MEMDEV-BENCH: DOES MEMORY HELP CODE AGENTS? BENCHMARKING MEMORY SYSTEMS OVER REAL REPOSITORY HISTORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Code agents increasingly work on evolving real repositories, and memory systems aim to help them reuse earlier experience. However, such experience is diverse and evolving, so it remains unclear how memory systems organize it and thereby affect code agents. Existing benchmarks cannot answer this reliably because they lack either temporally valid histories or control over prior-task relevance. In this paper, we present MemDev-Bench to evaluate how memory systems affect code agents over diverse repository histories. It pairs each task with relevance-annotated earlier tasks from the same repository to build histories with controlled relevance. We also build an automated pipeline that reliably makes these earlier tasks executable and verifiable, producing 4,869 historical tasks for 555 base tasks. Based on MemDev-Bench, our systematic evaluation of memory systems across diverse development scenarios yields three main findings. (1) Memory mainly improves fine-grained localization and editing, increasing functional correctness. (2) Memory also causes failures on otherwise solvable tasks even with related history, and unrelated history tends to increase such failures. These failures mainly stem from agents reusing memory without reassessing its applicability. (3) Compared with using memory systems, directly providing agents with raw historical trajectories substantially reduces their steps. Further analysis suggests that agents save these steps by reusing process information that compressed memories tend to omit. These results suggest that future memory systems should preserve process information and reassess memory applicability against current task context.

1 INTRODUCTION

Code agents powered by large language models (LLMs) have achieved substantial progress on real-world software engineering tasks, such as bug fixing and feature development. Despite this progress, existing agents typically solve each task in isolation and start from scratch. In contrast, human developers continuously accumulate experience from past issues and reuse it to resolve new problems efficiently. Memory systems (Chhikara et al., 2025; Li et al., 2025; Xu et al., 2025; Daga et al., 2026; Newman, 2025) offer a way to close this gap, allowing code agents to retain repository context and problem-solving experience across tasks. However, real repository histories mix valuable experience with abundant irrelevant information. This raises a critical question: do memory systems consistently benefit code agents across long-term repository development?

Addressing this question requires a benchmark with two properties: temporally valid histories from real repositories and controlled relevance between prior and target tasks. Temporal validity requires each prior task to precede the target task without revealing its solution; otherwise, memory gains may come from answer leakage rather than experience reuse. Controlled relevance specifies whether a prior task can help the target task, so a benchmark can separate ineffective memory use from missing useful experience and test whether irrelevant history misleads agents. To our knowledge, no existing benchmark satisfies both properties. Session-based benchmarks build histories from synthetic multi-session scenarios (Singh, 2026) or from task streams of long-term workflows (Wang et al., 2026a), rather than from real repository evolution. Repository-based benchmarks instead build tasks from GitHub histories. Strict temporal filtering is not always applied, so histories may include tasks that postdate the target task. Such histories may therefore leak target solutions (Zhu et al., 2026). Other

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.

benchmarks preserve temporal order. But they either overlook actual task relevance (Joshi et al., 2025) or use chains of almost entirely interdependent tasks (Jin et al., 2026). Both designs make it difficult to assess how memory systems behave across diverse development histories.

We therefore propose MemDev-Bench, a benchmark that evaluates how memory systems affect code agents' task-solving behavior across diverse repository histories. Each instance pairs a base task with relevance-annotated historical tasks from the same repository that precede it and do not reveal its solution. We take base tasks from SWE-bench Verified and SWE-bench Pro (Deng et al., 2026), mine earlier pull requests (PRs) through references and code history, and label each as Strong Related, Weak Related, or Unrelated based on how much it helps solve the base task. Following SWE-bench (Jimenez et al., 2024), we convert these PRs into executable, verifiable tasks. Each PR's patch is split into a solution patch that implements the fix and a test patch that verifies it. The environment is rebuilt from the repository state and dependencies at that time. To make this conversion more reliable, we build a new automated pipeline. Existing pipelines can misclassify test and solution changes or miss patch-induced environment changes, making task validation unreliable (Zhang et al., 2025; Kolavi, 2026; Guo et al., 2026b). To address these issues, our pipeline splits patches using repository test configurations and AST analysis. It also prepares the pre- and post-fix environments separately to capture patch-induced changes. Some pipelines rely heavily on agents for environment planning and debugging, making construction costly at scale (Guo et al., 2026a; Chen et al., 2026). We instead use a rule–experience–agent hierarchy that invokes an agent only when rules and validated experience fail. With this pipeline, we build 4,869 executable historical tasks for 555 base tasks across 22 repositories and 4 languages.

