implementation and debugging of experimental tooling, organization and interpretation of experimental results, and drafting and revision of this manuscript, including conversion of an earlier draft to the ICLR format. The research method itself uses language models as task executors, a code-generating Patch Agent, and, for web tasks, a correctness judge, as described in the method and experimental sections. Automated checks were used to recompute the additional result tables from retained records and to check the manuscript's compilation and references. These checks do not replace author verification of claims, citations, code, or AI-assisted content. The authors are responsible for the final submitted content and for completing the corresponding disclosure in the submission form.

REFERENCES

Anthropic. Equipping agents for the real world with agent skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, 2025.

Dzmitry Bahdanau, Nicolas Gontier, Gabriel Huang, Ehsan Kamalloo, Rafael Pardinas, Alex Piché, Torsten Scholak, Oleh Shliazhko, Jordan Prince Tremblay, Karam Ghanem, Soham Parikh, Mitul Tiwari, and Quaizar Vohra. Tapeagents: A holistic framework for agent development and optimization, 2024. URL https://arxiv.org/abs/2412.08445.

Igor Bogdanov, Chung-Horng Lung, Thomas Kunz, Jie Gao, Adrian Taylor, and Marzia Zaman. FORGE: Self-evolving agent memory with no weight updates via population broadcast. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 292–310. ACM, May 2026. doi: 10.1145/3786335.3813155. URL http://dx.doi.org/10.1145/3786335.3813155.

Ding Chen, Simin Niu, Kehang Li, Peng Liu, Xiangping Zheng, Bo Tang, Xinchi Li, Feiyu Xiong, and Zhiyu Li. Halumem: Evaluating hallucinations in memory systems of agents, 2025. URL https://arxiv.org/abs/2511.03506.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. In *ECAI 2025*. IOS Press, 2025. doi: 10.3233/FAIA251160. URL https://doi.org/10.3233/FAIA251160.

DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Desearch. Desearch: Web search API for AI agents. https://www.desearch.ai/, 2024.

Yiming Du, Wenyu Huang, Danna Zheng, Zhaowei Wang, Sebastien Montella, Mirella Lapata, Kam-Fai Wong, and Jeff Z. Pan. Rethinking memory in LLM based agents: Representations, operations, and emerging topics, 2025. URL https://arxiv.org/abs/2505.00675.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.866. URL https://aclanthology.org/2026.findings-acl.866/.

Tianqing Fang, Zhisong Zhang, Xiaoyang Wang, Rui Wang, Can Qin, Yuxuan Wan, Jun-Yu Ma, Ce Zhang, Jiaqi Chen, Xiyun Li, Yonglin Wang, Jingchen Ni, Tianshi Zheng, Chun Chen, Wenhao Yu, Zhenwen Liang, Hongming Zhang, Haitao Mi, and Dong Yu. Cognitive kernel-pro: A framework for deep research agents and agent foundation models training, 2025. URL https://arxiv.org/abs/2508.00414.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks, 2024. URL https://arxiv.org/abs/2411.04468.

Google. Langfun. https://github.com/google/langfun, 2023.

H2O.ai. H2O.ai tops GAIA leaderboard: A new era of AI agents. H2O.ai Blog, https://h2o.ai/blog/2024/h2o-ai-tops-gaia-leaderboard/, 2024.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.acl-long.1575. URL https://aclanthology.org/2025.acl-long.1575/.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Bowei Xia, Tao Sun, Ziyu Ye, Zhaoxuan Jin, Yingru Li, Qiguang Chen, Zeyu Zhang, Yifeng Wang, Qianshuo Ye, Bernard Ghanem, Ping Luo, and Guohao Li. OWL: Optimized workforce learning for general multi-agent assistance in real-world task automation, 2025b. URL https://arxiv.org/abs/2505.23885.

