EVOARENA: TRACKING MEMORY EVOLUTION FOR ROBUST LLM AGENTS IN DYNAMIC ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents perform strongly on many benchmarks, yet most evaluations assume static environments, while real-world agents must continually adapt to changing environments. To address this gap, we introduce **EvoArena**, a benchmark that models progressive environment evolution across terminal, software, and social domains, and **EvoMem**, a patch-based memory paradigm that records memory evolution as structured update histories. Current agents achieve only 39.6% average accuracy on EvoArena, while EvoMem improves EvoArena by 1.5% and standard benchmarks GAIA and LoCoMo by 6.5% and 3.3%, respectively. EvoMem further improves chain-level accuracy by 4% where success requires sustained reliability across evolving states, with analyses showing better preservation and use of evolving evidence. These results highlight the importance of explicitly modeling environmental evolution in both agent evaluation and memory. A subset of EvoArena is included in the supplementary materials and we will opensource the full code and data.

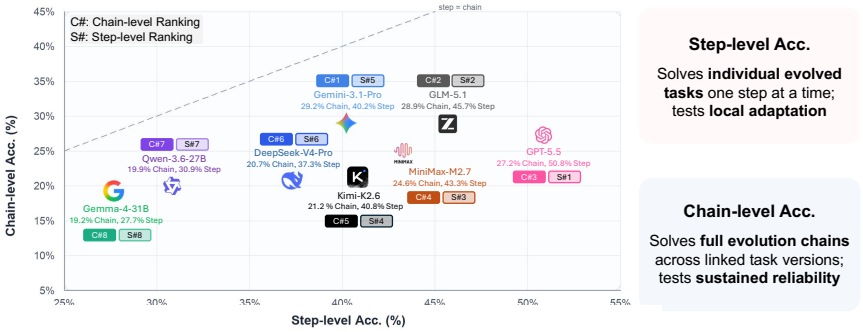


Figure 1: **Step accuracy vs. chain accuracy on EvoArena.** The closer to the upper-right corner the better.

1 INTRODUCTION

LLM agents are increasingly expected to operate in external environments such as web systems, terminal workspaces, software repositories, and personalized assistants. Modern agent systems combine reasoning (Yao et al., 2023), tool use (Zhou et al., 2026), and memory to execute multi-step procedures (Xu et al., 2025), act on environment state, and adjust their behavior within a task episode, achieving strong performance on benchmarks for web navigation (Zhou et al., 2023), software engineering (Jimenez et al., 2024), tool use (Mialon et al., 2024), and general agentic reasoning (Mialon et al., 2024). However, real-world tasks naturally unfold in evolving environments, where interfaces, rules, code states, and user preferences change over time, requiring agents to remain effective across successive environment states.

Yet this progress leaves a central deployment challenge underexplored: **agents are usually evaluated on static environment snapshots.** In most benchmarks, the interface, rules, task distribution, and

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.

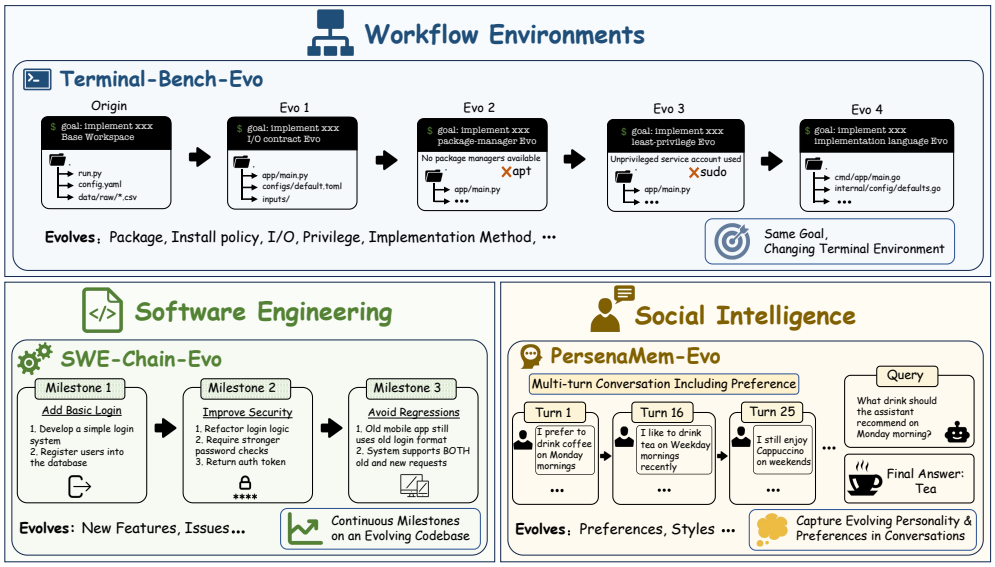


Figure 2: **EvoArena construction.** We convert static agent benchmarks into versioned evolution chains across executable workflows, software engineering, and social intelligence, testing whether agents can adapt to new changes while preserving still-valid prior behavior.

success criteria are fixed once the benchmark is constructed (Jimenez et al., 2024; Liu et al., 2024; Mialon et al., 2024; Zhou et al., 2023). Recent dynamic evaluations improve freshness or interaction realism through refreshed tasks, asynchronous events, or self-evolving instances (Froger et al., 2026; Li et al., 2026a; Wang et al., 2025a), but they rarely test persistent environment evolution, where the same setting changes across versions. In deployment, environments continually evolve, requiring agents to track what changed, what remains valid, and how to act accordingly.

To address this gap, we introduce **EvoArena**, a benchmark for evaluating agents under **persistent environment evolution**. EvoArena organizes each environment into a chain of **progressively evolving releases**, where the same underlying goal or setting is preserved while interfaces, rules, workflows, code states, or user preferences change over time. This formulation turns environment evolution into a measurable capability: an agent must solve the current task, identify which updates matter, and avoid reusing behaviors tied to obsolete versions. Concretely, EvoArena includes **Terminal-Bench-Evo** for evolving terminal workflows, **SWE-Chain-Evo** for evolving codebases, and **PersonaMem-Evo** for evolving user preferences, as shown in Figure 2. Together, these benchmarks evaluate both *forward adaptation* to new changes and *version compatibility* with still-valid prior knowledge, testing whether agents remain reliable beyond static snapshots.

Using EvoArena, we find that even strong agent systems degrade substantially in evolving environments. One recurring failure mode is **state collapse**: most memory-based agents maintain memory as *a single latest state*, such as a retrieved memory bank or episodic store (Chhikara et al., 2025; Xu et al., 2025; Zhou et al., 2026). This design is effective when newer information safely supersedes older information, but becomes brittle when different environment versions require different behaviors. For example, a workflow permission update may overwrite an earlier rule that still applies to an older release or a different organization. The agent then loses both the previous behavior and the context explaining when it was valid. This motivates **version-aware environment tracking**: agents must preserve the current state, recover relevant prior states, and reason about how they changed.

To address these limitations, we introduce **EvoMem**, a lightweight git-like memory paradigm for evolving environments. EvoMem augments a standard memory system with an append-only patch history that records meaningful memory changes. Each patch stores the pre-update memory, post-update memory, update rationale, and supporting evidence from the triggering context. This makes memory evolution *traceable*: agents can inspect what changed, why it changed, and which evidence justified the update. At inference time, the agent retrieves from the latest memory by default, while selecting relevant EvoMem patches to the current query. In this way, EvoMem turns memory updates into a compact evidence trail for reasoning over environment evolution.

Benchmark	Domain	What Evolves?	PE	IC	CE
WebArena (Zhou et al., 2023)	Web	Static tasks	✗	✗	✗
SWE-bench (Jimenez et al., 2024)	Software	Static issues	✗	✗	✗
GAIA (Mialon et al., 2024)	Tool / Web	Static tasks	✗	✗	✗
AgentBench (Liu et al., 2024)	Multi-env.	Static tasks	✗	✗	✗
AndroidWorld (Rawles et al., 2025)	Device control	Static tasks	✗	✗	✗
Terminal-Bench (Merrill et al., 2026)	Terminal	Static tasks	✗	✗	✗
WorkArena++ (Boisvert et al., 2024)	Workflow	Static tasks	✗	✗	✗
PersonaMem (Jiang et al., 2025)	Personalization	User memory	△	△	✗
SWE-bench-Live (Li et al., 2026a)	Software	Task refresh	✗	✗	✗
GAIA2 (Froger et al., 2026)	Tool / Web	Async events	△	✓	✗
Benchmark Self-Evolving (Wang et al., 2025a)	General	Generated tasks	✗	✗	✗
HorizonBench (Li et al., 2026b)	Personalization	Preference changes	△	✓	✗
EvoArena	**Multi-domain**	**Dynamic Env**	✓	✓	✓

Table 1: **Comparison of EvoArena with related agent benchmarks.** PE denotes persistent environment evolution, where the same environment or user state evolves across versions; IC denotes implicit change, where the agent must infer changes from context or interaction; CE denotes chain evaluation, where success is measured over temporally linked task sequences. ✓denotes support, ✗no support, and △partial support.

Empirically, **EvoMem improves agent robustness across both EvoArena and standard long-horizon agent benchmarks**, with an average 1.5% performance gain observed across agents and backbone models. In addition, EvoMem improves average chain-level accuracy by 4%, showing its ability to support agents in completing sequences of related tasks under a continuous environment evolving. Beyond aggregate performance, our analyses clarify why EvoMem helps. Controlled ablations indicate that its gains come from preserving structured transition information rather than simply providing more historical context. On Terminal-Bench-Evo, retrieved patches are increasingly reflected in downstream reasoning and executed commands, suggesting that EvoMem helps agents identify environmental changes and translate them into concrete actions. On SWE-Chain-Evo, it also reduces regressions against previously valid behavior. Together, these results suggest that reliable agents in evolving environments benefit from memory that preserves not only the current state, but also the structured transitions that produced it. Overall, we make the following contributions:

- We introduce **EvoArena**, a benchmark suite for progressively evolving workflow, software, and social environments.
- We propose **EvoMem**, a patch-based memory paradigm that preserves traces of environmental evolution for more robust reasoning.
- We provide controlled evidence that EvoMem's gains arise from structured versioned transitions, and diagnose its effects on transition uptake, software regression, and evolving-evidence preservation.

2 Related Work

Dynamic and Evolving Agent Benchmarks. Agent benchmarks increasingly evaluate realistic interaction settings, including web navigation (Zhou et al., 2023), software engineering (Jimenez et al., 2024), tool use and general agentic reasoning (Liu et al., 2024; Mialon et al., 2024), device control (Rawles et al., 2025), terminal workflows (Merrill et al., 2026), enterprise tasks (Boisvert et al., 2024), and personalized memory (Jiang et al., 2025). Recent dynamic benchmarks improve evaluation freshness or interaction realism through refreshed software tasks, asynchronous events, or generated benchmark variants (Froger et al., 2026; Li et al., 2026a; Wang et al., 2025a). Long-horizon personalization benchmarks further study how agents use evolving user context, but often introduce preference change as a single update rather than a multi-step version history (Li et al., 2026b). Concurrent benchmarks such as SWE-rebench (Badertdinov et al., 2025) and SWE-bench Goes Live (Zhang et al., 2025) primarily improve temporal freshness by continuously collecting new software-engineering tasks. EvoArena instead evaluates persistent environment evolution: the same setting changes across versions, requiring agents to adapt to new releases while preserving behavior that remains valid. We instantiate this setting by extending terminal, software-engineering, and personalization benchmarks into evolving chains.

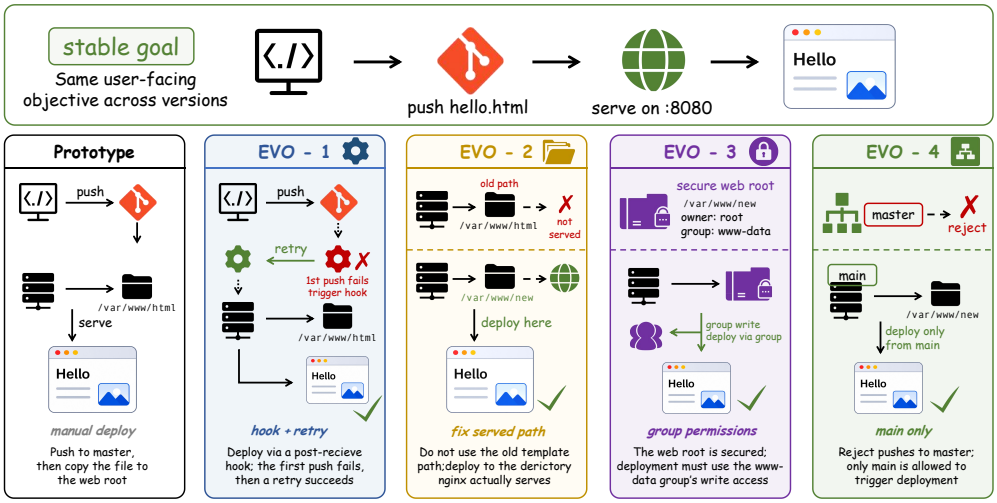


Figure 3: **An example of Terminal-Bench-Evo.** The end goal stays fixed: push `hello.html` and serve it on port 8080, while each version changes a key operational constraint: deployment mechanism, served path, permissions, or branch policy.

Self-Evolving Agents and Memory. Self-evolving agents improve their own behavior through reflection (Shinn et al., 2023), skill accumulation (Wang et al., 2024), reusable skill refinement (Zhou et al., 2026), or scaffold adaptation (Wang et al., 2025b). These methods focus on agent-side improvement, while EvoArena studies changes in the external environment. Memory systems such as structured long-term memory (Xu et al., 2025), production-scale persistent memory (Chhikara et al., 2025), and memory-based agent frameworks (LangChain, 2024) continuously update stored knowledge. However, they usually consolidate memory toward the latest state, which can obscure overwritten states and the reasons behind updates. EvoMem is designed as a complementary memory mechanism rather than a replacement: it augments existing memory systems with structured transition histories that preserve what changed and why.

3 EvoArena: Benchmarking Agents under Environment Evolution

We next describe the construction of **EvoArena** across three complementary settings: executable workflow evolution (Terminal-Bench-Evo), software evolution (SWE-Chain-Evo), and preference evolution (PersonaMem-Evo), as illustrated in Figure 2. Table 11 summarizes the resulting benchmark statistics. We first start with Terminal-Bench-Evo.

3.1 Executable Workflow Evolution: Terminal-Bench-Evo

Terminal-Bench-Evo extends Terminal-Bench (Merrill et al., 2026) into versioned workflow chains $\{v^{(1)}, \ldots, v^{(m)}\}$ that preserve the same user-facing objective while progressively changing operational conditions such as interfaces, dependencies, paths, permissions, and validation rules. For each task, we extract a structured workflow state from its instruction and executable materials, separating the stable objective from mutable components (Appendix D.3.2), then construct an ordered evolution plan using a predefined workflow-change taxonomy (Appendix D.3.3). Each update is materialized by jointly modifying the instruction, executable environment, files/configuration, reference solution, and tests, with version t inheriting the realized state of version $t-1$ so that changes accumulate coherently across the chain (Appendix D.4.1). Each release is then checked for instruction–environment–test consistency and oracle solvability, while inherited changes are verified across versions; ambiguous, inconsistent, or unsolvable releases are repaired or removed (Appendices D.4.2 and D.4.3). We report *Step Accuracy* over individual versioned tasks and *Chain Accuracy*, where a chain is correct only if the entire evolving versions are solved, measuring reliability across accumulated environment changes (Appendix D.3.1). Figure 3 shows an example; the final benchmark contains 89 chains and 441 task instances, with detailed statistics in Appendix D.4.4.

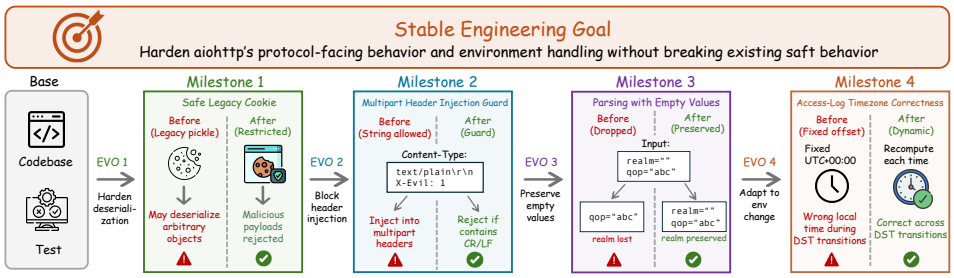


Figure 4: **Example evolution chain in SWE-Chain-Evo.** The aiohttp chain evolves across four milestones, progressively hardening protocol and environment boundaries while preserving existing compatible behavior.

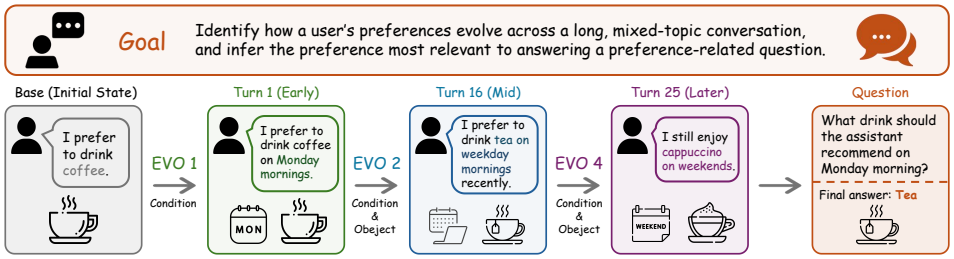


Figure 5: **Example evolution chain in PersonaMem-Evo.** The conversation history reveals evolving user preferences, and the query requires matching the relevant condition to the most appropriate current preference.