Using MemDev-Bench, we build three history scenarios: Relevant History with only related tasks, Noisy History with added unrelated tasks, and Repo History with longer-term development. In each scenario, the agent solves historical tasks in chronological order, and the memory system builds memory from trajectories. The agent then solves the base task with this memory. Based on these scenarios, we systematically evaluate representative memory systems, yielding three main findings: (i) Across all three scenarios, memory consistently helps solve tasks that the agent fails without it. These gains come mainly from finer-grained localization and editing within files. (ii) Memory also causes failures on tasks that the agent solves without it, even with related history, and unrelated history tends to increase them. (iii) Compared with using memory systems, directly providing raw historical trajectories to agents substantially reduces their steps. Further analysis suggests that this reduction comes from process information preserved in raw trajectories but often omitted by compressed memories. These results suggest that future memory systems should preserve process information and reassess memory applicability against current task context.

2 MemDev-Bench: Construction and Statistics

MemDev-Bench evaluates how memory systems affect code agents on later tasks after these systems build memory from agent trajectories on earlier tasks in the same repository. Each evaluation case pairs a software engineering task, referred to as the base task, with its historical tasks. We collect base tasks from SWE-bench Verified (Chowdhury et al., 2024) and SWE-bench Pro (Deng et al., 2026), and refer to the pull request (PR) that resolves a base task as its base PR. For each base task, we mine earlier PRs from the same repository and annotate how much each PR helps solve the base task. Each annotated PR is then converted into an executable historical task in a reproducible Docker environment, which restores the repository state and dependencies at that point in history.

MemDev-Bench contains 555 base tasks and 4,869 relevance-annotated historical tasks from 22 real-world GitHub repositories, spanning Python, Go, JavaScript, and TypeScript. Figure 1 shows how these tasks are distributed over repositories and languages.

2.1 Candidate Historical PR Mining

A useful set of historical tasks should include both tasks that genuinely help with the base task and other development tasks, reflecting the interleaved development of a real repository. Taking every PR before the base PR would give such a set, but real repositories hold thousands of PRs, and annotating them all and building their environments would be prohibitively expensive. We therefore mine candidate PRs potentially relevant to the base task along two complementary routes: explicit

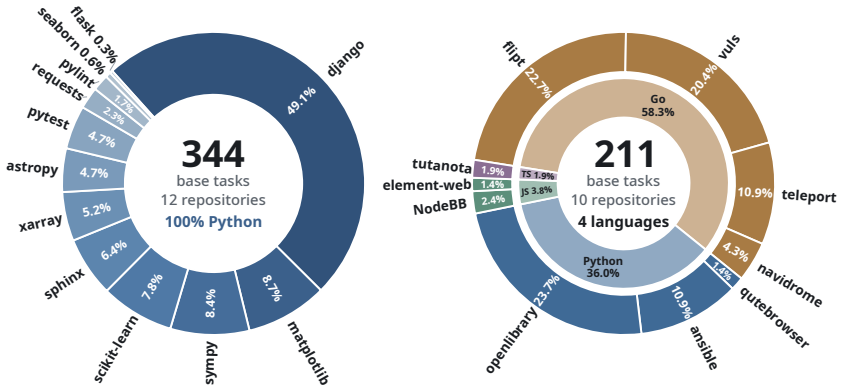


Figure 1: **Base tasks distribution.** Left: SWE-bench Verified (344 tasks, 12 Python repositories). Right: SWE-bench Pro (211 tasks, 10 repositories, four languages).