Xun Jiang, Feng Li, Han Zhao, Jiahao Qiu, Jiaying Wang, Jun Shao, Shihao Xu, Shu Zhang, Weiling Chen, Xavier Tang, Yize Chen, Mengyue Wu, Weizhi Ma, Mengdi Wang, and Tianqiao Chen. Long term memory: The foundation of AI self-evolution, 2024. URL https://arxiv.org/abs/2410.15665.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kimi Team. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

Joe Laws. Trase tops GAIA leaderboard. Red Cell Partners Perspectives, https://www.redcellpartners.com/perspectives/trase-tops-gaia-leaderboard/, September 2024.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Zhiwei Li and Yong Hu. Skillhone: A harness for continual agent skill evolution through persistent decision history, 2026. URL https://arxiv.org/abs/2606.08671.

Huawei Lin, Peng Li, Jie Song, Fuxin Jiang, and Tieying Zhang. Muse-autoskill: Self-evolving agents via skill creation, memory, management, and evaluation, 2026a. URL https://arxiv.org/abs/2605.27366.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026b. URL https://arxiv.org/abs/2604.25850.

Yitao Liu, Chenglei Si, Karthik R. Narasimhan, and Shunyu Yao. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14179–14198. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.694. URL https://aclanthology.org/2025.acl-long.694/.

Jinghao Luo, Yuchen Tian, Chuxue Cao, Ziyang Luo, Hongzhan Lin, Kaixin Li, Chuyi Kong, Ruichao Yang, and Jing Ma. From storage to experience: A survey on the evolution of LLM agent memory mechanisms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 41622–41652. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.2069. URL http://dx.doi.org/10.18653/v1/2026.findings-acl.2069.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

OpenAI. Hello gpt-4o, 2024a. URL https://openai.com/index/hello-gpt-4o/.

OpenAI. Gpt-4o mini: Advancing cost-efficient intelligence, 2024b. URL https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

OpenAI. Introducing deep research. https://openai.com/index/introducing-deep-research/, 2025.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, Maohao Shen, Vishy Tirumalashetty, George Lee, Jiawei Han, Tomas Pfister, and Chen-Yu Lee. Skillos: Learning skill curation for self-evolving agents, 2026a. URL https://arxiv.org/abs/2605.06614.

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*, 2026b. URL https://openreview.net/forum?id=jL7fwchScm.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Tianrui Qin, Qianben Chen, Sinuo Wang, He Xing, King Zhu, He Zhu, Dingfeng Shi, Xinxin Liu, Ge Zhang, Jiaheng Liu, Yuchen Eleanor Jiang, Xitong Gao, and Wangchunshu Zhou. Flash-searcher: Fast and effective web agents via DAG-based parallel execution. *arXiv preprint arXiv:2509.25301*, 2025.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution, 2025. URL https://arxiv.org/abs/2505.20286.

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

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 3982–3992. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410/.

Tao Ren, Weiyao Luo, Hui Yang, Rongzhi Zhu, Xiang Huang, Yuchuan Wu, Bingxue Chou, Jieping Ye, Jiafeng Liang, Yongbin Li, and Yijie Peng. Scaling self-evolving agents via parametric memory, 2026. URL https://arxiv.org/abs/2606.04536.

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, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021. URL https://arxiv.org/abs/2010.03768.

Miao Su, Yucan Guo, Zhongni Hou, Long Bai, Zixuan Li, Yufei Zhang, Guojun Yin, Wei Lin, Xiaolong Jin, Jiafeng Guo, and Xueqi Cheng. Beyond dialogue time: Temporal semantic memory for personalized llm agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 29935–29951. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1496. URL https://aclanthology.org/2026.findings-acl.1496/.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Jiabin Tang, Tianyu Fan, and Chao Huang. Autoagent: A fully-automated and zero-code framework for LLM agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 42924–42974. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.2129. URL https://aclanthology.org/2026.findings-acl.2129/.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. Agent KB: Leveraging cross-domain experience for agentic problem solving, 2025. URL https://arxiv.org/abs/2507.06229.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems*, volume 33, pp. 5776–5788. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Xiaoxing Wang, Ning Liao, Shikun Wei, Chen Tang, and Feiyu Xiong. AutoAgent: Evolving cognition and elastic memory orchestration for adaptive agents, 2026a. URL https://arxiv.org/abs/2603.09716.