3.2 Software Evolution: SWE-Chain-Evo

SWE-Chain-Evo models a code repository as an evolving environment, where each chain $\{m^{(1)}, \ldots, m^{(T)}\}$ consists of chronological, testable development milestones and earlier implementation changes remain active at later steps. We collect active repositories with reproducible test environments from Github (Appendix D.5.1), extract contiguous update windows, and group related commits into coherent milestones while removing unrelated or non-functional changes (Appendix D.5.2). Each milestone is converted into a SWE-bench-style task (Appendix D.6.1) and validated in Docker using Fail-to-Pass and Pass-to-Pass tests (Appendix D.6.2). The milestones are then assembled in chronological order; after each evaluation step, the *reference* update is applied to form the next repository state, preventing earlier agent failures from corrupting subsequent evaluation while preserving the true software history (Appendix D.6.3). We report milestone-level *Step Accuracy* and the fraction of consecutively solved milestones before the first failure as *Chain Accuracy* since requiring all milestones to be correct would leave many agents with near-zero chain accuracy and obscure differences between them. The benchmark contains 50 chains from 12 repositories, with 493 instances and 145 milestones (details in Appendix D.6.4). Figure 4 shows an example chain.

3.3 Social Intelligence Evolution: PersonaMem-Evo

PersonaMem-Evo models user preferences as trajectories $\{p^{(0)}, \ldots, p^{(k)}\}$ whose earlier states remain historically valid while later interactions revise their scope, strength, or applicability. Starting from PersonaMem-v2 (Jiang et al., 2025), we construct structured persona profiles and cleaned preference inventories (Appendix D.7.1). Each preference is expressed through realistic multi-turn interactions that reveal it implicitly through the user's requests, constraints, and behavior; these segments are combined into long mixed-topic conversation histories (Appendix D.8.6). We introduce temporally ordered preference updates while preserving earlier evidence (Appendix D.8.1), then generate preference-inference questions requiring reasoning over these evolving histories (Appendix D.8.2). Candidate questions undergo dual-blind filtering to remove cases answerable without the interaction history, followed by balanced answer-option construction (Appendices D.8.3 and D.8.4). Preference-evolution chains group chronologically related preferences within the same persona conversation based on semantic similarity (Appendix D.8.5). We report *Step Accuracy* over individual questions and *Chain Accuracy*, where a chain is correct only if all questions in that preference-evolution chain

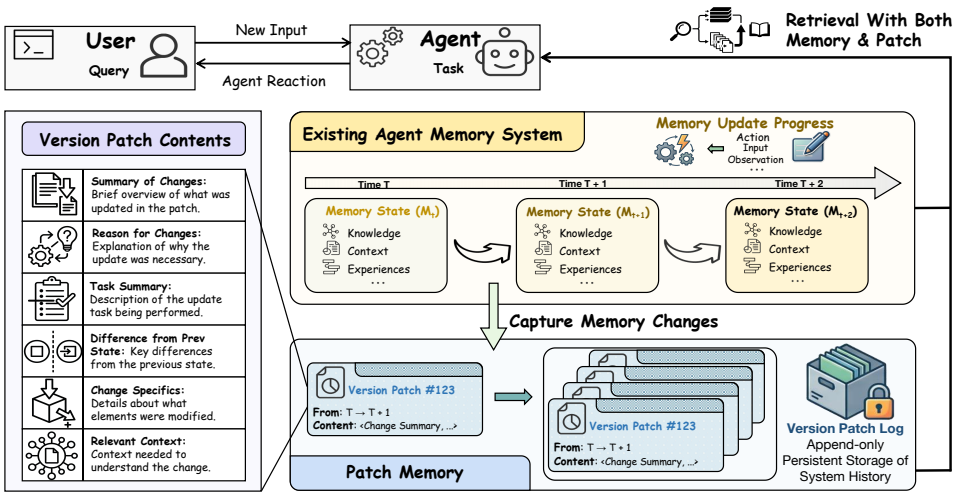


Figure 6: **Overview of EvoMem.** EvoMem augments a base memory system with an append-only patch history that records meaningful memory updates as structured evidence and retrieves relevant patches at inference time.

are answered correctly. The benchmark contains 10 persona conversations, 313 preference chains, and 505 questions, with detailed statistics in Appendix D.8.7. Figure 5 shows an example chain.

4 EvoMem: Patch-Based Memory Evolution

4.1 Memory as an Evolution Trace

EvoArena exposes this challenge by evaluating agents under persistent environment evolution. However, existing memory agents typically maintain a consolidated latest state, which works well when new information simply supersedes old information. In evolving environments, knowledge is often *version-dependent*: an updated workflow rule, implementation strategy, or user preference may replace an earlier state valid under another environment version. Consolidating memory toward the latest state can therefore erase the previous state and the evidence explaining why it changed.

EvoMem addresses this problem by augmenting the base memory system with an explicit *evolution trace*. As shown in Figure 6, EvoMem leaves the underlying memory updater unchanged and adds two lightweight operations: **patch recording**, which preserves meaningful non-additive memory changes, and **patch-augmented retrieval**, which retrieves relevant historical transitions alongside the latest memory when version-specific evidence is useful. We first introduce patch recording.

4.2 Recording Memory Evolution as Patches

Let x_t be the observation at time t and M_{t-1} the current memory state. The base agent updates its memory as $M_t = U(M_{t-1}, x_t)$, where U may correspond to updating a memory bank, user profile, memory graph, or reusable skill memory. Rather than replacing U, EvoMem monitors the resulting transition $\Delta_t = \mathrm{Diff}(M_{t-1}, M_t)$. A patch is created only when the update is *non-additive*, i.e., when existing memory is revised, overwritten, or reinterpreted. Purely additive information remains handled by the original memory system. For each such transition, EvoMem stores a patch $p_t = (\tau_t, C_t^-, C_t^+, r_t, z_t, e_t)$, where τ_t is temporal metadata; C_t^- and C_t^+ are the affected memory contents before and after the update; r_t records why the update occurred; z_t summarizes the change; and e_t stores the supporting interaction, task context, execution feedback, or environment evidence. The append-only patch history $\mathcal{P}_{1:t} = \{p_1, \ldots, p_t\}$ complements the consolidated memory M_t: M_t represents the latest state, while $\mathcal{P}_{1:t}$ preserves the transitions that produced it. This allows the agent to retain overwritten states together with their rationales and supporting evidence rather than collapsing the entire history into a single memory snapshot.

Table 2: **Main results on EvoArena.** We report step accuracy for all benchmarks and chain accuracy for the three EvoArena subsets. Terminal-Bench-Evo and PersonaMem-Evo count a chain as correct only when every step succeeds. SWE-Chain-Evo scores the fraction of milestones solved consecutively from the beginning of the chain, stopping at the first failure.

Benchmark	Agent	Model	Step Acc.			Chain Acc.		
			Base	+EvoMem	Δ	Base	+EvoMem	Δ
Terminal-Bench-Evo	Terminus 2	GPT-5.5	62.8	**65.1**	+2.3	31.8	**45.5**	+13.7
		Gemini-3.1-Pro	53.8	**56.5**	+2.7	39.3	**44.1**	+4.8
		Kimi-K2.6	40.8	**42.9**	+2.1	14.9	**22.7**	+7.8
		Deepseek-V4-Pro	37.3	**40.4**	+3.1	13.5	**22.4**	+8.9
		GLM-5.1	51.8	**55.3**	+3.5	34.2	**36.8**	+2.6
		MiniMax-M2.7	41.0	**42.4**	+1.4	18.2	**19.5**	+1.3
		Qwen3.6-27B	37.6	**40.9**	+3.3	11.1	**17.3**	+6.2
		Gemma4-31B	23.4	**24.5**	+1.1	9.0	**12.4**	+3.4
		Average	43.6	**46.0**	**+2.4**	21.5	**27.6**	**+6.1**
SWE-Chain-Evo	OpenHands	GPT-5.5	49.7	**50.9**	+1.2	12.2	**16.8**	+4.6
		Gemini-3.1-Pro	**20.5**	18.1	-2.4	8.8	**10.2**	+1.4
		Kimi-K2.6	**30.2**	27.6	-2.6	8.5	**12.1**	+3.6
		Deepseek-V4-Pro	26.7	**27.7**	+1.0	8.2	**13.3**	+5.1
		GLM-5.1	34.9	**36.1**	+1.2	9.9	**12.8**	+2.9
		MiniMax-M2.7	41.4	**42.3**	+0.9	14.7	**15.3**	+0.6
		Qwen3.6-27B	11.6	11.6	+0.0	10.1	**12.2**	+2.1
		Gemma4-31B	8.5	**12.0**	+3.5	5.2	**6.3**	+1.1
		Average	27.9	**28.3**	**+0.4**	9.7	**12.4**	**+2.7**
PersonaMem-Evo	A-Mem	GPT-5.5	40.0	**43.8**	+3.8	37.5	**41.2**	+3.7
		Gemini-3.1-Pro	46.4	**48.3**	+1.9	38.8	**40.8**	+2.0
		Kimi-K2.6	51.5	**55.5**	+4.0	40.2	**50.0**	+9.8
		Deepseek-V4-Pro	47.9	**51.6**	+3.7	40.4	**47.4**	+7.0
		GLM-5.1	**50.4**	47.5	-2.9	**42.5**	38.9	-3.7
		MiniMax-M2.7	47.5	**47.9**	+0.4	40.9	**41.4**	+0.4
		Qwen3.6-27B	43.5	**44.4**	+1.0	36.5	**39.9**	+3.4
		Gemma4-31B	51.1	**52.9**	+1.8	43.5	**45.8**	+2.3
		Average	47.3	**49.0**	**+1.7**	40.0	**43.2**	**+3.2**

4.3 Retrieving Versioned Memory Patches

At inference time, EvoMem retains the base agent's standard retrieval from the latest memory, $c_{\text{mem}} = R_{\text{mem}}(q, M_T)$, where q is the current query and M_T is the consolidated memory after all previous observations. In parallel, EvoMem retrieves relevant historical patches as $\mathcal{P}_q = R_{\text{patch}}(q, \mathcal{P}_{1:T})$, where R_{patch} returns the top-k transitions relevant to the current query. These patches expose earlier memory states, the changes that superseded them, and the evidence behind those changes. The final context is $c(q) = \text{Concat}(c_{\text{mem}}, \mathcal{P}_q)$. The latest memory thus provides the agent's current consolidated knowledge, while retrieved patches provide versioned evidence when the query depends on temporal changes, overwritten information, or prior environment states. EvoMem can therefore support version-aware reasoning without replacing the underlying memory architecture. EvoMem is instantiated across heterogeneous agents by defining the corresponding base memory state, patch-triggering updates, and patch retrieval mechanism. Agent-specific implementations for Terminus2 (Merrill et al., 2026), OpenHands (Wang et al., 2025b), A-Mem (Xu et al., 2025), and Memento-Skill (Zhou et al., 2026) are provided in Appendix E.

5 Experiments

5.1 Experimental Setup

We evaluate the state-of-the-art agents with and without EvoMem on the three domains of EvoArena and on two non-evolving agent settings, GAIA (Mialon et al., 2024) and LoCoMo (Maharana et al., 2024) to test EvoMem's generalization beyond evolving environments.

Agents and models. We pair each benchmark with an agent suited to its interaction setting: TERMINUS2 (Merrill et al., 2026) for TERMINAL-BENCH-EVO, OPENHANDS (Wang et al., 2025b) for SWE-CHAIN-EVO, A-MEM (Xu et al., 2025) for PERSONAMEM-EVO and LOCOMO, and MEMENTO-SKILL (Zhou et al., 2026) for GAIA. We evaluate GPT-5.4-MINI (OpenAI, 2026a), GPT-5.5 (OpenAI, 2026b), GEMINI-3.1-PRO (Google, 2026), KIMI-K2.6 (Moonshot AI, 2026), DEEPSEEK-V4-PRO (DeepSeek, 2026), GLM-5.1 (GLM-5 Team, 2026), MINIMAX-M2.7 (MiniMax, 2026), QWEN3.6-27B (Qwen Team, 2026), and GEMMA4-31B (Gemma Team, 2026). Full implementation details are provided in Appendix F.

Evaluation. We report each benchmark's primary task-level metric: task accuracy for EvoArena, LLM-judge accuracy for GAIA, and exact match for LOCOMO. For EvoArena, we additionally report the *Chain Accuracy* defined in Section 3, which measures reliability across evolving environments.

5.2 MAIN RESULTS

Table 2 summarizes the main results and we highlight the following findings. **First, Current agents struggle under persistent environment evolution, especially at the chain level.** Across EVOARENA as shown in Table 2, base agents achieve only moderate step-level performance and degrade substantially under chain-level evaluation, reaching 21.5%, 9.7%, and 40.0% chain accuracy on TERMINAL-BENCH-EVO, SWE-CHAIN-EVO, and PERSONAMEM-EVO, respectively, showing that solving individual states does not necessarily translate into reliable behavior across evolving environment sequences. **Besides, EvoMem improves robustness across both evolving and standard settings.** It improves average step accuracy by 2.4%, 0.4%, and 1.7% across the three EvoArena subsets, while also improving GAIA by 6.5% and LoCoMo by 3.3% on average (Table 3). **The gains are consistently larger at the chain level.** EvoMem improves chain accuracy by 6.1%, 2.7%, and 3.2% on the three EvoArena subsets, with cluster-aware paired permutation tests showing significant improvements on TERMINAL-BENCH-EVO ($p = 0.0125$), SWE-CHAIN-EVO ($p = 0.0473$), and PERSONAMEM-EVO ($p = 0.0285$). These results suggest that preserving memory evolution is useful when agents must maintain relevant knowledge across multiple related environment states.

Table 3: **Results on standard benchmarks.** We report task accuracy for GAIA and exact match for LoCoMo.

Benchmark	Agent	Model	Base	+EvoMem	Δ
GAIA	Memento-S	GPT-5.5	83.0	83.0	+0.0
		Gemini-3.1-Pro	57.0	**65.0**	+8.0
		Gemma4-31B	45.0	**54.0**	+9.0
		Deepseek-V4-Pro	70.0	**80.0**	+10.0
		GLM-5.1	70.0	**77.0**	+7.0
		Qwen3.6-27B	70.0	**75.0**	+5.0
		Average	65.8	**72.3**	**+6.5**
LoCoMo	A-Mem	GPT-5.5	32.9	**33.9**	+1.0
		Gemini-3.1-Pro	21.1	**28.6**	+7.5
		Gemma4-31B	52.3	**55.2**	+2.9
		Deepseek-V4-Pro	52.0	**56.5**	+4.5
		Kimi-K2.6	54.0	**57.7**	+3.7
		Qwen3.6-27B	26.0	**26.3**	+0.3
		Average	39.7	**43.0**	**+3.3**

6 MECHANISM ANALYSIS: WHEN AND WHY DOES EVOMEM HELP?

Beyond aggregate performance, we conduct an ablation study and examine the mechanisms in each EvoArena domain to understand how EvoMem supports adaptation under environment evolution.

6.1 ABLATION STUDY

Ablations confirm the effectiveness of EvoMem's design. We ablate EvoMem with GPT-5.5, comparing its structured transition representation with two simpler alternatives under comparable retrieval budgets: Simple Changelog with compact change summaries, and Git-RAG with version-control metadata and diffs. On PERSONAMEM-EVO, A-Mem also supports a strict matched-token control through its native memory retriever, which is not applicable to Terminus2 and OpenHands. Replacing EvoMem with simpler alternatives consistently lowers performance across all three domains (Table 4), supporting the value of its structured transition design. Implementation details are provided in Appendix C.5.

Table 4: Ablations study.

Method	Terminal	SWE	Persona
Matched-token retrieval	–	–	40.2
Simple Changelog	61.7	48.1	39.8
Git-RAG	61.5	49.9	42.3
EvoMem	**65.1**	**50.9**	**43.8**

6.2 TERMINAL-BENCH-EVO: OPERATIONALIZING RETRIEVED TRANSITIONS

Table 5: Gains under retrieval and uptake conditions.

Factor	Condition	Base	Evo	Δ	Gap
Patch retrieval	No	46.6	49.7	+3.1	
	Yes	87.1	93.5	+6.5	+3.4
Patch uptake	None	46.8	49.4	+2.6	
	> 0	80.6	88.9	+8.3	**+5.7**
Command uptake	None	46.2	49.4	+3.1	
	> 0	87.5	93.8	+6.2	+3.1

Terminal-Bench-Evo changes key execution requirements across versions, making agents prone to outdated behavior. EvoMem should help when retrieved patches inform reasoning and commands, leading to larger gains. We test this hypothesis through three observational signals: *patch retrieval*, whether a transition patch is retrieved; *patch uptake*, whether its content appears in later reasoning or actions; and *command uptake*, whether it appears in executed shell commands. Base is evaluated on the same EvoMem-defined subsets. **EvoMem helps more when retrieved transitions are operationalized.** Its gain increases from 3.1% to 6.5% with patch retrieval, from 2.6% to 8.3% with patch uptake, and from 3.1% to 6.2% with command uptake (Table 5), supporting the hypothesis that historical transitions are most useful when carried into reasoning and concrete actions.