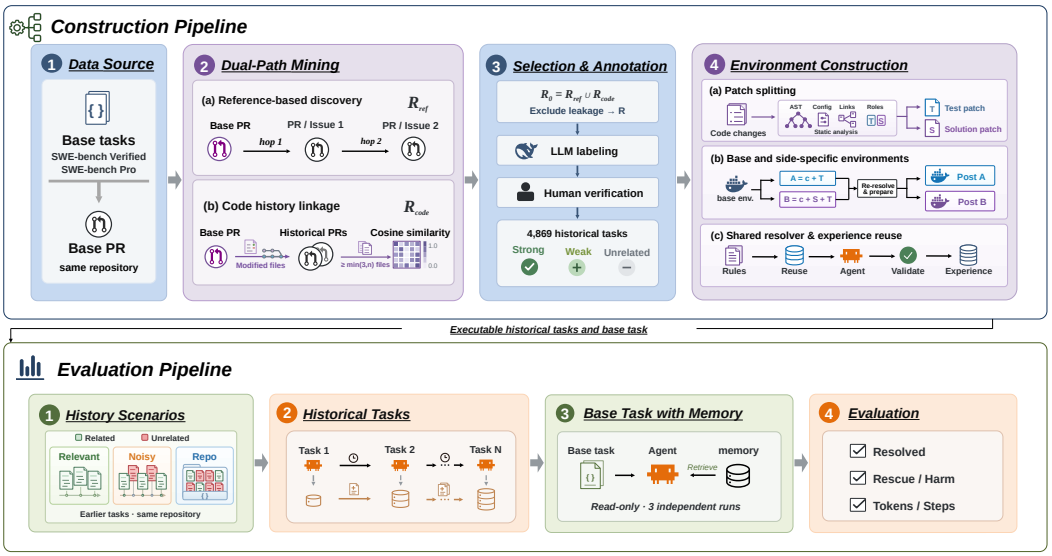


Figure 2: **Benchmark construction and evaluation pipeline.** *Top:* we mine candidate PRs, annotate their relevance, and convert them into executable historical tasks. *Bottom:* an agent solves historical tasks chronologically to build memory, then solves the base task with the frozen memory.

references and code history, as shown in the Dual-Path Mining stage of Figure 2. Temporal order is enforced using each candidate PR's terminal timestamp. A merged candidate must be merged before the base PR is created, while an unmerged candidate must be closed before the base PR is created.

Reference-based discovery. Developers often refer to earlier issues and PRs in PR descriptions and linked issues, and these referenced tasks often depend on or share designs with the current task. As shown in panel (a) of the Dual-Path Mining stage in Figure 2, we extract such references from the base PR and its linked issues, then expand once more from the resulting issues and PRs, yielding PRs up to two hops away. Those from the same repository form the candidate set R_{ref}.

Code-history-based discovery. Explicit references capture only the relationships that developers choose to record. Many other historical PRs are related to the base PR at the code level, but they are not explicitly linked. Panel (b) of the Dual-Path Mining stage in Figure 2 shows how these PRs are found. Starting from each file modified by the base PR, we trace its commit history and map the earlier commits to their PRs. Frequently modified files such as configuration and utility files can create accidental overlap between otherwise unrelated PRs. To reduce such overlap, the mapped PRs pass through two filters. The first filter keeps PRs that share at least $\min(3, n)$ changed files with

the base PR, where n is the number of files changed by the base PR. The second filter encodes each PR with the MiniMax embo-01 embedding model, using its title, body, changed file paths, commit messages, and code diffs as input. It then keeps PRs whose cosine similarity to the base PR exceeds 0.65, and these PRs form R_{code}. The two routes form a preliminary pool $R_0 = R_{\text{ref}} \cup R_{\text{code}}$. We screen R_0 for solution leakage, such as backports and near-duplicate fixes. Human reviewers then exclude historical tasks that directly reveal the base solution. The remaining PRs form the candidate pool R for relevance annotation.

2.2 Candidate Historical PR Annotation

Relevance levels. After building the candidate pool R, we assign each candidate one of three relevance levels. A **Strong Related** PR provides direct and actionable help for the base task, such as a reusable implementation or the same root cause. A **Weak Related** PR provides relevant implementation or testing context, but its concrete solution cannot transfer to the base task. An **Unrelated** PR provides no substantive help, even if it shares files or code with the base PR. Strong/Weak Related PRs include merged PRs only, since accepted changes provide more reliable experience for later tasks. Unrelated PRs also include some unmerged PRs, which reflect the failed attempts and incomplete records of real development history. We further assign each candidate one of 20 fine-grained relation types that characterize its relationship to the base task. Appendix B details these types.