Yuyao Wang, Zhongjian Zhang, Mo Chi, Kaichi Yu, Yuhan Li, Miao Peng, Bing Tong, Chen Zhang, Yan Zhou, and Jia Li. EvoMemBench: Benchmarking agent memory from a self-evolving perspective, 2026b. URL https://arxiv.org/abs/2605.18421.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-Memory: Benchmarking LLM agent test-time learning with self-evolving memory, 2025. URL https://arxiv.org/abs/2511.20857.

Lilian Weng. Harness engineering for self-improvement. *Lil'Log*, July 2026. URL https://lilianweng.github.io/posts/2026-07-04-harness/.

xbench. xbench-DeepSearch, 2025. URL https://huggingface.co/datasets/xbench/DeepSearch. DeepSearch-2505 benchmark release; evaluation code at https://github.com/xbench-ai/xbench-evals.

Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs, 2026a. URL https://arxiv.org/abs/2602.07755.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts llm agents: An empirical study of experience-following behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 623–645. Association for Computational Linguistics, 2026b. doi: 10.18653/v1/2026.acl-long.27. URL https://aclanthology.org/2026.acl-long.27/.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. Autoskill: Experience-driven lifelong learning via skill self-evolution, 2026. URL https://arxiv.org/abs/2603.01145.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2310.02304.

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

Guibin Zhang, Haotian Ren, Chong Zhan, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. MemEvolve: Meta-evolution of agent memory systems. In *International Conference on Machine Learning*, 2026a. URL https://icml.cc/virtual/2026/poster/61379.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves, 2026b. URL https://arxiv.org/abs/2606.09498.

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, 2026c. URL https://arxiv.org/abs/2602.02474.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *SuperIntelligence: Robotics, Safety & Alignment*, 2(3), 2025b. doi: 10.70777/si.v2i3.15063. URL https://doi.org/10.70777/si.v2i3.15063.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory, 2026d. URL https://arxiv.org/abs/2601.03192.

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, 2025c. doi: 10.1145/3748302. URL https://doi.org/10.1145/3748302.

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, pp. 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://doi.org/10.1609/aaai.v38i17.29946.

He Zhu, Tianrui Qin, King Zhu, Heyuan Huang, Yeyi Guan, Jinxiang Xia, Hanhao Li, Yi Yao, Ningning Wang, Pai Liu, Tianhao Peng, Xin Gui, Li Xiaowan, Yuhui Liu, Xiangru Tang, Jian Yang, Ge Zhang, Xitong Gao, Yuchen Eleanor Jiang, Changwang Zhang, Jun Wang, Jiaheng Liu, and Wangchunshu Zhou. Oagents: An empirical study of building effective agents. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 13354–13369. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.720. URL https://aclanthology.org/2025.findings-emnlp.720/.

A PATCH-AGENT REPAIR PROMPT

The following is the recorded Patch-Agent provider-repair prompt template; run-specific inputs appear as placeholders.