6.3 SWE-CHAIN-EVO: PRESERVING HISTORICAL BEHAVIOR

Table 6: **P2P regression rate on SWE-Chain-Evo.** Lower is better.

Model	Base	+EvoMem	Δ
Qwen3.6-27B	9.01%	6.73%	−2.28%
Kimi-K2.6	7.14%	3.33%	−3.81%
Gemini-3.1-Pro	11.11%	8.89%	−2.22%
Average	**9.09%**	**6.32%**	−2.77%

EvoMem reduces regressions during software evolution. A key challenge in SWE-CHAIN-EVO is implementing new requirements without breaking earlier behavior. By preserving prior updates as retrievable transition evidence, EvoMem can surface constraints that should remain valid, motivating our regression analysis. We test this hypothesis using `PASS_TO_PASS` failures, where the current patch breaks previously valid behavior. EvoMem reduces the average regression rate from 9.09% to 6.32% across all evaluated backbones (Table 6), suggesting that transition memory helps agents incorporate new requirements while preserving prior behavior.

6.4 PERSONAMEM-EVO: PRESERVING EVOLVING PREFERENCE EVIDENCE

Table 7: **Preference evidence capture on PersonaMem-Evo.** Cells report Base → EvoMem capture (gain).

Question Type	Clause-level	Row-level
Single-Pattern	74.4 → 74.4 (+0.0)	74.4 → 74.4 (+0.0)
Multi-Pattern	79.9 → 81.6 (+1.7)	56.0 → 59.5 (+3.5)
Conflict Resolution	82.9 → 83.8 (+0.9)	66.7 → 68.6 (+1.9)
Temporal Trajectory	98.5 → 99.2 (+0.7)	92.2 → 96.6 (+4.4)
Overall	89.4 → **90.3** (+0.9)	72.5 → **74.9** (+2.4)

Patch histories improve complete evidence preservation. In PERSONAMEM-EVO, answering correctly often requires recovering multiple preference states scattered across a long interaction history. By preserving preference updates as structured patches, EvoMem should retain more of the evidence needed for downstream reasoning. We test the hypothesis through gold-evidence capture at the *clause level*, measuring whether each atomic preference is preserved, and the *row level*, requiring all evidence for an example to be retained. EvoMem improves overall clause-level capture by 0.9% and row-level capture by 2.4% (Table 7). The largest row-level gains occur on temporal trajectory (+4.4%) and multi-pattern synthesis (+3.5%), supporting the role of patch histories in preserving complete evolving preference evidence.

7 CONCLUSION

In this paper, we introduce EvoArena, a benchmark for evaluating LLM agents in evolving environments. We show that current agents degrade substantially under such evolving conditions, especially at the chain level, and propose EvoMem, a patch-based memory paradigm that records memory changes and their update context. Across EvoArena and standard agent benchmarks, EvoMem improves performance and better preserves evolving evidence. Controlled ablations and mechanism analyses further validate EvoMem's design and clarify how its transition memory improves agent behavior in evolving environments. We hope this work supports future research on agents that can track changes, recover historical context, and adapt reliably in real-world evolving settings.

AI USE STATEMENT

Generative AI tools were used for limited language polishing and coding assistance. All research conceptualization, methodological decisions, experimental design, validation, analysis, and final scientific judgments were made by the authors. Separately, LLMs were used as components of the research methodology, including synthetic benchmark construction for EvoArena and model-based components in the evaluated agent and memory pipelines, as described in the paper. All generated benchmark instances were subject to the validation and filtering procedures described in the appendix. The authors reviewed all AI-assisted work and take full responsibility for the final content.

REPRODUCIBILITY STATEMENT

We provide detailed benchmark construction and validation procedures, dataset statistics, EvoMem implementation details, agent and model configurations, retrieval settings, evaluation metrics, ablation setups, and benchmark-specific experimental protocols in the appendix. Representative examples are also included for all three EvoArena domains. A subset of EvoArena is included in the supplementary materials and we will opensource the full code and data.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier de Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

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 - 28th European Conference on Artificial Intelligence, 25-30 October 2025, Bologna, Italy - Including 14th Conference on Prestigious Applications of Intelligent Systems (PAIS 2025)*, pp. 2993–3000, 2025.

DeepSeek. Deepseek-v4-pro ga release. `https://api-docs.deepseek.com/news/news260813/`, August 2026.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Menard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. In *The Fourteenth International Conference on Learning Representations*, 2026.

Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv preprint arXiv:2406.20094*, 2024.

Gemma Team. Gemma 4 technical report, 2026. URL `https://arxiv.org/abs/2607.02770`.

GLM-5 Team. Glm-5: From vibe coding to agentic engineering, 2026. URL `https://arxiv.org/abs/2602.15763`.

Google. Gemini 3.1 pro: A smarter model for your most complex tasks. `https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/`, 2026.

Bowen Jiang, Yuan Yuan, Maohao Shen, Zhuoqun Hao, Zhangchen Xu, Zichen Chen, Ziyi Liu, Anvesh Rao Vijjini, Jiashu He, Hanchao Yu, et al. Personamem-v2: Towards personalized intelligence via learning implicit user personas and agentic memory. *arXiv preprint arXiv:2512.06688*, 2025.

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.

LangChain. Langgraph documentation. `https://docs.langchain.com/oss/python/langgraph`, 2024. Accessed: 2026-05-05.

Kenan Li, Rongzhi Li, Linghao Zhang, Qirui Jin, Liao Zhu, Xiaosong Huang, Geng Zhang, Yikai Zhang, Shilin He, Chengxing Xie, et al. Repolaunch: Automating build&test pipeline of code repositories on any language and any platform. *arXiv preprint arXiv:2603.05026*, 2026a.

Shuyue Stella Li, Bhargavi Paranjape, Kerem Oktar, Zhongyao Ma, Gelin Zhou, Lin Guan, Na Zhang, Sem Park, Lin Chen, Diyi Yang, et al. Horizonbench: Long-horizon personalization with evolving preferences. *arXiv preprint arXiv:2604.17283*, 2026b.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024.

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 Annual Meeting of the Association for Computational Linguistics*, pp. 13851–13870, 2024.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024.

MiniMax. Minimax m2.7: Early echoes of self-evolution. `https://www.minimax.io/news/minimax-m27-en`, March 2026.

Moonshot AI. Kimi k2.6. `https://www.kimi.com/ai-models/kimi-k2-6`, 2026.

OpenAI. Introducing gpt-5.4 mini and nano. `https://openai.com/index/introducing-gpt-5-4-mini-and-nano/`, 2026a.

OpenAI. Introducing gpt-5.5. `https://openai.com/index/introducing-gpt-5-5/`, 2026b.

Qwen Team. Qwen3.6-27b: Flagship-level coding in a 27b dense model. `https://qwen.ai/blog?id=qwen3.6-27b`, 2026.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William E Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Kenji Toyama, Robert James Berry, Divya Tyamagundlu, Timothy P Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=il5yUQsrjC`.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

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. ISSN 2835-8856.

Siyuan Wang, Zhuohan Long, Zhihao Fan, Xuanjing Huang, and Zhongyu Wei. Benchmark self-evolving: A multi-agent framework for dynamic LLM evaluation. In *Proceedings of the International Conference on Computational Linguistics*, pp. 3310–3328, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for LLM agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2025.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, et al. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

LIST OF APPENDICES

A Limitations and Future Directions **15**

B Broader Impact **15**

C Additional Analysis **16**

C.1 Robustness to Increasing History Length with Difficulty Control 16

C.2 Generalization to an Alternative Host Memory System 16

C.3 Incremental Token and Latency Overhead 16

C.4 Efficiency–Accuracy Trade-off 17

C.5 Implementation Details of Controlled Ablations 17

D EvoArena Dataset Construction Details **18**

D.1 EvoArena Distribution 18

D.2 EvoArena Dataset Statistics 18

D.3 Terminal-Bench-Evo: Task Analysis and Evolution Taxonomy 19

D.3.1 Evaluation Unit and Metrics 19

D.3.2 Original Task Analysis 20

D.3.3 Evolution Taxonomy 20

D.4 Terminal-Bench-Evo: Version Construction and Validation 20

D.4.1 Version-Chain Construction 20

D.4.2 Quality Control 21

D.4.3 Oracle Validation and Solvability Verification 22

D.4.4 Metadata and Dataset Statistics 22

D.5 SWE-Chain-Evo: Repository and Milestone Construction 22

D.5.1 Repository Collection and Domain Coverage 23

D.5.2 Milestone Construction 23

D.6 SWE-Chain-Evo: Evaluation Packaging and Chain Assembly 23

D.6.1 Task Format and Problem Statement Construction 23

D.6.2 Docker Evaluation Environment 24

D.6.3 Chain Assembly 24

D.6.4 Dataset Statistics 24

D.7 PersonaMem-Evo: Persona and Interaction History Construction 24

D.7.1 Seed Persona Expansion and Preference Cleaning 25

D.7.2 Implicit Interaction History Construction 25

D.8 PersonaMem-Evo: Preference Evolution and Benchmark Construction 26

D.8.1 Temporal Preference Evolution 26

D.8.2 Preference-Inference Question Generation 27

D.8.3 Dual-Blind Filtering 29

D.8.4 Answer-Option Construction 29
D.8.5 Benchmark Assembly and Metadata 29
D.8.6 Interaction Formats and Preference Records 29
D.8.7 Dataset Statistics 30

E EvoMem Implementation Details **31**
E.1 Terminus2 with EvoMem 31
E.2 OpenHands with EvoMem 34
E.3 Memento-Skill with EvoMem 36
E.4 A-Mem with EvoMem 38

F Experiments Setting **41**
F.1 Terminal-Bench-Evo 41
F.2 SWE-Chain-Evo 41
F.3 PersonaMem-Evo 41
F.4 GAIA 41
F.5 LoCoMo 42

G Example of EvoArena **42**
G.1 Terminal-Bench-Evo 42
G.2 SWE-Chain-Evo 43
G.3 PersonaMem-Evo 43

A Limitations and Future Directions

EvoArena and EvoMem take a first step toward evaluating and improving LLM agents in evolving environments. Our benchmark focuses on three representative forms of environment evolution: executable workflow changes, software repository evolution, and long-horizon preference shifts. These domains are intentionally chosen because they expose different version-aware capabilities, including adapting to updated interfaces, reasoning over accumulated codebase changes, and preserving temporally grounded user preferences.

This scope also highlights the broader significance of the problem. Environment evolution is a general challenge for deployed agents, and similar dynamics arise in robotics, embodied interaction, scientific workflows, multi-agent collaboration, and other long-running systems. Extending EvoArena to these settings would enable the community to study how agents track physical state changes, scientific protocol updates, collaborative dependencies, and evolving multi-agent roles. We view these extensions as an important direction for building agents that remain reliable across changing real-world conditions.

B Broader Impact

This work studies how LLM agents behave in evolving environments, where workflows, codebases, tools, and user preferences change over time. As LLM agents are increasingly deployed in long-running settings, failures to track such changes can lead to brittle behavior, outdated decisions, or incorrect reuse of prior procedures. EvoArena provides a benchmark suite for evaluating these risks, and EvoMem offers a lightweight memory mechanism for preserving update histories as retrievable evidence. We expect this work to support the development of more reliable agents that can adapt to changing task conditions while retaining relevant prior knowledge.

A positive impact of this work is improved evaluation for agent robustness. Many current evaluations focus on fixed task snapshots, which can overestimate agent reliability in deployment settings where rules, interfaces, and user needs evolve. By modeling versioned environment changes, EvoArena can help researchers diagnose whether agents understand the current state of an environment and whether they can avoid applying obsolete behaviors. This may benefit applications such as software maintenance, enterprise workflow automation, long-horizon personal assistants, and other settings where agents must operate across repeated updates.

EvoMem may also improve transparency in memory-based agents. By recording what memory changed, why it changed, and which evidence triggered the update, patch histories make memory evolution more inspectable. This can help developers audit agent behavior, trace failures to specific memory updates, and distinguish current knowledge from previously valid knowledge. Such traceability is especially important in settings where agent actions affect user-facing decisions, organizational procedures, or software systems.

This work also raises potential risks. More persistent and adaptive memory can strengthen agent capabilities in ways that may be misused, for example by helping agents maintain long-term strategies in adversarial or manipulative tasks. Memory histories may also contain sensitive user information if deployed without appropriate privacy safeguards. In addition, patch retrieval can surface outdated or context-specific information if the agent fails to judge whether a prior state remains applicable. These risks highlight the need for careful access control, data minimization, retention policies, and explicit auditing of memory contents in real deployments.

EvoArena itself is intended for research evaluation and should be used with attention to dataset scope and domain coverage. The benchmark focuses on representative forms of environment evolution, including terminal workflows, software repositories, and long-horizon user preferences. Future work should extend this evaluation to additional domains, larger deployment horizons, and human-centered validation of agent behavior under changing conditions.

Overall, our results suggest that reliable agent deployment requires treating memory as an evolving record of grounded updates. We hope EvoArena and EvoMem encourage future work on agents that can adapt to changing environments in a traceable, auditable, and privacy-conscious manner.

Table 8: **Performance across increasing PersonaMem-Evo history lengths.** Each cell reports Base → EvoMem accuracy, with the improvement in parentheses.

Persona	History Tokens	Overall	L1	L2	L3
18	31,719	37.3 → 45.1 (+7.8)	33.3 → 50.0 (+16.7)	36.8 → 36.8 (+0.0)	40.0 → 50.0 (+10.0)
41	76,245	27.5 → 29.4 (+2.0)	33.3 → 25.0 (-8.3)	31.6 → 36.8 (+5.3)	20.0 → 25.0 (+5.0)
11	82,878	49.0 → 51.0 (+2.0)	41.7 → 58.3 (+16.7)	47.4 → 42.1 (-5.3)	55.0 → 55.0 (+0.0)
21	92,339	43.1 → 39.2 (-3.9)	25.0 → 16.7 (-8.3)	36.8 → 36.8 (+0.0)	60.0 → 55.0 (-5.0)
1	107,381	42.9 → 42.9 (+0.0)	41.7 → 41.7 (+0.0)	35.3 → 41.2 (+5.9)	50.0 → 45.0 (-5.0)
0	241,987	36.0 → 34.0 (-2.0)	50.0 → 50.0 (+0.0)	31.6 → 21.1 (-10.5)	31.6 → 36.8 (+5.3)
10	250,696	38.8 → 49.0 (+10.2)	33.3 → 41.7 (+8.3)	23.5 → 35.3 (+11.8)	55.0 → 65.0 (+10.0)
20	260,241	43.1 → 49.0 (+5.9)	50.0 → 58.3 (+8.3)	31.6 → 42.1 (+10.5)	50.0 → 50.0 (+0.0)
40	271,501	35.3 → 47.1 (+11.8)	25.0 → 41.7 (+16.7)	31.6 → 31.6 (+0.0)	45.0 → 65.0 (+20.0)
30	367,935	47.1 → 51.0 (+3.9)	33.3 → 41.7 (+8.3)	57.9 → 52.6 (-5.3)	45.0 → 55.0 (+10.0)

Table 9: **EvoMem applied to Mem0.**

Benchmark	Model	Mem0	+EvoMem	Δ
LoCoMo	GPT-5.4-mini	26.53	28.57	+2.04
PersonaMem-Evo	GPT-5.4-mini	17.27	19.54	+2.27
PersonaMem-Evo	GPT-5.5	28.15	29.76	+1.61
PersonaMem-Evo	Gemini-3.1-Pro	21.11	28.64	+7.53

C Additional Analysis

C.1 Robustness to Increasing History Length with Difficulty Control

We examine whether EvoMem's benefit degrades as interaction histories grow. We use the complete GPT-5.5 evaluation on PersonaMem-Evo as shown in Figure 8, where Base and EvoMem are evaluated on exactly the same questions. To reduce confounding from intrinsic question difficulty, we additionally control for the construction-defined L1–L3 reasoning levels: L1 covers direct transfer or recognition, L2 requires cross-domain or latent-preference inference, and L3 requires conflict resolution or trajectory-level generalization.

We regress the paired per-question improvement on log-scaled history length while controlling for difficulty as a categorical variable. The estimated effect is $+0.94\%$ for each doubling of history length, with a persona-clustered 95% confidence interval of $[-2.63, +4.51]$ and an exact persona-level permutation test of $p = 0.5673$. Thus, within the evaluated 31.7K–367.9K-token range, we find no evidence that EvoMem's relative benefit systematically declines with longer histories. This analysis does not constitute a controlled noisy-history stress test, which remains an important direction for future evaluation.

C.2 Generalization to an Alternative Host Memory System

EvoMem is designed as an augmentation to an existing memory system rather than a replacement for a particular memory architecture. To test this, we additionally integrate EvoMem with Mem0 (Chhikara et al., 2025) and evaluate the same host system with and without patch-based memory evolution.

As shown in Table 9, EvoMem improves Mem0 across all evaluated settings, indicating that its benefit is not specific to A-Mem's graph-based representation. This supports our intended positioning of EvoMem as a patch-based evolution layer that can augment heterogeneous persistent-memory systems.

C.3 Incremental Token and Latency Overhead

We measure EvoMem's incremental inference cost on PersonaMem-Evo and LoCoMo, where the underlying agents already maintain persistent memory and therefore permit a direct comparison under the same memory pipeline.