Annotation procedure. To annotate the candidates, we first use DeepSeek-V4-Pro (DeepSeek-AI, 2026) to label each candidate based on the textual descriptions and code changes of both the candidate and the base task. Appendix C details the LLM annotation setup. Next, three senior software engineers with extensive development experience in the benchmark languages verify these annotations. Samples are partitioned among them and exchanged so that every sample is examined by all three, and disagreements are resolved through discussion. After cross-checking, 87.82% of the LLM annotations are retained, while the remainder are replaced with the human consensus labels. Because this verification exposes annotators to the LLM annotations, another three engineers independently label $2{,}270$ samples from scratch, covering all Strong Related and Weak Related samples plus 500 randomly sampled Unrelated cases. These blind annotations reach Fleiss' κ (Fleiss, 1971) of 0.88 for relevance level and 0.80 for fine-grained relation type. They match the main-verification labels for 97.1%, 92.0%, and 96.2% of Strong Related, Weak Related, and Unrelated samples, respectively. Remaining disagreements are resolved through discussion to obtain the final labels.

2.3 Executable Historical Task Construction

After annotation, we convert each candidate PR into an executable historical task, which includes a reproducible environment and its solution and test patches. Prior pipelines face three key issues in environment construction (Jimenez et al., 2024; Zhang et al., 2025; Kolavi, 2026; Guo et al., 2026b;a; Chen et al., 2026). As shown in the Environment Construction stage of Figure 2, our pipeline addresses them with three components. First, prior pipelines mainly split test and solution patches by file paths or names, which can misclassify changes when repositories use custom test layouts. To avoid this, our patch splitter combines repository-specific test configurations with AST-based static analysis to separate the two patches according to code dependencies. Second, many pipelines reuse one fixed environment recipe after applying the patch and thus miss environment changes introduced by the patch. We therefore prepare the pre- and post-fix environments separately, so these changes no longer make correct fixes fail or incorrect fixes pass. Third, pipelines that rely heavily on agents for planning and debugging are costly at scale. We use a resolver with a rule–experience–agent hierarchy, invoking an agent only when deterministic rules and validated fixes from earlier tasks both fail. Validated agent fixes are then stored for later tasks with matching conditions. Appendix D details each component and compares these designs with prior methods.

2.4 Historical Task Statistics

Through the above process, we construct MemDev-Bench with 555 base tasks and 4,869 associated historical tasks. The SWE-bench Verified and Pro base tasks are associated with 2,987 and 1,882 historical tasks, averaging 8.68 and 8.92 per base task, respectively. We next analyze these tasks by relevance level, time gap, and change size.

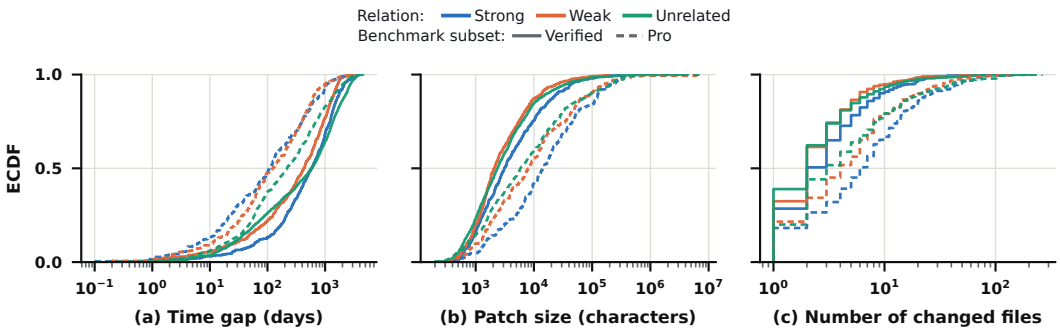


Figure 3: **Empirical cumulative distributions of historical task properties.** (a) Time gap to the base task; (b) non-test patch size in characters; (c) number of changed files.

Label composition. As shown in Table 1, Strong Related and Weak Related tasks, which provide information useful for the base task, account for 40.1% and 30.4% of the historical tasks in the Verified and Pro subsets, respectively. Unmerged PRs appear only in Unrelated, reflecting failed attempts and incomplete historical records.

Table 1: **Relevance composition** (%).