A.1 PATCH-AGENT PROVIDER REPAIR

Listing 1: Revised Patch-Agent provider-repair prompt template.

```
You are MemScope PatchProposer, a coding agent for segment-local memory provider repair.

Your job is to make one minimal, evidence-backed patch to the CURRENT provider.
You are NOT designing a new provider, NOT creating a new skill, and NOT exploring a new
     architecture.

The current execution segment's accuracy is below the best prior-episode aggregate accuracy.
      Inspect the available evidence to determine whether a localized code repair is
     warranted.

You do NOT run the full benchmark. You may run only local syntax/contract checks through the
      allowed tools.
The outer trainer records subsequent task outcomes in Patch History to inform future repair.

## Inputs

Current provider identity:
- provider_name: <PROVIDER_NAME>
- provider_class_name: <PROVIDER_CLASS_NAME>
- memory_type: <MEMORY_TYPE>
- skill_name: <SKILL_NAME>
- failure_count: <FAILURE_COUNT>
- success_count: <SUCCESS_COUNT>
- evidence_mode: trace

Score context:
<SCORE_CONTEXT_JSON>

Files:
- original_provider_file: <ORIGINAL_PROVIDER_FILE>
- candidate_provider_file: <CANDIDATE_PROVIDER_FILE>
- skill_file: <SKILL_FILE>

Trace workspace index:
<TRACE_TASK_INDEX_JSON>

Use task_id values from this workspace index when calling trace tools. Do not invent task
     ids. If you need more detail, call list_traces or read the specific task_id from the
     index.

Patch history for this provider:
<PATCH_HISTORY>

Use this history as feedback. If a previous patch direction had weak follow-up outcomes,
     such as low fixed rate, guided_wrong, or success_regressed, do not repeat that direction
      unless the current evidence clearly contradicts the history. Choose a different repair
     direction or explicitly reject the patch.

Important:
- Edit ONLY candidate_provider_file.
- Do NOT edit original_provider_file.
- Do NOT create a new provider file.
- Do NOT create a new skill file.
- Do NOT add a v2 name, v2 class, or v2 skill.
- If a skill update is useful, write it only as learned_fix_note in patch_manifest.json.

## Available Tools

Trace tools:
- list_traces with arguments {"outcome": "all|failed|success"}
- read_trace_header with arguments {"task_id": "..."}
- read_trace_steps with arguments {"task_id": "...", "step_indices": [0, 1]}
- grep_traces with arguments {"pattern": "...", "max_matches": 10, "outcome": "all|failed|
     success"}

Memory inspection:
- memory_store with arguments {"action": "stats|query|list|read", "query": "...", "task_id":
      "...", "step_index": 0, "context": "...", "status": "BEGIN|IN", "entry_id": "...", "
     outcome": "all|failed|success", "limit": 10, "include_filtered": true}
```

```
Provider patch tools:
- read_candidate_provider with arguments {"max_chars": 24000}
- read_skill_file with arguments {"max_chars": 12000}
- edit_candidate_provider with arguments {"old_text": "...", "new_text": "...", "
    expected_occurrences": 1}
- run_candidate_checks with arguments {}
- write_patch_manifest with arguments {"manifest": {...}}

Use the bound native tools directly. Base each next action on the actual tool results
    returned to you. Independent reads may be requested together, but do not request edits,
    checks, or manifest creation before observing their prerequisites. Do not print JSON
    that merely describes a tool call.

If write_patch_manifest reports missing evidence, perform the named missing inspection and
    call write_patch_manifest again. The first rejected manifest is not a durable completion
     artifact. The durable completion artifact is patch_manifest.json.

## Workflow Guidance

Step 1: Inspect evidence.
- Call list_traces, then inspect at least one failed trace in detail.
- Read at least one successful trace, when available, and state what the patch must preserve