Benchmark	Base	EvoMem	Increase
PersonaMem-Evo	13,824	15,044	+8.8%
LoCoMo	7,465	8,392	+12.4%

(a) **Average input tokens.**

Model	Δ Latency
Gemini-3.1-Pro	+4.75 s
MiniMax-M2.7	+3.94 s
GPT-5.5	+4.57 s

(b) **PersonaMem-Evo latency / example.**

Table 10: **Incremental inference overhead of EvoMem.**

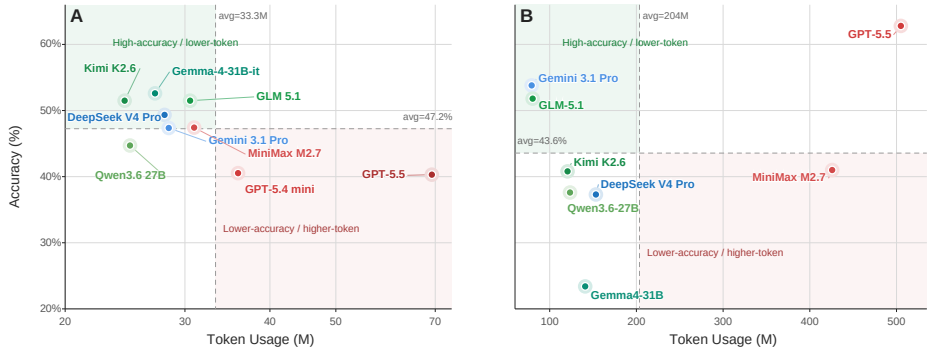


Figure 7: **Accuracy versus total token usage across backbone models.** Lower token usage indicates higher inference efficiency; dashed lines denote cross-model averages.

Table 10 shows that EvoMem increases average input-token usage by 8.8% on PERSONAMEM-EVO and 12.4% on LOCOMO, with an additional end-to-end latency of approximately 4–5 seconds per PersonaMem-Evo example for the evaluated models. Because patch histories are append-only, storage grows with the number of recorded revisions, while inference-time context remains bounded by the retrieval budget.

C.4 EFFICIENCY–ACCURACY TRADE-OFF

Higher token usage does not necessarily translate into better accuracy. Figure 7 shows that higher token consumption does not consistently translate into stronger performance. On PERSONAMEM-EVO, several models achieve above-average accuracy with below-average token usage; for example, Gemma4-31B reaches 51.1% accuracy with 27.1M tokens, while Kimi-K2.6 achieves 51.5% with only 24.5M. In contrast, GPT-5.5 uses 69.2M tokens but reaches 40.0% accuracy. A similar trade-off appears on TERMINAL-BENCH-EVO: GPT-5.5 achieves the highest accuracy (62.8%) but consumes 505.0M tokens, whereas Gemini-3.1-Pro and GLM-5.1 obtain competitive accuracy (53.8% and 51.8%) with substantially lower usage (79.2M and 80.3M tokens). Overall, these results indicate that longer trajectories alone do not guarantee better agent performance, motivating joint consideration of accuracy and inference efficiency.

C.5 IMPLEMENTATION DETAILS OF CONTROLLED ABLATIONS

We conduct controlled ablations on the full evaluation sets of all three EvoArena domains using GPT-5.5 and their corresponding host agents: Terminus2 for TERMINAL-BENCH-EVO, OpenHands for SWE-CHAIN-EVO, and A-Mem for PERSONAMEM-EVO. All variants use the same host agent and task instances as EvoMem, while alternative representations use comparable or larger retrieval budgets.

Simple Changelog replaces EvoMem patches with compact summaries of environment changes: workflow and configuration updates for Terminal, repository and code changes for SWE, and memory or preference updates for Persona. **Git-RAG** represents these revisions as Git-style commits and uses commit metadata and diffs as the retrieval source, without EvoMem's structured transition fields such as rationale and fine-grained before/after evidence. On PERSONAMEM-EVO, we additionally evaluate **Matched-token Retrieval**, which removes patch retrieval and increases A-Mem's standard retrieval depth so that it receives at least as many retrieved tokens as EvoMem. This control is not applicable

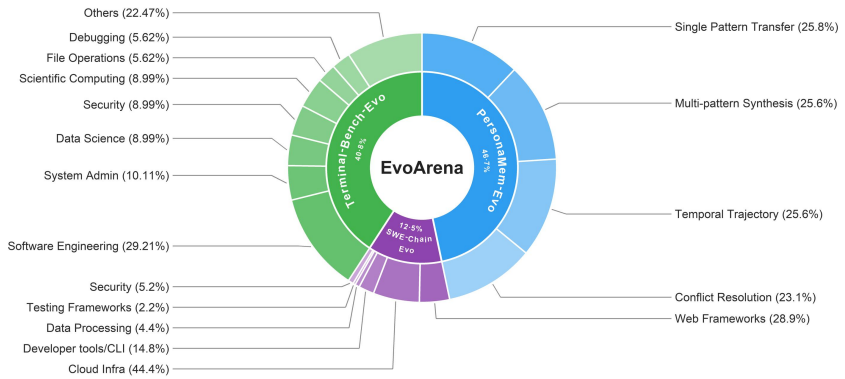


Figure 8: **Distribution of EvoArena.** The central circle shows the distribution across the three EvoArena domains, while the surrounding panels show the task or question-type distribution within each domain.

TERMINAL-BENCH-EVO		SWE-CHAIN-EVO		PERSONAMEM-EVO	
Statistics	**Numbers**	**Statistics**	**Numbers**	**Statistics**	**Numbers**
Init. tasks / chains	89	Repositories	12	Persona conv.	10
Constructed evol.	356	Evol. chains	50	Pref. chains	313
Invalid removed	4	Unique miles.	145	Total inst.	505
Evolved tasks	352	Total inst.	493	Avg conv. length	174.7K
Total inst.	441	Avg commits / mile.	1.00	Avg turns	597
Chain length	4.96 / 5	*Chain steps*		Q / persona	50.5
Len. range	4–5	Mean	9.86	Q type counts	130/129/129/117
Difficulty		5-step / 7-step	10 / 9	Temp. src.	6 / 6.2
Easy	20	10-step / 12-step	10 / 7	*Difficulty*	
Medium	268	13-step / 15-step	7 / 7	Easy	120
Hard	152	Avg P2P tests	25.85	Medium	186
Expert	1	Avg F2P tests	7.13	Hard	199

Table 11: **Key statistics of the three EvoArena subsets.** Chain length reports mean / median. P2P and F2P denote Pass-to-Pass and Fail-to-Pass tests.

to Terminus2 or OpenHands because they do not provide an analogous native persistent-memory retriever.

D EVOARENA DATASET CONSTRUCTION DETAILS

D.1 EVOARENA DISTRIBUTION

Figure 8 summarizes the composition of EvoArena across its three domains and the corresponding task or question types within each subset.

D.2 EVOARENA DATASET STATISTICS

Figure 2 illustrates how EvoArena converts the three source benchmarks into versioned evolution chains, while Table 11 summarizes the resulting scale and structure. TERMINAL-BENCH-EVO contains 89 workflow chains and 441 executable task instances, with nearly all chains spanning five versions. SWE-CHAIN-EVO includes 50 evolution chains from 12 repositories, covering 493 chain-step instances and 145 unique software milestones with executable regression tests. PERSONAMEM-EVO contains 10 long-horizon persona conversations, 313 preference-evolution chains, and 505 preference-inference questions spanning multiple reasoning types and difficulty levels. Together, the three subsets cover complementary forms of environment evolution while differing substantially in interaction length, chain structure, and evaluation unit. More fine-grained statistics for each subset are provided in their corresponding construction sections.

Algorithm 1 Construction of Terminal-Bench-Evo

Require: Original Terminal-Bench tasks $\mathcal{X} = \{x_1, \ldots, x_{89}\}$; evolution taxonomy $\mathcal{G}$; validation executor $\mathcal{V}$
Ensure: Terminal-Bench-Evo benchmark $\mathcal{B}$

$\mathcal{B} \leftarrow \emptyset$
for each original task $x_i \in \mathcal{X}$ **do**
  $a_i \leftarrow \textsc{AnalyzeTask}(x_i)$ ▷ Extract objective, environment, interfaces, I/O contracts, dependencies, and tests
  $\Pi_i \leftarrow \textsc{ConstructEvolutionPlan}(x_i, a_i, \mathcal{G})$
  $E_i^{(0)} \leftarrow \textsc{InitializeEnvironment}(x_i)$
  $C_i \leftarrow \emptyset$
  for $t = 1$ to $|\Pi_i|$ **do**
    $\pi_i^{(t)} \leftarrow \Pi_i[t]$
    $(I_i^{(t)}, E_i^{(t)}, T_i^{(t)}, M_i^{(t)}) \leftarrow \textsc{RealizeVersion}(E_i^{(t-1)}, \pi_i^{(t)})$ ▷ Instantiate instruction, environment, tests, and metadata
    if $\textsc{ValidateVersion}(I_i^{(t)}, E_i^{(t)}, T_i^{(t)}, \mathcal{V})$ **then**
      $v_i^{(t)} \leftarrow (I_i^{(t)}, E_i^{(t)}, T_i^{(t)}, M_i^{(t)})$
      $C_i \leftarrow C_i \oplus v_i^{(t)}$
    end if
  end for
  $C_i \leftarrow \textsc{VerifyChainConsistency}(C_i)$
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{C_i\}$
end for
return $\mathcal{B}$

D.3 Terminal-Bench-Evo: Task Analysis and Evolution Taxonomy

Goal. *Terminal-Bench-Evo* is designed to evaluate whether an agent can adapt to an evolving executable terminal environment while solving the same underlying workflow objective. Unlike static terminal benchmarks where the task instruction, environment, and validation rules remain fixed, Terminal-Bench-Evo emphasizes *temporal workflow evolution*: dependencies, toolchains, directory layouts, interfaces, I/O contracts, and validation requirements may change across versions. Agents must therefore infer the current environment state and adjust their solution, instead of reusing a previously valid procedure blindly.

Built on top of Terminal-Bench (Merrill et al., 2026), Terminal-Bench-Evo converts all 89 original tasks into evolving workflow chains. For each original task x_i, we construct a chronologically ordered chain

$$C_i = \{v_i^{(1)}, \ldots, v_i^{(m_i)}\},$$

where each version $v_i^{(t)}$ contains a version-specific task instruction, executable environment, and validation tests. All versions in C_i preserve the same high-level objective as x_i, but differ in the surrounding execution conditions, such as environment configuration, command-line interface, filesystem structure, dependency setup, input/output contract, or evaluation logic.

Each $v_i^{(t)}$ is an executable benchmark episode with a fixed instruction and environment; the environment does not change during that episode. The chain C_i provides the release history used to organize inherited changes and to compute chain-level reliability. We summarize the construction pipeline in Algorithm 1. The pipeline consists of original-task conversion, version-update design, executable-environment realization, version-specific validation, quality filtering, and benchmark assembly. The following paragraphs describe the evaluation unit, evolution taxonomy, environment realization, metadata, validation procedure, and dataset statistics.

D.3.1 Evaluation Unit and Metrics

Terminal-Bench-Evo separates the executable unit from the chain-level aggregation. A single benchmark episode corresponds to one versioned task instance $v_i^{(t)}$: the agent is given the version-

specific instruction and container/workspace, then its terminal actions are evaluated by the tests attached to that version. Thus, "evolving" does not mean that the filesystem or dependency state mutates unexpectedly during one run; it means that the same workflow objective is released as a sequence of discrete, executable versions.

We report two metrics for this setting. *Step accuracy* is the average success rate over all versioned task instances, so each solved version contributes one correct step. *Chain accuracy* is stricter: a workflow chain C_i is counted as correct only if all of its versions are solved under the evaluation protocol. Step accuracy measures adaptation to the current executable state, while chain accuracy measures whether an agent remains reliable across the full inherited workflow history. In memory-based experiments, versions from the same chain can be executed in chronological order so that earlier observations are available to later episodes; different chains remain independent.

D.3.2 Original Task Analysis

For each original task x_i, we first analyze its executable workflow components before constructing the evolution chain. Specifically, we identify the task objective, required commands, dependency and toolchain assumptions, relevant source and configuration files, input and output contracts, filesystem layout, and validation logic. This analysis yields a structured task state

$$s_i = (o_i, e_i, f_i, d_i, c_i, r_i),$$

where o_i denotes the high-level objective, e_i the executable environment, f_i the relevant files and directory structure, d_i the dependency and toolchain requirements, c_i the interface and I/O contracts, and r_i the validation rules.

The structured state s_i serves as the basis for constructing version updates. Each update modifies one or more surrounding workflow components while preserving o_i, ensuring that different versions remain instances of the same underlying task instead of unrelated terminal problems. In addition to recurring evolution dimensions shared across many tasks, we also curate task-specific evolutions when the original workflow admits a natural semantic change, such as migrating the required implementation language from R to Python or changing the form in which the final artifact must be produced. These task-specific updates are recorded under the *Other* category in our evolution statistics.

D.3.3 Evolution Taxonomy

Based on the structured task state s_i, we assign each non-initial version update a primary evolution label $g_i^{(t)} \in \mathcal{G}$. The taxonomy is organized around the main components of an executable terminal workflow that may naturally change over time: problem-facing contracts such as input/output paths, formats, schemas, or protocols; invocation interfaces such as CLI/API arguments and configuration entries; execution substrates such as dependencies, runtimes, package managers, toolchains, and environment capabilities; workspace organization such as directory layouts, module paths, imports, and staging flows; and task semantics or evaluation policies such as stricter constraints, security requirements, updated target rules, or additional edge cases.

We choose these dimensions because they jointly cover the main layers that define a terminal task beyond its high-level objective: what data the task consumes and produces, how the workflow is invoked, what execution environment it assumes, how the workspace is organized, and how correctness is evaluated. Each dimension can evolve independently while the underlying objective remains unchanged, making it suitable for constructing temporally evolving versions of the same task without turning them into unrelated new tasks. Updates that are natural to a specific workflow but do not fit the recurring categories are retained as task-specific evolutions and grouped under the *Other* category. The full category distribution is reported in Table 12.

D.4 Terminal-Bench-Evo: Version Construction and Validation

D.4.1 Version-Chain Construction

For each original task x_i, we use an agent-assisted construction pipeline to generate a set of candidate version updates grounded in the structured task state s_i. Each candidate describes a plausible evolution

Evolution category	Count	Ratio
I/O-output path contract	118	33.5%
CLI/API surface change	37	10.5%
Format/protocol change	35	9.9%
Language/version/toolchain	25	7.1%
I/O-input path contract	20	5.7%
Staging/promotion flow	19	5.4%
Naming/import/module contract	16	4.5%
Workspace/source layout	12	3.4%
Security/policy hardening	9	2.6%
Evaluation target rule	7	2.0%
Environment capability toggle	3	0.9%
Other task-specific evolution	51	14.5%
Total	352	100.0%

Table 12: Distribution of evolution categories in Terminal-Bench-Evo. Counts are computed over non-initial version updates, excluding the first version of each workflow chain.

Workflow component group	Count	Ratio
I/O or protocol contract	173	49.1%
Workspace, module, or staging flow	47	13.4%
CLI/API invocation surface	37	10.5%
Dependency, toolchain, or capability	28	8.0%
Semantic, policy, or evaluation rule	16	4.6%
Other task-specific evolution	51	14.5%
Total	352	100.0%

Table 13: Grouped evolution statistics for Terminal-Bench-Evo. Groups aggregate the fine-grained categories in Table 12.

of the original workflow, such as modifying the invocation interface, dependency setup, filesystem layout, I/O contract, execution constraints, or validation requirements. We inspect these candidates for semantic plausibility and objective preservation, retaining only updates that represent natural changes to the surrounding terminal workflow while keeping the high-level objective o_i unchanged.

For each retained update, we instantiate the full task materials, including the revised instruction, executable environment, validation tests, reference solution, and metadata. The pipeline provides initial task materials and solutions when applicable, while reference solutions for difficult or failure-prone versions are constructed or corrected by human experts to ensure reliability. The retained updates are then ordered from simpler to more complex changes and composed into a single evolution chain. During chain assembly, version t is constructed by applying its update to the realized environment of version $t-1$. Unchanged files, dependencies, paths, permissions, interface conventions, and validation assumptions are inherited, while only the components specified by the current update are patched. Each stage therefore reflects the accumulated workflow history, but is still required to remain a complete and independently solvable terminal task. After assembling the chain, we run oracle validation to verify executability and test correctness, iteratively repair environment or validation issues, and finally perform a semantic consistency check to ensure that the full chain forms a coherent workflow evolution rather than a collection of unrelated variants.

D.4.2 Quality Control

We apply multiple rounds of quality control to ensure that each EVO stage is a valid, executable, and independently solvable Terminal-Bench task instance, beyond a prompt-only variant. For each retained update, we instantiate the full task materials, including the revised instruction, executable environment, source and configuration files, dependency specifications, input/output paths, validation tests, reference solution, and metadata when applicable. We then check that the instruction, environment, and tests are mutually consistent, and that the update preserves the original high-level objective while introducing a meaningful workflow change.

Statistic	Value
Original Terminal-Bench tasks	89
Evolving workflow chains	89
Versioned task instances	441
Non-initial version updates	352
Average versions per chain	4.96
Median versions per chain	5
Minimum versions per chain	4
Maximum versions per chain	5
Four-version chains	4
Five-version chains	85
Executable task environments	441

Table 14: Summary statistics of Terminal-Bench-Evo.