Relevance level	Verified	Pro
Strong Related	19.6	13.5
Weak Related	20.5	16.9
Unrelated, merged	47.0	65.2
Unrelated, unmerged	12.9	4.4

Time gap. As illustrated in Figure 3a, the gap between a historical task and its base task ranges from under a day to several years at every relevance level in both subsets. MemDev-Bench therefore covers both recent and long-term development history.

Change size. As illustrated in panels (b) and (c) of Figure 3, non-test patch size and changed-file count span a wide range at every relevance level. The tasks range from compact local edits to large changes across many files, with SWE-bench Pro tasks involving larger changes.

Together, these distributions show that historical tasks vary substantially in relevance, temporal distance, and change scale. This diversity enables MemDev-Bench to construct evaluation scenarios that better reflect the histories encountered in real repository development.

3 Experimental Setup

To control evaluation cost, all experiments use the Verified subset of MemDev-Bench, which contains 344 base tasks and 2,987 historical tasks from 12 Python repositories.

Agent and base model. We use Claude Code (Anthropic, 2025) as the agent and DeepSeek-V4-Flash (DeepSeek-AI, 2026) as the base model. Automatic context compaction is disabled to avoid affecting the measured effect of memory. Configurations remain fixed throughout the evaluation; Appendix E.1 provides full details and hyperparameters.

Information isolation. Measuring the effect of memory requires the agent to solve each task on its own. However, SWE-bench tasks come from public GitHub repositories, and their original PRs and fix patches remain available online. When web access is unrestricted, the agent often retrieves them and resolves $93.31\% \pm 0.87\%$ of the 344 SWE-bench Verified tasks even without memory. Such answer leakage makes the resolve rate reflect answer retrieval rather than task solving, and leaves limited headroom for measuring the effect of memory. We therefore apply an information-isolation protocol to all experiments. The prompt prohibits the agent from searching for task solutions. In addition, tool-level rules block sources that may reveal answers, such as search engines and public code-hosting platforms, while keeping the tools needed for normal task solving. Under this protocol, the resolve rate drops to $73.16\% \pm 0.61\%$, which better reflects the agent's genuine task-solving ability. Appendix E.2 gives the full prompt, the blocking rules, and the tools that remain available.

Task subsets and history scenarios. The three no-memory runs under information isolation define two task subsets based on baseline consistency. **Improvement Tasks** comprise the 133 tasks that fail in at least two runs and evaluate whether memory can solve otherwise failed tasks. From the 211 tasks that succeed in at least two runs, we sample 50 **Preservation Tasks** covering all SWE-bench

Verified repositories to evaluate whether memory preserves task-solving ability while controlling evaluation cost. Each subset is evaluated under three history scenarios based on the relevance annotations. **Relevant History** contains Strong Related and Weak Related tasks (3–10), representing continued development around related features or code regions. **Noisy History** additionally includes Unrelated tasks (7–21), representing interleaved related and unrelated work. **Repo History** reflects longer-term development. It keeps all Strong Related and Weak Related tasks and adds the most recent earlier tasks associated with other base tasks in the same repository. To control cost and avoid exceeding the context window, we cap each history at 35 tasks, giving 13–35 tasks per base task.

Baselines and memory systems. We use two baselines. **NoMem** provides no historical information and serves as the main reference; its results come from three fresh runs on each subset, independent of the runs used for task selection. **FullTraj** places raw historical trajectories into the agent context in chronological order, without compression or selection, giving a long-context reference for assessing the benefit of memory organization. Against these baselines, we evaluate five representative systems that integrate with Claude Code and cover academic methods, open-source systems, and mature production systems: A-Mem (Xu et al., 2025), Mem0 (Chhikara et al., 2025), MemOS (Li et al., 2025), Supermemory (Daga et al., 2026), and Claude-Mem (Newman, 2025). For broader coverage, Appendix F reports results for 29 memory systems under Relevant History.

Evaluation protocol. For each memory system, task subset, and history scenario, we follow the evaluation pipeline in Figure 2: the agent solves historical tasks in chronological order while the memory system generates, stores, and organizes memory. The resulting memory store is then frozen and used read-only for three base-task runs with distinct random seeds, preventing information from the current task or one run from affecting another.