    .
- Prefer read_trace_steps for the smallest relevant step range.
- Ground the patch in observed trace evidence. Do not invent trace contents.
- Before proposing retrieval parameter changes, use memory_store to verify whether relevant
    memories exist and whether they are filtered or malformed.
- Compare raw_store_matches, selection_evidence, and provider_execution on at least one
    failed case and one successful case before changing shared retrieval behavior.
- task_outcome is authoritative for answer correctness. A trace status such as "success" may
     only mean execution completed; never treat it as a correct answer when task_outcome is
    "failed" or judgement is incorrect.

Step 2: Inspect current provider.
- Read candidate_provider_file before editing.
- Identify the smallest code location that can plausibly fix the observed failure pattern.

Allowed local patch scope:
- retrieval logic inside provide_memory
- ranking or filtering logic
- thresholds, top_k, score normalization, deduplication
- storage/extraction of fields already compatible with current records
- formatting of memory guidance
- defensive error handling
- small helper functions used by the existing provider

Forbidden scope:
- new provider class
- changed provider class name
- changed MemoryType
- new provider identity
- new skill identity
- new schema that invalidates existing memories
- replacing most of the provider
- unrelated refactor
- broad architecture rewrite
- dataset-specific hardcoded answers, task IDs, or benchmark names

Step 3: Patch candidate provider.
- Use edit_candidate_provider. Do not output a full replacement provider as final text.
- Prefer one small exact replacement. If multiple edits are necessary, keep them targeted.
- Preserve initialize(), provide_memory(), and take_in_memory contracts.
- take_in_memory must return tuple[bool, str].
- initialize must return bool.

Step 4: Validate locally.
- Run candidate checks after editing.
- If checks fail, fix the candidate file and re-run checks.

Step 5: Write patch_manifest.json.
- Write the manifest when you have a local patch, an explicit no-patch decision, or a new-
    provider handoff.
- If the required fix is architecture-level, do not edit code; write needs_new_provider=true
    .
- If current evidence does not justify a safe change, do not edit code; write patch_type=
    no_patch.

Manifest schema when patched:
{
  "patch_type": "local_patch",
  "needs_new_provider": false,
```

```
  "provider_name": "<PROVIDER_NAME>",
  "provider_class_name": "<PROVIDER_CLASS_NAME>",
  "memory_type": "<MEMORY_TYPE>",
  "skill_name": "<SKILL_NAME>",
  "evidence_mode": "trace",
  "candidate_provider_file": "<CANDIDATE_PROVIDER_FILE>",
  "edited_files": ["<CANDIDATE_PROVIDER_FILE>"],
  "trace_tools_used": ["list_traces", "..."],
  "trace_files_consulted": ["..."],
  "failed_pattern": "observed recurring behavior",
  "success_preservation": "successful behavior to preserve",
  "memory_store_diagnostic": "result of store inspection",
  "patch_history_decision": "not_repeating_failed_direction | repeating_with_reason |
     no_history",
  "root_cause": "one concrete code-level failure cause",
  "patch_summary": "one sentence describing the exact patch",
  "expected_effect": "why this should improve the failed segment",
  "risk": "what could regress",
  "validation": {
    "syntax_check": "passed|failed|not_run",
    "import_check": "passed|failed|not_run",
    "isolated_retrieval_check": "passed|failed|not_run"
  },
  "learned_fix_note": "short note appended to the original skill only if accepted"
}

If evidence does not justify a safe patch, use patch_type="no_patch", leave edited_files
     empty, and provide rejection_reason. If a local patch is insufficient, set
     needs_new_provider=true, leave edited_files empty, and provide new_provider_reason.

Start now.
```