D.4.3 Oracle Validation and Solvability Verification

Each realized version is verified through oracle execution. The reference solution is run in the constructed environment, and its outputs are evaluated using the version-specific validation tests. We reject or repair versions with build failures, nondeterministic behavior, ambiguous instructions, incomplete dependencies, inconsistent validation logic, or mismatches between the task specification and expected outputs. After composing versions into a chain, we further check that later EVO stages correctly inherit earlier modifications while remaining complete standalone tasks. This process ensures that every retained version is executable, testable, and solvable, and that each chain forms a coherent workflow evolution instead of a collection of unrelated variants.

D.4.4 Metadata and Dataset Statistics

For each versioned task instance, we store structured metadata describing its position and role in the evolution chain, including the original Terminal-Bench task identifier, chain identifier, EVO stage index, primary evolution category, changed workflow components, instruction-change summary, environment-change summary, validation-test summary, and reference-solution information. These annotations support both dataset analysis and error analysis by making it possible to identify which types of workflow evolution are most challenging for agents.

Terminal-Bench-Evo contains 89 evolving workflow chains and 441 versioned task instances, with an average of 4.96 versions per chain and a median chain length of 5. Each chain contains four to five validated versions: 85 chains contain five versions and 4 chains contain four versions. Table 12 reports the fine-grained evolution categories over non-initial version updates, Table 13 reports grouped workflow-component statistics, and Table 14 summarizes the overall dataset statistics.

D.5 SWE-Chain-Evo: Repository and Milestone Construction

Goal and task formulation. SWE-Chain-Evo is designed to evaluate agents under continuous software evolution, where the evolving environment is the codebase itself. In this setting, APIs, dependencies, tests, implementation choices, and architectural constraints introduced by earlier changes become part of the environment that later tasks must operate on. SWE-Chain-Evo therefore organizes software tasks into temporally ordered chains that require agents to adapt to accumulated codebase changes.

Formally, each chain is defined as

$$C_i = \{m_i^{(1)}, m_i^{(2)}, \ldots, m_i^{(T_i)}\},$$

where each milestone $m_i^{(t)}$ represents a semantically coherent development objective in repository i. Given the repository state $S_i^{(t-1)}$ before the milestone and a task instruction $q_i^{(t)}$, the agent must produce a code modification that satisfies the milestone requirement. After applying the reference milestone update, the repository transitions to $S_i^{(t)}$, which becomes the starting environment for the next task. This formulation preserves the temporal and dependency structure of real software

Algorithm 2 Milestone Construction for SWE-Chain-Evo

Require: Repository r; commit history H_r; validation pipeline $\mathcal{V}$
Ensure: Milestone set M_r

$M_r \leftarrow \emptyset$
$W_r \leftarrow \textsc{ExtractContinuousWindows}(H_r)$
for each update window $w \in W_r$ **do**
  $G_w \leftarrow \textsc{CandidateCommitTransitions}(w)$
  for each candidate commit transition $g \in G_w$ **do**
    $\Delta_g \leftarrow \textsc{InspectCodeChanges}(g)$
    if $\neg\, \textsc{CoherentObjective}(g, \Delta_g)$ **then**
      continue
    end if
    if $\textsc{AmbiguousOrUnreliable}(g)$ **then**
      continue
    end if
    $q \leftarrow \textsc{SynthesizeTaskDescription}(g)$
    if $\textsc{ManualVerify}(q, g)$ **and** $\textsc{ExecutableAndStable}(q, g; \mathcal{V})$ **then**
      $M_r \leftarrow M_r \cup \{(q, g)\}$
    end if
  end for
end for
return M_r

development, allowing us to test whether agents can adapt to new requirements while maintaining compatibility with previously accumulated codebase changes.

D.5.1 Repository Collection and Domain Coverage

We retain 12 active repositories with reproducible dependencies and executable tests. The latest SWE-Chain-Evo dataset contains 50 evolution chains built from these repositories. Its 493 chain-step instances cover 145 unique repository milestones: a milestone can appear in more than one chain context. The evaluated instances comprise 404 Go and 89 Python chain-step instances. The repository-level domain counts from the earlier dataset version are omitted because they do not describe this evaluated release.

D.5.2 Milestone Construction

For each retained repository, we extract continuous update windows from its commit history and identify candidate transitions. Each retained milestone represents one or more real commit transitions with a coherent, testable development objective. We inspect the commit message, code diff, and local context, and exclude transitions that are unrelated to the evolution chain or cannot be evaluated reliably.

We manually verify each candidate milestone for semantic coherence, executability, and evaluation reliability. For valid milestones, we synthesize a SWE-bench-style task description from the pre-milestone repository state, specifying the intended functionality or maintenance goal without exposing the reference patch. Ambiguous, overly broad, trivial, or unstable candidates are discarded. Algorithm 2 summarizes this construction process.

D.6 SWE-Chain-Evo: Evaluation Packaging and Chain Assembly

D.6.1 Task Format and Problem Statement Construction

Each milestone is converted into a SWE-bench-style task instance. We use the milestone commit message, code-change summary, and local repository context to draft a problem statement, then manually refine it to describe the intended functionality, bug fix, or maintenance requirement. The agent receives the pre-milestone repository snapshot and natural-language task description; the reference patch is reserved for construction and evaluation.

Statistic	Value
Retained repositories	12
Evolution chains	50
Chain-step instances	493
Unique milestones	145
Mean chain length	9.86
Median chain length	10
Chain-length range	5–15
Mean modified files per instance	2.72
Multi-file instances	38.6%
Median gold patch size (diff lines)	53
Mean Fail-to-Pass tests per instance	7.13
Mean Pass-to-Pass tests per instance	25.85

Table 15: Summary statistics of the SWE-Chain-Evo dataset used in the main experiments. Repeated milestones in different chains count as separate chain-step instances.

D.6.2 Docker Evaluation Environment

For evaluation, we build Docker-based environments for the milestone instances using the Repo-Launch pipeline from SWE-Evo. The pipeline installs repository dependencies, configures build and test commands, and runs automated test discovery to identify Fail-to-Pass and Pass-to-Pass tests. Fail-to-Pass tests validate behavior that should be introduced or repaired by the milestone, while Pass-to-Pass tests check that existing behavior is preserved. When the automatic pipeline misses necessary test signals, we inspect the code semantics and manually supplement tests if the target behavior is clear. We remove instances with flaky builds, ambiguous failures, incomplete fields, missing dependencies, or unstable test outcomes.

D.6.3 Chain Assembly

After individual milestones are packaged and validated, we assemble them into temporally ordered chains within the same repository. Each chain is selected from a continuous update window and follows the repository's natural commit order. Given a chain $C_i = \{m_i^{(1)}, \ldots, m_i^{(T_i)}\}$, the repository snapshot before $m_i^{(1)}$ serves as the initial state. After each step, the corresponding reference milestone update is applied to obtain the repository state for the next step. Thus, later tasks are evaluated on codebases that already include accumulated changes from earlier milestones.

We manually review each assembled chain for temporal consistency, semantic continuity, executable dependency, and appropriate difficulty. Chains are discarded if later milestones do not depend on or naturally follow the preceding repository state, if the sequence mixes unrelated development objectives, or if any step cannot be reliably evaluated in the constructed Docker environment. This ensures that SWE-Chain-Evo evaluates cumulative codebase evolution: agents must satisfy new requirements while respecting constraints introduced by earlier development steps.

D.6.4 Dataset Statistics

Table 15 summarizes the latest SWE-Chain-Evo dataset used in the main experiments: 50 chains from 12 repositories, with 493 chain-step instances and 145 unique milestones. Chains contain 5–15 steps (mean 9.86; median 10). The 493 instances, rather than the deduplicated milestone count, are the unit of step-level evaluation.

For evaluation, each chain-step instance includes at least one Fail-to-Pass test. The latest dataset averages 7.13 Fail-to-Pass tests (median 2) and 25.85 Pass-to-Pass tests (median 4) per instance, with no instance lacking Pass-to-Pass tests.

D.7 PersonaMem-Evo: Persona and Interaction History Construction

Goal. *PersonaMem-Evo* is designed to evaluate whether an agent can infer, track, and generalize a user's evolving latent preferences from long-horizon interaction histories. Unlike factual memory

benchmarks where target information is often explicitly stated, PersonaMem-Evo emphasizes *implicit behavioral evidence*: user preferences are revealed through choices, constraints, repeated requests, contextual behavior, and decision patterns across realistic conversations.

Built on top of PersonaMem-v2 (Jiang et al., 2025), PersonaMem-Evo extends static preference memory with structured temporal preference evolution and preference inference. For each user persona u_i, we construct a long-context interaction history H_i and a set of multiple-choice questions

$$Q_i^{\text{ood}} = \{q_{i1}^{\text{ood}}, \ldots, q_{im_i}^{\text{ood}}\},$$

where answering each question requires abstracting user-specific behavioral patterns from H_i and applying them to a novel decision setting.

We summarize the construction pipeline in Algorithm 3. The pipeline consists of seed persona sampling, persona expansion, preference synthesis, implicit interaction generation, temporal preference evolution, question generation, dual-blind filtering, and benchmark assembly. Algorithm 4 describes the construction of multi-step preference trajectories, and Algorithm 5 describes question generation with dual-blind validation.

D.7.1 Seed Persona Expansion and Preference Cleaning

Following PersonaMem-v2, we sample seed personas from PersonaHub (Ge et al., 2024), specifically the `Persona_Hub_200000.jsonl` subset. PersonaHub provides a large-scale repository of synthetic personas, which allows the benchmark to cover diverse demographic, occupational, lifestyle, and personality backgrounds. Each seed persona $u_i^{(0)}$ is expanded into a richer structured profile

$$u_i = \mathcal{E}\left(u_i^{(0)}\right),$$

where $\mathcal{E}(\cdot)$ denotes the persona expansion operator. The expanded profile contains demographic attributes, occupation, personality traits, lifestyle information, social background, and personal context. These fields provide diverse grounding for generating realistic behavioral preferences while keeping the benchmark synthetic and privacy-safe.

From the expanded persona u_i, we synthesize a preference set

$$P_i = P_i^{\text{st}} \cup P_i^{\text{anti}} \cup P_i^{\text{neu}} \cup P_i^{\text{th}} \cup P_i^{\text{med}},$$

where P_i^{st}, P_i^{anti}, and P_i^{neu} denote stereotypical, anti-stereotypical, and neutral preferences, respectively, while P_i^{th} and P_i^{med} denote therapy-related and health/medical preferences. Anti-stereotypical preferences are especially important because they reduce the reliability of demographic priors and encourage models to rely on behavioral evidence from the interaction history.

Generated preferences are post-processed using validation and deduplication routines. We remove empty preferences, duplicated entries, and semantically inconsistent preferences across categories. Each preference is also assigned a coarse topical label $z \in \mathcal{Z}$, which supports semantic coverage analysis and downstream stratification.

D.7.2 Implicit Interaction History Construction

Given the preference set P_i, we generate a collection of interaction episodes

$$\mathcal{C}_i = \{c_{i1}, \ldots, c_{in_i}\}.$$

Each episode c is a multi-turn dialogue

$$c = \left[(r_1, x_1), (r_2, x_2), \ldots, (r_T, x_T)\right],$$

where $r_t \in \{\texttt{user}, \texttt{assistant}\}$ denotes the speaker role and x_t denotes the utterance. For each preference $p \in P_i$, the generator creates one or more episodes in which p is expressed indirectly through user behavior. Interaction scenarios include daily-life conversations, email-style communication, writing assistance, professional communication, recommendation requests, repeated knowledge queries, troubleshooting, translation, and multimodal image-grounded conversations.

To avoid reducing the benchmark to explicit preference lookup, the generator is instructed to express the target preference implicitly through choices, constraints, tone, repeated requests, or contextual

Algorithm 3 Construction of PersonaMem-Evo

Require: Seed personas $\mathcal{U}^{(0)} = \{u_1^{(0)}, \ldots, u_N^{(0)}\}$; LLM generator $\mathcal{G}$; validator $\mathcal{V}$
Ensure: Benchmark set $\mathcal{B}^{\text{ood}}$

$\mathcal{B}^{\text{ood}} \leftarrow \emptyset$
for each seed persona $u_i^{(0)} \in \mathcal{U}^{(0)}$ **do**
  $u_i \leftarrow \textsc{ExpandPersona}(u_i^{(0)}; \mathcal{G})$
  $P_i \leftarrow \textsc{GeneratePreferences}(u_i; \mathcal{G})$
  $P_i \leftarrow \textsc{CleanAndDeduplicate}(P_i; \mathcal{V})$
  $P_i \leftarrow \textsc{AssignTopics}(P_i; \mathcal{G})$
  $\mathcal{C}_i \leftarrow \emptyset$
  for each preference $p \in P_i$ **do**
    $c \leftarrow \textsc{GenerateImplicitConversation}(u_i, p; \mathcal{G})$
    $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{c\}$
    if $\textsc{EligibleForEvolution}(p)$ **then**
      $(\phi, \mathcal{C}^{\text{evo}}) \leftarrow \textsc{SamplePreferenceEvolution}(u_i, p; \mathcal{G})$
      $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \mathcal{C}^{\text{evo}}$
    end if
  end for
  $H_i \leftarrow \textsc{BuildLongContextHistory}(\mathcal{C}_i)$
  $Q_i^{\text{ood}} \leftarrow \textsc{GenerateOODQuestions}(u_i, H_i; \mathcal{G}, \mathcal{V})$
  for each validated question $q \in Q_i^{\text{ood}}$ **do**
    $\mathcal{B}^{\text{ood}} \leftarrow \mathcal{B}^{\text{ood}} \cup \{(H_i, q)\}$
  end for
end for
return $\mathcal{B}^{\text{ood}}$

behavior, without stating it as an isolated declarative fact. With some probability, an episode is framed as describing another person's preference, yielding a label

$$\texttt{who} \in \{\texttt{self}, \texttt{others}\}.$$

This distinction tests whether models can separate user-specific memories from information about other entities.

Some short snippets are expanded into longer multi-turn interactions to improve conversational realism. In addition, a subset of episodes is followed by explicit forgetting requests, where the user asks the assistant not to retain or use a previously mentioned preference. These cases test whether memory systems can handle user-controlled preference removal or suppression.

The generated episodes are assembled into a persona-level long-context history

$$H_i = \Pi(\mathcal{C}_i),$$

where $\Pi(\cdot)$ denotes the history construction operator. In practice, Π preserves episode boundaries while concatenating episodes into a single mixed-topic interaction stream. The resulting histories are rendered under 32k and 128k context settings.

D.8 PersonaMem-Evo: Preference Evolution and Benchmark Construction

D.8.1 Temporal Preference Evolution

A central property of PersonaMem-Evo is that user preferences are not assumed to be static. For a large subset of self-related preferences, we sample a structured evolution plan

$$\phi = (f, k, \tau),$$

where $f \in \mathcal{F}$ is a change family, k is the number of sequential updates, and

$$\tau = \left(p^{(0)}, p^{(1)}, \ldots, p^{(k)}\right)$$

Change family	Ratio	Transition pattern	Example
Same-object attitude revision	30%	The attitude toward the same object is revised due to new context or experience.	Prefers intense hiking → avoids intense hiking after an injury → prefers light trail walks after recovery.
Object replacement	30%	The preferred object changes within the same semantic category.	Espresso → pour-over coffee → matcha.
Conditional preference shift	13%	A general preference becomes conditional on a situational constraint.	Likes reading → reads only when alone → reads mostly on weekends.
Attribute shift	13%	The preferred attribute changes while the broader category remains related.	Prefers red cars → prefers blue cars → prefers SUVs.
Temporal-validity shift	14%	The preference becomes valid only under a time span, routine, or recency condition.	Exercises daily → exercises only in summer → exercises in the morning.

Table 16: Five change families used to construct multi-step preference trajectories.

Algorithm 4 Sampling Multi-Step Preference Evolution

Require: Persona u_i; initial preference p; generator $\mathcal{G}$
Ensure: Change plan $\phi = (f, k, \tau)$; trajectory conversations $\mathcal{C}^{\text{evo}}$

Sample change family $f \sim \text{Cat}(\mathcal{F})$
Sample number of updates $k \in \{1, \ldots, 5\}$
Initialize trajectory $\tau \leftarrow (p^{(0)})$, where $p^{(0)} = p$
$\mathcal{C}^{\text{evo}} \leftarrow \emptyset$
for $s = 1$ to k **do**
    $(p^{(s)}, \eta^{(s)}) \leftarrow \textsc{ApplyChangeFamily}(p^{(s-1)}, f, u_i; \mathcal{G})$
    $c^{(s)} \leftarrow \textsc{GenerateImplicitConversation}(u_i, p^{(s)}, \eta^{(s)}; \mathcal{G})$
    Annotate $c^{(s)}$ with $(f, k, s, p^{(s-1)}, p^{(s)}, \eta^{(s)})$
    $\tau \leftarrow \tau \oplus p^{(s)}$
    $\mathcal{C}^{\text{evo}} \leftarrow \mathcal{C}^{\text{evo}} \cup \{c^{(s)}\}$
end for
$\phi \leftarrow (f, k, \tau)$
return $\phi, \mathcal{C}^{\text{evo}}$

is the resulting preference trajectory. Here, $p^{(0)}$ is the initial preference, and each updated state $p^{(t)}$ is generated conditioned on the previous state $p^{(t-1)}$ and a family-specific transition rule.

We sample $k \in \{1, \ldots, 5\}$ updates for each evolved preference chain. Approximately 85% of eligible self-related preferences receive a structured CHANGEPLAN. The goal is not to create arbitrary binary reversals, but to model natural preference evolution caused by new experiences, changing constraints, updated routines, contextual conditions, or temporal validity.

For each updated preference state $p^{(t)}$, we synthesize a new conversational episode that implicitly reflects the updated state. We also store metadata including the change family f, total trajectory length k, current step t, previous preference state $p^{(t-1)}$, updated preference state $p^{(t)}$, and the full trajectory τ. Thus, the final history H_i contains not only stable user traits but also temporally grounded evidence of how preferences evolve over time. This design supports evaluation of current-state recovery, outdated-state disambiguation, and reasoning over the direction of preference change.

D.8.2 PREFERENCE-INFERENCE QUESTION GENERATION

PersonaMem-Evo includes an preference-inference stage to test whether a model can generalize from observed behavioral patterns to new decision settings. These questions cannot be answered from the persona summary alone or from generic commonsense knowledge; instead, they require abstraction over the long interaction history.

Algorithm 5 Question Generation with Dual-Blind Filtering

Require: Persona u_i; long-context history H_i; generator $\mathcal{G}$; validator $\mathcal{V}$
Ensure: Validated question set Q_i^{ood}

$Q_i^{\text{ood}} \leftarrow \emptyset$
Define types $\mathcal{T}$ and complexity levels $\mathcal{D}$
for each question type $t \in \mathcal{T}$ **do**
  for each complexity level $d \in \mathcal{D}$ **do**
    for $r = 1$ to TARGETCOUNT(t, d) **do**
      $q \leftarrow$ DRAFTOODQUESTION$(u_i, H_i, t, d; \mathcal{G})$
      if $q = \varnothing$ **then**
        continue
      end if
      $q \leftarrow$ BUILDANSWEROPTIONS$(q; \mathcal{G})$
      if $\neg$ BALANCEDOPTIONS(q) **then**
        continue
      end if
      if PERSONABLINDCORRECT$(q, u_i; \mathcal{V})$ **then**
        continue
      end if
      if NOCONTEXTCORRECT$(q; \mathcal{V})$ **then**
        $q \leftarrow$ ADVERSARIALREWRITE$(q; \mathcal{G})$
        if $q = \varnothing$ **or** NOCONTEXTCORRECT$(q; \mathcal{V})$ **then**
          continue
        end if
      end if
      $Q_i^{\text{ood}} \leftarrow Q_i^{\text{ood}} \cup \{q\}$
    end for
  end for
end for
return Q_i^{ood}

Each question is represented as

$$q = (x^q, a^+, \mathcal{A}^-, t, d),$$

where x^q is the query, a^+ is the correct answer, $\mathcal{A}^-$ is a set of distractors, t is the question type, and d is the complexity level. We consider four types:

$$\mathcal{T} = \left\{ \begin{array}{l} \texttt{single-pattern-transfer}, \\ \texttt{multi-pattern-synthesis}, \\ \texttt{conflict-resolution}, \\ \texttt{temporal-trajectory} \end{array} \right\}, \qquad \mathcal{D} = \{\text{L1}, \text{L2}, \text{L3}\}.$$

The four types test complementary reasoning abilities:

- **Single-pattern transfer**: transfer one uncommon or counter-stereotypical preference pattern to a new domain.
- **Multi-pattern synthesis**: combine multiple observed preferences, including at least one counter-stereotypical pattern, into a narrower inference.
- **Conflict resolution**: infer the user's implicit priority structure when multiple preferences conflict.
- **Temporal trajectory prediction**: infer the directional evolution of a user's preferences and extrapolate that trajectory to a new domain.

The three complexity levels capture different forms of generalization. L1 corresponds to direct transfer, L2 requires cross-domain linking, and L3 requires more abstract meta-reasoning over multiple behavioral signals. These levels define qualitatively different reasoning categories, with increasing abstraction across levels.

D.8.3 DUAL-BLIND FILTERING

To ensure that accepted questions require conversational memory, we apply a dual-blind validation procedure that filters shortcut-solvable cases. Given a candidate question q, we define two validators:

$$V_{\text{pers}}(q, u_i) \in \{0, 1\}, \qquad V_{\emptyset}(q) \in \{0, 1\}.$$

Here, $V_{\text{pers}}(q, u_i) = 1$ indicates that the question can be answered correctly using only the persona profile u_i without the conversation history, while $V_{\emptyset}(q) = 1$ indicates that the question can be answered correctly with neither the persona profile nor the history. We accept a question only if

$$V_{\text{pers}}(q, u_i) = 0 \quad \text{and} \quad V_{\emptyset}(q) = 0.$$

The first constraint filters out items answerable from demographic priors, explicit persona facts, or stereotypes. The second constraint filters out items answerable from commonsense reasoning, answer-option artifacts, or stylistic imbalance.

When a candidate fails the no-context validation but is otherwise useful, we apply adversarial option rewriting. The rewrite makes the correct answer less guessable from generic priors and makes distractors more plausible, while preserving the intended behavioral inference. The rewritten item is accepted only if it passes both blind checks.

D.8.4 ANSWER-OPTION CONSTRUCTION

Each accepted question is packaged as a four-way multiple-choice item:

$$\mathcal{A}(q) = \{a^+\} \cup \mathcal{A}^-, \qquad |\mathcal{A}(q)| = 4.$$

The generator enforces answer-option balance so that no candidate is trivially identifiable by length, specificity, formatting, or vagueness. Distractors are designed to be plausible and semantically close to the correct answer. For questions involving temporal evolution, at least one distractor is often selected from an earlier preference state, making it necessary to identify the currently valid preference while accounting for historically valid alternatives. For anti-stereotypical preferences, at least one distractor is designed to be stereotype-consistent, creating a deliberate trap for models that rely on demographic priors without grounding their answers in behavioral evidence.

D.8.5 BENCHMARK ASSEMBLY AND METADATA

Finally, each validated question q_{ij}^{ood} is paired with the corresponding long-context history H_i to form a benchmark instance

$$b_{ij}^{\text{ood}} = \left(H_i, x_{ij}^q, a_{ij}^+, \mathcal{A}_{ij}^-, t_{ij}, d_{ij}, m_{ij}\right),$$

where m_{ij} denotes metadata associated with the instance. The metadata includes the source persona, source preference, topic label, scenario type, whether the preference belongs to the user or another person, whether the preference is updated, the associated conversation snippet, and, when applicable, the change family, trajectory length, trajectory step, full change plan, type, and complexity level.

By construction, PersonaMem-Evo isolates a setting in which success depends on remembering, disambiguating, and generalizing user-specific behavioral evidence, while filtering cases that can be solved through explicit fact lookup or demographic shortcuts.

D.8.6 INTERACTION FORMATS AND PREFERENCE RECORDS

For each generated preference, we synthesize one or more short dialogue segments between a simulated user and an AI assistant. Each segment follows a predefined interaction format, such as email revision, translation, knowledge-seeking, recommendation, troubleshooting, or consultation-style advice. The format controls the surface form of the exchange, while the target preference remains implicit: it is expressed through the user's constraints, wording, examples, repeated requests, or surrounding context, without using a direct statement such as "I like X."

Each segment is paired with a structured *preference record*, which specifies the latent preference expressed by the dialogue and its relation to other records. A record contains the preference text, source category, owner, temporal status, and the generated conversation. The source category indicates

where the preference originates in the generation pipeline, e.g., stereotypical, anti-stereotypical, neutral, health-related, therapy-related, multimodal, or OOD-inference. The owner field indicates whether the preference describes the simulated user (`self`) or another person mentioned by the user (`others`). The temporal fields specify whether the preference is static, updates a previous preference, or corresponds to an explicit forgetting request.

Concretely, each record contains fields such as

`preference`, `pref_type`, `who`, `updated`, `prev_pref`, `conversations`.

For preference-evolution cases, we additionally store trajectory-level metadata:

`change_family`, `change_k`, `change_step`, `change_plan`.

Here, `conversations` contains the actual dialogue turns, while the remaining fields specify the latent preference and its temporal relation to other preference records. During benchmark assembly, these dialogue segments are concatenated into mixed-topic long-context histories while preserving their boundaries and metadata for question generation and analysis.

D.8.7 DATASET STATISTICS

We report diagnostic statistics for the 10-persona PERSONAMEM-EVO evaluation subset used in our analysis. This subset contains 505 preference-inference questions from 10 personas, with 49–51 questions per persona. The question types are approximately balanced across single-pattern transfer, multi-pattern synthesis, conflict resolution, and temporal trajectory prediction. The subset also includes a broad difficulty distribution, with 120 L1, 186 L2, and 199 L3 questions.

Statistic	**Count**	**Mean / Median**	**Range**
Personas	10	–	–
Questions	505	50.5 / 51 per persona	49–51
Single-pattern transfer	130	–	–
Multi-pattern synthesis	129	–	–
Conflict resolution	117	–	–
Temporal trajectory prediction	129	–	–
L1 questions	120	–	–
L2 questions	186	–	–
L3 questions	199	–	–

Table 17: Scale, question-type, and difficulty statistics for the 10-persona PERSONAMEM-EVO subset.

We further characterize how much preference evidence each question requires. A *source preference* is a gold evidence preference listed in `source_preferences`; it corresponds to an observed user preference that should be used to answer the question. This is a mention-level count, so a single question may require multiple source-preference mentions. In total, the 505 questions contain 1,553 source-preference mentions. As shown in Table 18, single-pattern transfer questions require one source preference, conflict-resolution questions require two, and multi-pattern synthesis questions require three. Temporal trajectory questions are the most evidence-intensive, requiring 2–10 source preferences with median 6, making them a strong test of temporal memory and multi-evidence composition.

Question Type	**#Questions**	**Min**	**Median**	**Mean**	**Max**
Single-pattern transfer	130	1	1	1.0	1
Conflict resolution	117	2	2	2.0	2
Multi-pattern synthesis	129	3	3	3.0	3
Temporal trajectory prediction	129	2	6	6.2	10

Table 18: Number of source preferences required per question.

The source-preference composition is designed to support fine-grained diagnosis of memory-agent failures. Among the 1,553 source-preference mentions, the subset contains comparable numbers of

stereotypical and anti-stereotypical preferences, which helps test whether agents rely on observed user behavior instead of persona-level priors. We also distinguish static preferences from changed preferences. Among source preferences matched to raw metadata, 801 mentions correspond to changed preferences and 664 correspond to static preferences. Changed preferences are further divided into object replacement, same-object attitude revision, conditional preference shift, temporal-validity shift, and attribute shift, allowing us to analyze which kinds of preference updates are most challenging.

Source Category	Mentions
Stereotypical preference	574
Anti-stereotypical preference	564
Neutral preference	223
Health / medical preference	35
Therapy-background preference	1
Other matched source mention	68
Unmatched exact string	88
Total source mentions	1,553
Matched to raw metadata	1,465
Matched to observed dialogue	1,397

(a) Source-preference composition.

Preference Type	Mentions
Static preference	664
Object replacement	273
Same-object attitude revision	177
Conditional preference shift	179
Temporal-validity shift	111
Attribute shift	61

(b) Static and changed preferences.

Table 19: Source-preference composition and change-family statistics. Counts are mention-level.

Finally, we characterize the length of the generated interaction histories. Each benchmark question is answered against the full persona-level chat history instead of a short declarative profile. As shown in Table 20, histories are long: the median persona has 597 messages and 174.7K tokens. This makes PERSONAMEM-EVO a challenging setting for memory agents, since relevant implicit preference evidence must be extracted from long, mixed-topic conversations.

Chat-History Statistic	Min	Median	Mean	Max
Messages per persona	113	597.0	634.7	1,077
User messages per persona	57	308.0	325.6	551
Assistant messages per persona	55	288.5	308.1	525
Words per persona	25,177	134,552	136,110	267,777
Tokens per persona	31,719	174,684	178,292	367,935

Table 20: Generated chat-history length statistics for the 10-persona subset.

Together, these statistics show that PERSONAMEM-EVO tests more than factual recall. Source-preference counts measure evidence aggregation, source composition tests resistance to stereotype shortcuts, change-family statistics test preference updating, and long histories impose a realistic memory-retrieval burden.

E EVOMEM IMPLEMENTATION DETAILS

E.1 TERMINUS2 WITH EVOMEM

Overview. We implement EvoMem for Terminus2 as a chain-scoped patch memory layer for terminal-native task solving. This chain memory instantiates M_T for Terminus2: it is synthesized from prior terminal trajectories and supplied as an external memory context. Terminus2 keeps its original terminal interaction loop: it observes the task, issues shell actions, reads terminal feedback, and decides when to finish. EvoMem augments the context that Terminus2 receives before a task and updates a persistent chain memory after a task has produced a terminal trajectory.

The implementation is designed for sequential task chains. Earlier tasks in the same chain provide internal evidence for synthesizing reusable procedural memory, while later tasks often introduce small but consequential changes in input location, output contract, environment, tool availability,

or artifact format. EvoMem records these shifts as compact transition patches that pair a strategy-level adaptation with the task or environment condition that caused it. Importantly, EvoMem does not expose raw previous trajectories, verifier outcomes, reference patches, or task-specific solution artifacts to later tasks. Prior executions are used only to synthesize compact, sanitized procedural summaries and transition-level adaptation hints. At the next task, EvoMem retrieves only the chain memory that is applicable to the current instruction and environment, so the agent receives targeted guidance about how the solution strategy should adapt.

Chain-scoped terminal memory. Let

$$\mathcal{X} = \{x_1, x_2, \ldots, x_T\}$$

denote a sequence of related terminal tasks in one chain. After processing the first t tasks, EvoMem maintains two chain-level stores:

$$\mathcal{L}_t = \{\ell_1, \ell_2, \ldots, \ell_t\}, \qquad \Pi_t = \{\pi_2, \pi_3, \ldots, \pi_t\}.$$

Here ℓ_i is a compact execution ledger for task x_i, and π_i is a transition patch that summarizes how the strategy changed from x_{i-1} to x_i. The ledger records abstract reusable procedure information from a prior terminal solution: the task goal, a short strategy summary, relevant artifact types, high-level command patterns, ineffective strategy notes, and output requirements. The transition patch records the non-additive change across neighboring tasks: what requirement or environment condition changed, which previous assumption was superseded, what adaptation was observed, and which concrete values from the previous task should not be copied blindly.

This representation separates a base recipe from revision evidence. A base ledger gives the agent an initial solving template for the chain, while patches explain how that template should be revised when the current task differs from previous ones. This is especially important for evolving terminal tasks, where the high-level workflow may remain stable but a path, input convention, output format, or toolchain constraint changes.

Store-time update. After Terminus2 finishes a task x_t, EvoMem extracts a structured terminal history h_t from the agent trajectory. This history contains the agent's messages, shell commands, and terminal observations in a normalized event format. EvoMem also collects a lightweight environment snapshot e_t, including available command-line tools and a compact sketch of the task workspace. The environment snapshot is best-effort and is used only to characterize applicability; failure to obtain it does not change the underlying Terminus2 loop.

For $t > 1$, EvoMem compares the current instruction with the previous task instruction:

$$\delta_t = \mathrm{Diff}(x_{t-1}, x_t).$$

The update summarizer then receives the current instruction, the previous instruction, the instruction delta, the terminal history, the environment snapshot, and bounded evidence about changed artifacts. It produces a ledger and, when a previous task exists, a transition patch:

$$(\ell_t, \pi_t) = U(x_t, x_{t-1}, \delta_t, h_t, e_t).$$

The ledger ℓ_t captures abstract reusable execution knowledge from the current task, while removing task-specific answer artifacts. The patch π_t captures how the current task revises the previous chain behavior and why that revision was needed under the new task or environment conditions. If the current task is the first task in a chain, EvoMem writes only the ledger and treats it as the initial chain recipe.

Patch representation. Each transition patch is represented as

$$\pi_t = (c_t, a_t, o_t, g_t, d_t, r_t)\,,$$

where c_t denotes the type of task or environment change, a_t is the superseded assumption, o_t is the observed adaptation, g_t is a generalized pattern for future tasks, d_t lists concrete prior details that should not be copied unless the current instruction requires them, and r_t records the concise cause of the revision. The implementation infers c_t from instruction differences, environment signals, and execution evidence, without using task-specific hand labels. Examples include changes in

input source, output destination, formatting contract, working directory convention, data layout, and toolchain or environment requirements.

The patch is intentionally written as a transition example for adaptation across related tasks. This distinction matters because Terminus2 operates in a live terminal environment: a previous strategy may be useful as evidence, but the current instruction remains authoritative. EvoMem therefore preserves only abstract command patterns when they are useful as reusable procedural guidance, and pairs them with patch fields that describe which parts must be re-derived from the current task.

Retrieval-time construction. Before solving a new task x_{t+1}, EvoMem first checks whether the chain contains reusable memory. If no ledger or relevant patch exists, the original Terminus2 instruction is passed through unchanged. This avoids prompting the agent to search for memory when no useful memory has been created.