B Repair Cases and Evidence

The recorded cases illustrate evidence-guided repair decisions; the table summarizes edit locality.

B.1 GAIA Failure-Localization Case

The GAIA case contrasts a broad storage instruction with a specific retrieval-code change. Summary-only groups two zero-step failures with three incorrect-answer cases, diagnoses insufficient cross-verification, and adds a generic verification rule to the storage prompt. Trace-only inspects `trajectory_length`=0 and empty `memory_guidance_steps`, identifies an early return under `MemoryStatus.IN` as the proposed cause of missing guidance, and removes that guard. The case makes the diagnostic difference concrete: raw execution evidence identifies a code path that the summary-based proposal leaves unchanged.

B.2 Recorded No-Patch Decision

Table 7 gives a complementary xBench case: the Kimi proposer retains the current provider after inspecting retrieved memory, downstream failures, and a previous unsuccessful threshold change. The quoted manifest shows how the evidence can support leaving retrieval unchanged.

Table 7: A recorded `no_patch` decision in xBench.

Original field	Recorded value
`patch_type`	`"no_patch"`
`needs_new_provider`	`false`
`edited_files`	`[]`
`patch_history_decision`	`"not_repeating_failed_direction"`

Original `rejection_reason`

No evidence-backed local provider change is justified. The 0.7 threshold is correctly blocking an irrelevant correct memory in task 28, and lowering it would risk guided_wrong. Tasks 24 and 29 fail after correct guidance is delivered, so retrieval changes cannot address them. The previous threshold-based gating patch already had weak follow-up outcomes (success regressed), so repeating or tightening that direction is unsafe without contradictory evidence.

Source: `cheatsheet_memory_skill_1788529914321_5e7245.patch_manifest.json`.

B.3 Contrasting Retrieval Repairs

The DeepSeek and Kimi cases show that evidence-guided repair can adjust retrieval in either direction. DeepSeek's proposer attributes failures to correct-only retrieval at a 0.7 threshold and fallback to stale global cheatsheet text; its installed patch searches all records, lowers the threshold to 0.5, expands the candidate pool, and boosts correct records. Kimi's proposer observes 37 correct records among 54 and diagnoses weak cross-domain matches under permissive retrieval; its patch raises the threshold from 0.7 to 0.8. The contrast illustrates how the proposed repair depends on the inspected memory and failure pattern rather than applying a uniform threshold change.

B.4 Edit Locality

Table 8 measures edit scope across 14 archived provider-snapshot transitions: one from Trace + Memory Inspection, six from Summary-only, and seven from Trace-only. Median changes involve 16.5 lines and one function, against median provider-size proxies of 369 lines and 16 functions. The median recorded per-transition normalized fraction is 4.3862%, reported as 4.4%. This fraction is computed across individual transitions, rather than by dividing the line-count medians.

Main-table baseline sources. The GAIA and xBench baselines in Table 2 include Alita (Qiu et al., 2025), Langfun Agent (Google, 2023), TraseAgent (Laws, 2024), OpenAI Deep Research (OpenAI,

Table 8: Syntactic changes across archived provider snapshots.

Metric	Snapshot change	Provider-size proxy
Lines	16.5	369
Functions	1	16
Median recorded normalized fraction	4.4%	100%

2025), h2oGPTe (H2O.ai, 2024), Desearch (Desearch, 2024), and OMNE (Jiang et al., 2024). The remaining agent frameworks are OWL (Hu et al., 2025b), TapeAgents (Bahdanau et al., 2024), AutoAgent (Tang et al., 2026), Magentic-One (Fourney et al., 2024), Cognitive Kernel-Pro (Fang et al., 2025), OAgents (Zhu et al., 2025), Agent KB (Tang et al., 2025), and Flash-Searcher (Qin et al., 2025).

B.5 Evolution Cost Analysis

The comparison separates diagnosis, code modification, and post-modification validation. This stage-wise account explains where localized repair changes the work performed during evolution.

Diagnosis. Diagnosis is shared by complete architecture generation and localized repair. MemEvolve diagnoses architectural defects before redesign, and ALMA analyzes candidate designs and evaluation logs (Zhang et al., 2026a; Xiong et al., 2026a). MemScope uses execution evidence and prior repair outcomes to identify a code region for modification.

Code modification. Complete generation emits a full provider implementation for each candidate. MemScope generates replacements for selected regions and reuses the rest of the implementation. The archived edit-size measurements in Table 8 quantify this limited modification scope; they measure changed code rather than model-token expenditure.

Post-modification validation. MemEvolve evaluates candidate architectures on task batches before selecting parents for further evolution (Zhang et al., 2026a). ALMA performs trial runs and debugging, then evaluates candidates through memory collection and deployment (Xiong et al., 2026a). MemScope instead checks a local code update and continues task execution with the updated architecture and retained runtime memory. Patch History records the subsequent recovered failures and regressed successes so later proposals can correct an unsuccessful direction. Behavioral feedback is thus collected along a continuing evolution trajectory, rather than through a separate task evaluation for each of multiple complete candidates.