When chain memory exists, EvoMem builds a retrieval query from the current task instruction, the change summary against the previous task, and the current environment snapshot:

$$q_{t+1} = Q(x_{t+1}, \delta_{t+1}, e_{t+1}).$$

The retriever selects a base ledger and a small set of relevant transition patches:

$$C_{t+1} = \mathrm{Retrieve}(q_{t+1}, \mathcal{L}_t, \Pi_t).$$

Patch matching combines two signals. First, EvoMem compares the current task-contract change with the change types summarized in previous patches, so that output-format changes retrieve relevant output-format adaptations. Second, it compares task text, environment signals, and salient paths or artifacts at a lexical level. Retrieved patches are then ordered according to their position in the task chain, preserving the temporal structure of the evolving task sequence.

Prompt construction and container hydration. Terminus2 can be sensitive to long prompt prefixes and transcript-like memory. We therefore do not paste the full chain memory directly into the main task instruction. Instead, EvoMem renders the selected context into a compact memory document and materializes it inside the task container before Terminus2 starts solving. The prompt receives only a short memory reference telling the agent that chain memory is available and that the current task instruction is authoritative:

$$P(x_{t+1}) = \mathrm{Prompt}(x_{t+1}, \mathrm{Ref}(C_{t+1}))\,.$$

If container hydration is unavailable, EvoMem falls back to an inline compact rendering under the same length budget.

The rendered memory contains no raw terminal transcript or previous task answer. It contains three sanitized components. The first summarizes current differences from the previous task. The second provides a base execution ledger, including abstract recipe steps or high-level command patterns when available. The third lists relevant transition examples, each phrased as a conditional adaptation pattern. This format gives Terminus2 enough information to adapt prior experience while avoiding a large, noisy replay of earlier trajectories.

Safety and anti-copying safeguards. EvoMem includes safeguards to prevent stale memory from overriding the current task. First, the prompt explicitly states that retrieved memory is guidance and that the current instruction is authoritative. Second, transition patches distinguish generalized adaptation patterns from concrete old values. Prior output fragments, answer literals, and task-specific prefixes are redacted or abstracted before they are exposed to the agent. Third, patches include "do-not-copy" evidence, which marks old paths, values, or output fragments as examples that should be reused only if the current instruction independently calls for them.

These safeguards are important for terminal tasks because a memory item can be partially correct but obsolete in a later environment. EvoMem is therefore not a replay buffer. It is a patch-conditioned context mechanism: the agent is expected to use prior execution as evidence, infer the current task's own contract, and adapt the solution accordingly.

Algorithm 6 EvoMem: Chain-Scoped Patch Memory for Terminus2

Require: Task chain $\mathcal{X} = \{x_1, \ldots, x_T\}$; Terminus2 solver M; chain ledgers $\mathcal{L}$; transition patches Π
Ensure: Completed terminal trajectories and updated chain memory
Initialize $\mathcal{L} \leftarrow \emptyset$, $\Pi \leftarrow \emptyset$
for $t = 1$ to T **do**
  Obtain environment snapshot e_t
  if $\mathcal{L}$ or Π contains reusable chain memory **then**
    Compute instruction delta δ_t against the previous chain task
    Build query $q_t \leftarrow Q(x_t, \delta_t, e_t)$
    Retrieve base ledger and transition patches $C_t \leftarrow \mathrm{Retrieve}(q_t, \mathcal{L}, \Pi)$
    Materialize compact sanitized memory context for the task container
    Compose prompt $P_t \leftarrow \mathrm{Prompt}(x_t, \mathrm{Ref}(C_t))$
  else
    Set $P_t \leftarrow \mathrm{Prompt}(x_t)$
  end if
  Run Terminus2: $(y_t, h_t) \leftarrow M(P_t)$
  Extract normalized terminal history h_t
  Synthesize sanitized ledger $\ell_t \leftarrow U_{\mathrm{ledger}}(x_t, h_t, e_t)$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{\ell_t\}$
  if $t > 1$ **then**
    Compute transition patch $\pi_t \leftarrow U_{\mathrm{patch}}(x_{t-1}, x_t, \delta_t, h_t, e_t)$
    $\Pi \leftarrow \Pi \cup \{\pi_t\}$
  end if
end for

Relation to Terminus2. The integration is intentionally non-invasive. Terminus2 remains responsible for planning, terminal command generation, observation handling, and completion decisions. EvoMem adds only a persistent chain memory store, a retrieval-and-rendering step before execution, and a post-run summarization step after the terminal trajectory is available:

$$\text{Terminus2+EvoMem} = \text{Terminus2} + \text{Chain Memory} + \text{Transition Patch Retrieval}.$$

This preserves Terminus2's terminal-native behavior while allowing it to reuse abstract procedural experience across evolving tasks without exposing raw prior task executions or blindly repeating procedures that were valid only for earlier chain members.

E.2 OpenHands with EvoMem

Overview. We implement EvoMem for OpenHands as an external patch-memory layer for sequential software-engineering tasks. OpenHands preserves its original autonomous coding loop: it reads the task, inspects the repository, edits code, runs commands, and produces a patch. EvoMem does not modify the OpenHands agent internals. Instead, it wraps the task interface: before each task, it retrieves compact historical code context and prepends it to the task description; after the task, it distills the resulting trajectory and patch into feature-level memory records.

For OpenHands, M_T is instantiated as a distilled code-context memory:

$$M_T = \{\rho_1, \rho_2, \ldots, \rho_T\},$$

where each record ρ_i summarizes a prior implementation change. The purpose is not to replay old patches verbatim, but to provide historical debugging context: which files and symbols were involved, what behavior changed, why the change was needed, which constraints should be preserved, and what code evidence supports the summary. When a later task supersedes an earlier implementation assumption, the corresponding record captures the old behavior, the revised logic, and the concrete behavioral cause of the revision.

Feature-level patch records. Given a task x_t, OpenHands produces a trajectory h_t and a code patch d_t. EvoMem first groups the patch into feature-level units using changed files, code hunks, and

execution evidence around file reads, code edits, tests, and command observations. For each feature group g, a summarizer constructs a patch record

$$\rho = \left(f,\ s,\ b^-,\ b^+,\ c,\ r,\ \Delta\right),$$

where f denotes affected files, s denotes relevant symbols or feature tags, b^- and b^+ summarize behavior before and after the change, c lists constraints to preserve, r gives the concrete reason for the revision, and Δ stores bounded code evidence. The record is intentionally compact: it preserves the information needed for future debugging while avoiding long trajectory replay.

This representation matches the software-engineering setting. A prior task may have modified a helper, parser, API boundary, or configuration path in a way that remains relevant to later tasks. If a later instruction touches overlapping code, the retrieved record reminds the agent of the previous behavioral intent and reduces the risk of undoing earlier logic.

Retrieval. At inference time, EvoMem builds a query from the current task description. The query includes visible file hints, natural-language feature terms, and symbol-like identifiers:

$$q_t = Q(x_t).$$

For each prior record ρ_i, EvoMem estimates relevance using both semantic and structural evidence. The semantic component matches the task query against the record's feature title, tags, symbols, behavioral summary, and code-context text. The structural component compares file-path segments and exact file overlap:

$$s(q_t, \rho_i) = \alpha\, s_{\text{text}}(q_t, \rho_i) + \beta\, s_{\text{path}}(q_t, \rho_i) + \gamma\, s_{\text{exact}}(q_t, \rho_i).$$

The top-ranked records are rendered under a context budget. If no useful prior memory exists, EvoMem leaves the OpenHands task prompt unchanged.

Prompt construction. The selected records are rendered as a short patch-memory block and prepended to the original task description:

$$P(x_t) = \text{Prompt}(x_t, \{\rho_{i_1}, \ldots, \rho_{i_k}\}).$$

Each rendered record contains the affected files, the revision intent, constraints that should be preserved, and a bounded code excerpt when available. The prompt explicitly states that the current task remains authoritative and that prior patches should be used only as contextual evidence. Thus, EvoMem supplies historical debugging memory without changing the OpenHands planner, editor, tool interface, or repository interaction loop.

Algorithm 7 EvoMem: Patch Memory for OpenHands

Require: Sequential tasks $\mathcal{X} = \{x_1, \ldots, x_T\}$; OpenHands solver $\mathcal{A}$; memory $M_0 = \emptyset$
Ensure: Updated patch memory M_T

for $t = 1$ to T **do**
  Set $M_t \leftarrow M_{t-1}$
  Build query $q_t \leftarrow Q(x_t)$ from visible files, feature terms, and symbols
  Retrieve relevant records $R_t \leftarrow \text{Retrieve}(q_t, M_{t-1})$
  if R_t is nonempty **then**
    Compose prompt $P_t \leftarrow \text{Prompt}(x_t, R_t)$
  else
    Set $P_t \leftarrow \text{Prompt}(x_t)$
  end if
  Run OpenHands: $(d_t, h_t) \leftarrow \mathcal{A}(P_t)$
  Segment d_t into feature-level groups $\mathcal{G}_t$
  for each group $g \in \mathcal{G}_t$ **do**
    Summarize (g, d_t, h_t) into patch record ρ_g
    $M_t \leftarrow M_t \cup \{\rho_g\}$
  end for
end for

Relation to OpenHands. EvoMem is a lightweight augmentation to OpenHands:

OpenHands+EvoMem = OpenHands + Patch Record Store + Semantic-Structural Retrieval.

The base agent remains responsible for repository exploration, code editing, command execution, and patch generation. EvoMem only determines which historical code-context records should be shown before the next task and how the newly produced patch should be summarized for future reuse.

E.3 MEMENTO-SKILL WITH EVOMEM

Overview. We implement EvoMem as a lightweight memory layer on top of the existing Memento-Skill task-solving loop. The key design choice is to keep the base skill router and skill execution API unchanged, while augmenting only the learning-tip subsystem. Concretely, Memento-Skill continues to select and execute skills in its original workflow. EvoMem intervenes at two points: before solving a task, it retrieves historically relevant tip versions and injects them into the prompt; after a task failure yields reusable feedback, it writes a new versioned tip patch into memory. This preserves the original Memento-Skill control flow while enabling continual accumulation and selective reuse of task-solving experience.

Versioned tip memory. In Memento-S, reusable experience is stored in a global `TIP.md` file. EvoMem replaces this single mutable file with a family of versioned tip memories. Let

$$\mathcal{F}_T = \{v_1, v_2, \ldots, v_T\}$$

denote the tip-version family after T updates. Each version is represented as

$$v_t = (\tau_t, m_t, p_t),$$

where τ_t is the full tip-text snapshot, m_t is retrieval-oriented metadata, and p_t is a parent pointer indicating lineage. In our implementation, the family is stored under the global identifier `global_learning_tips`; each version is serialized as a JSON file under `.memgit/families/tips/global_learning_tips/versions/`.

The metadata m_t records the evidence and rationale behind a version:

$$m_t = \left(r_t, \Delta_t, S_t^+, S_t^-, \Gamma_t^+, \Gamma_t^-, \Sigma_t\right).$$

Here, r_t denotes the update reasoning, Δ_t summarizes the difference from the previous version, S_t^+ and S_t^- denote compressed success and failure case summaries, Γ_t^+ and Γ_t^- summarize successful and failed trajectories, and Σ_t contains selection signals used as retrieval cues. Although we use the term "patch", the implementation stores full replacement snapshots rather than textual diffs. The patch semantics are captured by the lineage pointer p_t and the difference metadata Δ_t, while inference can directly read the full snapshot τ_t without replaying an edit chain.

Store-time update. EvoMem writes a new tip version only when Memento-Skill observes a reusable learning signal, typically after an incorrect attempt followed by judge feedback and execution-trace analysis. Let x denote the task, y the system output, and j the failure analysis produced from the judge and trace. Given the previous tip version $v_T = (\tau_T, m_T, p_T)$, EvoMem uses an LLM-based updater to synthesize a new tip snapshot and metadata:

$$(\tau_{T+1}, m_{T+1}) = U(x, j, \tau_T, m_T),$$

where $U(\cdot)$ is prompted to produce concise, task-specific guidance, organize it by task type, and generate retrieval-friendly selection signals.

The new version is written in two steps. First, EvoMem derives a child version from the current latest version:

$$v_{T+1} \leftarrow \textsc{Derive}(v_T),$$

which copies the parent lineage and initializes the child metadata. Second, EvoMem records the new tip snapshot and metadata:

$$v_{T+1} \leftarrow \textsc{Record}(v_{T+1}, \tau_{T+1}, m_{T+1}).$$

The family then evolves monotonically:

$$\mathcal{F}_{T+1} = \mathcal{F}_T \cup \{v_{T+1}\}.$$

This update rule maintains versioned tip histories, allowing different historical tips to remain available for different task regimes.

Inference-time retrieval. At inference time, EvoMem converts the incoming task into a retrieval query q using stable task descriptors already available in Memento-S, such as the question text, answer type, category, subject, difficulty, and attachment names:

$$q = Q(x).$$

For each tip version v_t, EvoMem constructs a retrieval document

$$d_t = \texttt{concat}(\tau_t, m_t),$$

which includes the tip text, update reasoning, difference summary, success and failure summaries, trajectory summaries, and selection signals. The first-stage retrieval score is BM25:

$$s_t^{\mathrm{bm25}} = \mathrm{BM25}(q, d_t).$$

Versions are ranked within the tip family, and the top-k candidates are selected. We use $k = 2$ by default.

The implementation also supports optional hybrid reranking. Let c_t be the TF-IDF cosine similarity between q and d_t, and let $\hat{s}_t^{\mathrm{bm25}}$ denote the BM25 score normalized by the maximum score among retrieved candidates. The hybrid score is

$$s_t = \lambda \hat{s}_t^{\mathrm{bm25}} + (1 - \lambda) c_t,$$

where $\lambda \in [0, 1]$ and the default value is $\lambda = 0.5$. A relative threshold ρ can further filter weak matches:

$$\frac{s_t}{\max_j s_j} \geq \rho,$$

with $\rho = 0$ by default. Unless otherwise stated, we use the default BM25 retrieval setting.

Prompt construction and execution. The selected versions are rendered into a prompt block containing the version identifier, selection signals, a truncated tip snapshot, representative success and failure summaries, and the update note. The final Memento-Skill prompt is

$$P(x) = \texttt{Prompt}(x, \{v_{t_1}, \ldots, v_{t_k}\}),$$

where the retrieved versions provide contextual guidance for the current task and do not override the task instruction. Memento-Skill then solves the task using its original solver:

$$(y, \mathrm{trace}) = M(P(x)).$$

If the task succeeds, the output is returned directly. If the task fails, EvoMem may create a new tip version using the store-time update rule above; when retry is enabled, the newly written version can be pinned into the next prompt.

Algorithm 8 EvoMem: Versioned Tip Patching on Top of Memento-S

Require: Task x; Memento-Skill solver M; tip family $\mathcal{F}_T = \{v_1, \ldots, v_T\}$; top-$k$; interpolation weight λ; threshold ρ
Ensure: Answer y
Build retrieval query $q \leftarrow Q(x)$
for each tip version $v_t \in \mathcal{F}_T$ **do**
  Build retrieval document $d_t \leftarrow \texttt{concat}(\tau_t, m_t)$
  Compute BM25 score $s_t^{\mathrm{bm25}} \leftarrow \mathrm{BM25}(q, d_t)$
end for
Select top-k candidates by s_t^{bm25}
if hybrid reranking is enabled **then**
  for each selected candidate v_t **do**
    Compute TF-IDF cosine score $c_t \leftarrow \cos(q, d_t)$
    Normalize BM25 score $\hat{s}_t^{\mathrm{bm25}}$
    $s_t \leftarrow \lambda \hat{s}_t^{\mathrm{bm25}} + (1-\lambda) c_t$
  end for
  Re-rank candidates by s_t
else
  Set $s_t \leftarrow s_t^{\mathrm{bm25}}$
end if
Retain candidates satisfying $s_t / \max_j s_j \geq \rho$
Compose prompt $P(x)$ by appending retained tip versions to the original Memento-Skill prompt
Run Memento-S: $(y, \mathrm{trace}) \leftarrow M(P(x))$
if y is correct **then**
  return y
else
  Obtain failure analysis j from judge feedback and execution trace
  $(\tau_{T+1}, m_{T+1}) \leftarrow U(x, j, \tau_T, m_T)$
  $v_{T+1} \leftarrow \textsc{Derive}(v_T)$
  $v_{T+1} \leftarrow \textsc{Record}(v_{T+1}, \tau_{T+1}, m_{T+1})$
  $\mathcal{F}_{T+1} \leftarrow \mathcal{F}_T \cup \{v_{T+1}\}$
  if retry is enabled **then**
    Compose retry prompt with v_{T+1} pinned
    $(y, \mathrm{trace}) \leftarrow M(P_{\mathrm{retry}}(x))$
  end if
  return y
end if

Relation to Memento-S. EvoMem is intentionally non-invasive. It does not replace the Memento-Skill skill selector, modify the skill execution API, or alter the base solving loop. Instead, it modifies only the prompt construction stage and the post-failure feedback stage. This separates capability routing from experience reuse: the Memento-Skill skill router determines which tool or skill family to use, while EvoMem determines which historical reasoning patches are most relevant for the current task.

E.4 A-MEM WITH EVOMEM

Overview. We implement EvoMem as a patch-augmented extension of A-Mem. A-Mem serves as the base memory system, maintaining a structured memory graph that stores the latest consolidated user information. EvoMem keeps this mechanism unchanged, but adds a patch layer that records non-additive memory revisions caused by later turns. This design targets a limitation of long-horizon personalization: after many updates, the current memory may preserve the latest belief but obscure how the belief changed, when it changed, and which earlier state was superseded. EvoMem therefore maintains both the current A-Mem graph and a temporally indexed history of memory edits.

Formally, let

$$G_t = (V_t, E_t)$$

denote the A-Mem memory graph after processing turn x_t. Each node $v \in V_t$ is a memory note containing textual content and structured attributes such as context, keywords, and tags; each edge $e \in E_t$ represents an association between notes. A-Mem updates this graph by inserting new information, revising existing notes, and updating note links. EvoMem augments this process with patch construction, patch indexing, and patch-conditioned retrieval. Algorithm 9 summarizes the full procedure.

Algorithm 9 EvoMem: Patch-Augmented A-Mem

Require: Conversation stream $\mathcal{X} = \{x_1, \ldots, x_T\}$; question q
Ensure: Answer $\hat{y}$

```
Initialize A-Mem graph G_0 ← ∅
Initialize patch memory Π ← ∅
for t = 1 to T do
    G_{t-1}^{old} ← G_{t-1}
    (G_t, τ_t) ← AMemUpdate(G_{t-1}, x_t)
    Δ_t ← GraphDiff(G_{t-1}^{old}, G_t)
    if Δ_t is non-additive and x_t is user-driven then
        π_t ← SummarizePatch(x_t, τ_t, Δ_t)
        Π ← Π ∪ {π_t}
        IndexPatch(π_t)
    end if
end for
z(q) ← KeywordRewrite(q)
C^cur ← RetrieveCurrent(G_T, z(q), k)
P(q) ← RetrievePatches(Π, z(q), m)
P(q) ← SortByTemporalOrder(P(q))
C^patch ← FormatPatches(P(q))
C(q) ← [C^cur; C^patch]
ŷ ← f_θ(q, C(q))
return ŷ
```

Patch construction during memory ingestion. For each incoming turn x_t, EvoMem first applies the original A-Mem update routine:

$$(G_t, \tau_t) = \mathcal{M}(G_{t-1}, x_t),$$

where $\mathcal{M}$ denotes the A-Mem insertion procedure and τ_t is the evolution trace produced during memory editing. EvoMem then compares the graph before and after the update:

$$\Delta_t = \mathrm{Diff}(G_{t-1}, G_t).$$

In our implementation, Δ_t tracks newly created notes, updated notes, changed fields, and modified links. Changed fields include note content, context, keywords, tags, and graph relations.

EvoMem creates a patch only for non-additive updates. Purely additive updates, where a new note is added without modifying existing memory, are already preserved in the current graph and do not require a patch. We therefore create patches only when the update changes existing memory semantics or graph structure. In practice, this includes two main cases: `overwrite_update`, where note content or metadata is revised, and `link_rewrite_update`, where the neighborhood of an existing note is rewired. We also suppress patches triggered by assistant turns, so that patches primarily reflect user-driven memory evolution and avoid capturing assistant-side paraphrases or generated explanations.

Patch representation. When a non-additive update is detected, EvoMem materializes a patch record

$$\pi_t = (\mathrm{id}_t, \mathrm{meta}_t, \Delta_t, S_t, R_t)\,,$$

where id_t is a unique patch identifier, meta_t stores turn-level metadata, Δ_t records the structured graph edit, S_t is an LLM-generated summary of the update, and R_t contains before/after evidence for affected notes. The metadata includes session index, turn index, timestamp, and speaker role.

More concretely, each patch stores the trigger turn, patch type, affected notes, note-level before/after states, edge rewrites, an update summary, an estimated preference domain, an inferred change type, and a temporal order key. For each changed note v, EvoMem records a local edit block

$$r_t(v) = (v, \mathcal{F}_t(v), \mathbf{b}_t(v), \mathbf{a}_t(v)),$$

where $\mathcal{F}_t(v)$ is the set of changed fields, and $\mathbf{b}_t(v)$ and $\mathbf{a}_t(v)$ denote the note state before and after the update. This exposes what was revised instead of retaining only the final consolidated note.

Patch indexing and storage. EvoMem stores the patch history separately from the A-Mem graph. For a sample i, let

$$\Pi_i = \{\pi_1, \pi_2, \ldots, \pi_{T_i}\}$$

denote its patch set. Each patch is converted into a compact retrieval document containing the temporal position, inferred preference domain, previous state, updated state, change type, and a short trigger snippet. We embed these patch documents with the same sentence encoder used by the retriever; in our implementation, we use `all-MiniLM-L6-v2`. The embedded patches are inserted into a separate patch retriever.

Thus, EvoMem maintains two retrieval spaces:

$$\mathcal{R}_{\text{mem}} : q \mapsto \text{current memory evidence from } G_T,$$
$$\mathcal{R}_{\text{patch}} : q \mapsto \text{historical revision evidence from } \Pi_i.$$

The base graph is optimized for retrieving the latest consolidated memory, while the patch store is optimized for retrieving historical changes that explain preference evolution, conflicts, or overwritten states.

Query-time retrieval. At inference time, EvoMem combines current graph evidence with relevant patches. Given a question q, we first rewrite it into a retrieval query

$$z(q) = \text{LLMKeyword}(q),$$

using a lightweight keyword-generation prompt. We then retrieve current memory evidence from A-Mem:

$$C^{\text{cur}}(q) = \mathcal{R}_{\text{mem}}(z(q), k),$$

where k is the number of retrieved notes. In our default evaluation configuration, we set $k = 10$.

In parallel, EvoMem retrieves the top-m relevant patches:

$$P(q) = \mathcal{R}_{\text{patch}}(z(q), m),$$

with default $m = 2$. By default, patch retrieval is embedding-based. The implementation also supports an optional hybrid scorer that interpolates dense similarity and BM25:

$$s_{\text{hyb}}(\pi, q) = \alpha s_{\text{dense}}(\pi, q) + (1 - \alpha) s_{\text{BM25}}(\pi, q),$$

where $\alpha = 0.7$ in the hybrid setting. Unless otherwise stated, we use the standard embedding-based patch retrieval and disable hybrid retrieval, patch-node reranking, and gating.

After retrieval, selected patches are sorted by temporal order. This enforces a simple precedence rule: when multiple patches refer to the same preference domain, later patches should be considered more recent evidence than earlier ones.

Patch-conditioned answer generation. EvoMem renders the selected patches into a textual evidence block and concatenates them with current memory evidence:

$$C(q) = \left[C^{\text{cur}}(q); C^{\text{patch}}(q)\right].$$

Each patch block includes the session and turn indices, patch type, temporal order, trigger utterance, update summary, inferred change pattern, and key before/after evidence. The final answer is generated by conditioning the answering model on the question and the augmented context:

$$\hat{y} = f_\theta(q, C(q)).$$

In our evaluation harness, EvoMem uses an always-on patch mode: whenever relevant patches are retrieved, they are injected into the answer context. The implementation also supports a gated variant in which the model first decides whether patch details are necessary, but we do not enable this option in the default experiments.

Relation to A-Mem. EvoMem is a lightweight augmentation of A-Mem. The original A-Mem system still performs note creation, note evolution, link updates, and graph-based retrieval. EvoMem adds three components:

$$\text{EvoMem} = \text{A-Mem} + \text{Diff Module} + \text{Patch Store} + \text{Patch Retrieval}.$$

This preserves A-Mem's ability to organize consolidated memory while adding an explicit representation of memory evolution. Consequently, EvoMem can answer questions that depend not only on what the user currently prefers, but also on how a preference changed, which earlier state was replaced, and which revision should be treated as the latest valid belief.

F Experiments Setting

In this section, we provide benchmark-specific implementation details. Since each benchmark requires different interaction protocols and agent capabilities, we organize the settings by dataset.

F.1 Terminal-Bench-Evo

For *Terminal-Bench-Evo*, we use Terminus2 as the baseline terminal agent. The baseline does not use persistent memory. For Terminus2 + EvoMem, we keep the same Terminus2 configuration and add chain-scoped patch memory. Tasks within the same evolution chain are run sequentially so that memory from earlier variants is available to later variants; different chains are independent and may be run in parallel. The rendered memory context is kept compact with a budget of 3500 characters. Both baseline and EvoMem use the same model configuration and agent execution budget.

F.2 SWE-Chain-Evo

For *SWE-Chain-Evo*, we use OpenHands with the CodeActAgent scaffold as the baseline software-engineering agent. The baseline disables patch memory. For OpenHands + EvoMem, we keep the same OpenHands inference setup with maximum iterations set to 100 and one run per milestone, and prepend retrieved patch-memory context to the task description. Milestones within a chain are processed sequentially, while baseline and EvoMem use the same task order and model configuration. The patch-memory context budget is set to 2200 characters, and patch summarization uses temperature 0.0.

F.3 PersonaMem-Evo

For *PersonaMem-Evo*, we use A-Mem (Xu et al., 2025) as the baseline memory agent with memory retrieval top-$k = 12$. The A-Mem baseline does not use patch retrieval. For *A-Mem + EvoMem*, we keep the same memory retrieval budget, set patch retrieval top-$k = 3$, and use a patch similarity threshold of 0.4. For answer generation, we use temperature 0.0 to reduce sampling variance in multiple-choice prediction. The memory-building temperature is set to 0.7 for both the baseline and EvoMem settings.

F.4 GAIA

For *GAIA*, we use *Memento-Skill* (Zhou et al., 2026) as the base agent. We evaluate on the repository split `gaia_data/data/split_by_level_60_40/train`, which contains 100 GAIA instances balanced across difficulty levels. We use this split because it provides a larger and more balanced evaluation set than the held-out test split. Although the original Memento-Skill workflow uses this training split to learn skills and evaluates them on the test split, our goal is different: we compare the same Memento-Skill agent with and without EvoMem on the same set of tasks. Thus, the experiment isolates whether EvoMem improves reuse of task-solving experience, while keeping the task set fixed across compared agents.

Both the baseline and EvoMem settings use the same `read-write-optimize` protocol, with one feedback round (`-optimize-attempts 1`) and the post-update unit-test gate enabled. Final answer correctness is judged using `gpt-5.4-mini`. The baseline uses `-learning-tips-mode baseline`, which injects only the persistent global `TIP.md` file and does not retrieve task-specific

patch memories. EvoMem uses the hybrid tip pipeline: it keeps the same global `TIP.md` guidance and additionally retrieves the top-$k = 2$ task-versioned tips from the MemGit store using BM25 relevance to the current task. Therefore, EvoMem changes only the memory retrieval layer, while the task-solving and optimization loop remain identical across settings.

F.5 LoCoMo

For *LoCoMo*, we use A-Mem as the baseline memory agent with memory retrieval top-$k = 10$. The A-Mem baseline does not use patch retrieval. For *A-Mem + EvoMem*, we keep the same memory retrieval budget, set patch retrieval top-$k = 2$, and use a patch similarity threshold of 0.0. For answer generation, we use temperature 0.5 for category-5 questions and the default temperature otherwise. The memory-building temperature is set to 0.7 for both the baseline and EvoMem settings.

G Example of EvoArena

G.1 Terminal-Bench-Evo

Example: Terminal-Bench-Evo configure-git-webserver workflow

Evaluation setting. The agent operates in a Linux terminal environment and must configure a small git-backed web deployment service. The stable objective is to let a user push an HTML file to a remote git repository and then retrieve the deployed page from a web server. Across variants, the same high-level service goal is preserved, but the required deployment mechanism, filesystem target, permission model, and branch policy evolve.

Stable workflow objective.

> Configure the server so that pushing `hello.html` to the git repository automatically makes the file available from the web server on port 8080.

Evo stages.

• **Prototype: Basic git-to-web deployment.** The agent configures a git server and a web server so that a user can clone the repository, commit `hello.html`, push to `master`, and then retrieve the page through HTTP.

• **EVO-1: Hook-based deployment with transient rejection.** The service objective remains the same, but deployment must now happen through a git `post-receive` hook rather than manual file copying. The environment provides hook templates, and the first push is expected to fail because of the deployment logic before a retry succeeds. The deployed web root must also contain a marker showing that deployment happened through the hook.

• **EVO-2: Reconcile serving path and deployment path.** The workflow still requires automatic deployment after git push, but the environment introduces an intentional mismatch between template assumptions and the actual web-serving directory. The agent must align the hook deployment target with the directory served by nginx, rather than blindly using the previous web root.

• **EVO-3: Secure web root and group-permission deployment.** The task adds a permission policy. The web root moves to a secure directory owned by `root:www-data` with group-write deployment semantics. The git user must deploy through group membership, so a solution that only changes paths without fixing ownership and permissions is insufficient.

• **EVO-4: Main-branch-only deployment.** The git workflow changes from `master` to `main`. Pushes to `master` must be rejected server-side, and deployment should occur only for `main`. The secure web root and group-permission model from the previous stage must still be preserved.

Why this is an evolution chain.

> A solution that manually copies files into the web root may satisfy the prototype but fails once deployment must be mediated by git hooks. A hook that deploys to the old web root fails when nginx serves a different directory. A path-only adaptation fails when the task introduces group-based permission constraints, and a `master`-based deployment fails once the server enforces a `main`-only branch policy. Thus, the user-facing goal remains stable, but the correct terminal strategy must be patched across deployment mechanism, filesystem layout, permission model, and branch semantics.

G.2 SWE-CHAIN-EVO

Illustrative SWE-Chain-Evo aiohttp milestone sequence

Evaluation setting. The agent works on `aiohttp`, a Python HTTP client/server library. Each milestone targets a different boundary where external input enters the library: persisted cookies, multipart form headers, digest authentication challenges, and access-log timestamp formatting. The stable engineering objective is to harden protocol-facing behavior while preserving existing compatibility.

Stable workflow objective.

Improve aiohttp's handling of externally supplied protocol data and runtime environment details without breaking existing safe behavior.

Evo stages.

• **Milestone 1: Safe legacy cookie loading.** `CookieJar.load()` supports existing saved cookie files, but the legacy pickle path may deserialize arbitrary Python objects. The agent must add a restricted legacy loader that accepts only expected cookie container types, rejects malicious payloads, and keeps safe JSON cookie save/load behavior working.

• **Milestone 2: Multipart header-injection guard.** `FormData.add_field()` already validates non-string `content_type` values, but string values containing carriage returns or newlines can still be accepted and later injected into multipart headers. The agent must reject CR/LF-bearing content types while preserving the existing type-error behavior for non-string inputs.

• **Milestone 3: Digest challenge parsing with empty values.** Digest authentication challenge parsing drops fields whose value is the empty string. Since RFC 7616 permits an empty realm, the parser should retain fields such as `realm=""` instead of discarding them. The agent must preserve empty-string challenge values without changing normal challenge parsing.

• **Milestone 4: Access-log timezone correctness.** `AccessLogger` formats timestamps using a constant timezone offset, which can produce incorrect logs across daylight-saving transitions. The agent must recompute and cache the local timezone from the current localtime offset while preserving existing access-log atom formatting.

Why this is an evolution chain.

The milestones are not isolated toy edits: they form a coherent illustrative sequence over aiohttp's protocol and environment boundaries. A solution pattern that only adds input rejection is appropriate for malicious cookie payloads and header injection, but it would be wrong for digest authentication, where the correct behavior is to preserve an allowed empty value. Similarly, timestamp correctness requires adapting runtime-derived state rather than parsing user protocol input. Thus, the chain preserves a stable engineering theme while forcing the agent to distinguish unsafe input rejection, compatibility-preserving parsing, and environment-sensitive formatting.

G.3 PERSONAMEM-EVO

Example: PersonaMem-Evo item with temporal preference evolution

Evaluation setting. The full history for this persona contains hundreds of turns (over 2×10^5 tokens). We show only the relevant snippets and a few distracting snippets; the model receives the full history.

Relevant context excerpts.

• **Earlier preference state.**
User: "I'm trying to make grocery shopping less chaotic … Do you have suggestions for supermarkets that work best if I want straightforward staples, clear labeling, and not too many random specialty ingredients?"
Assistant: "… Mainstream supermarkets with predictable layouts … clear aisle signage … easy-to-find regular ingredients rather than too many specialty items …"

• **Later preference state.**
User: "I've been trying to branch out with groceries more … I keep seeing sumac, tahini, preserved lemons, and different kinds of chili crisp at the international market near me. Can you give me a practical guide to what these taste like and how to use them?"
Assistant: "Absolutely … those are all very usable ingredients … Sumac: tangy, lemony … Tahini: nutty, earthy … preserved lemons … fermented sauces …"

Distracting context excerpts.

• *User:* "Why do some brunch places feel heavier than others even when the menu sounds similar?"

• *User:* "When a brunch menu is huge, is there a good way to tell whether a place will feel lighter and more balanced?"

Question (temporal trajectory, L1).

She wants to spend a Saturday afternoon food-shopping for a cooking project. Which kind of store and approach would fit her best?

Answer options.

(A) She would most likely choose a standard supermarket, stick to clearly labeled basics, and avoid anything too unfamiliar so the cooking project feels predictable and easy to manage.

(B) She would probably head to a specialty or international market and enjoy asking questions, browsing regional pantry items, and picking up a few unfamiliar ingredients to build the meal around.

(C) She would likely start at a regular grocery store for most items, then maybe add one stop at a specialty shop for a single recommended ingredient if it seemed approachable.

(D) She would probably pick whichever nearby grocery store is most convenient, use a recipe with easy-to-find ingredients, and keep the trip simple so it does not take too much energy.

Correct answer.

(B) The later conversation shows that the user has shifted from preferring mainstream supermarkets to actively exploring specialty or international markets and unfamiliar ingredients. The item therefore requires using the updated preference state rather than the earlier one.