4 Results and Analysis

4.1 End-to-End Performance

Tables 2 and 3 compare end-to-end performance across systems and history scenarios. Resolved measures task success; Rescue and Harm quantify gains and regressions relative to NoMem; Input Tokens, Output Tokens, and Agent Steps measure solving cost. Appendix H.1 defines the metrics.

Table 2: Results on 133 **Improvement Tasks** under Relevant (Rel.), Noisy (Noi.), and Repo history. Second lines give three-run SDs, or 95% bootstrap CIs for Rescue/Harm. Bold marks the best non-NoMem result; lower is better for Harm and cost. Appendix H.1 defines the metrics.

	Resolved (%)			Rescue (%)			Harm (%)			Tok. In (M)			Tok. Out (K)			Steps		
System	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo
NoMem	35.3 ±2.7			–			–			6.45 ±0.23			59.5 ±3.6			185.1 ±5.3		
FullTraj	39.1 ±0.8	37.3 ±1.6	39.6 ±1.7	12.2 8.9–15.9	11.2 7.8–14.9	11.8 8.4–15.4	8.4 5.3–11.9	9.2 6.4–12.0	7.5 5.1–10.0	16.31 ±0.15	24.11 ±1.46	32.86 ±2.18	**50.2** ±1.3	**41.2** ±1.2	**37.4** ±3.8	**141.7** ±1.7	**114.5** ±6.0	**113.1** ±7.6
A-Mem	40.4 ±2.4	38.1 ±4.3	41.6 ±1.6	11.4 8.1–14.8	10.4 7.1–13.9	**13.2** 9.6–17.1	6.3 4.0–9.0	7.6 5.1–10.3	6.9 4.6–9.6	6.11 ±0.07	6.47 ±0.36	6.51 ±0.31	61.9 ±1.6	63.0 ±2.2	62.7 ±1.7	178.0 ±0.8	180.6 ±5.0	183.8 ±4.1
Mem0	40.1 ±2.2	38.1 ±4.1	41.1 ±1.1	**12.5** 8.8–16.5	11.1 7.8–14.6	11.9 8.0–16.2	7.8 5.2–10.6	8.3 5.5–11.4	6.1 3.4–9.2	6.26 ±0.27	6.39 ±0.33	**5.85** ±1.00	63.9 ±1.6	62.2 ±2.6	52.1 ±9.8	171.8 ±1.4	177.0 ±4.2	166.4 ±11.9
MemOS	36.8 ±2.3	37.3 ±2.3	35.3 ±5.3	10.7 7.7–13.9	11.3 8.2–14.8	9.9 7.3–12.7	9.2 6.3–12.4	9.3 6.3–12.4	9.9 6.8–13.0	6.49 ±0.53	6.56 ±0.17	6.68 ±0.43	63.4 ±4.2	65.8 ±2.6	66.5 ±4.4	189.5 ±5.1	188.0 ±1.2	189.2 ±5.8
Supermemory	40.6 ±1.3	**40.6** ±2.7	39.3 ±1.6	11.7 8.5–15.2	**13.1** 9.5–17.0	11.2 7.8–15.0	6.4 4.3–8.8	7.8 5.3–10.5	7.2 4.9–9.6	6.39 ±0.14	6.25 ±0.06	6.20 ±0.06	62.8 ±0.7	61.6 ±1.6	60.7 ±2.6	183.8 ±0.6	182.3 ±2.4	183.3 ±2.5
Claude-Mem	**41.6** ±2.3	40.1 ±3.8	**42.6** ±1.1	12.2 8.9–15.7	11.5 8.3–15.0	12.1 8.9–15.5	**5.9** 3.8–8.2	**6.8** 4.2–9.7	**4.8** 3.2–6.7	**5.82** ±0.09	**6.03** ±0.20	6.23 ±0.41	57.9 ±2.2	59.8 ±1.4	60.1 ±4.0	180.0 ±2.6	181.1 ±4.4	180.8 ±4.5

Regressions offset gains, especially on already solvable tasks. For Improvement Tasks, Table 2 shows that Resolved rises from 35.3% for NoMem to 39.6% with FullTraj and up to 42.6% with memory systems. These gains are nevertheless limited by regressions: Rescue ranges from 9.9% to 13.2%, while Harm remains 4.8%–9.9%. Claude-Mem leads under Relevant and Repo history (41.6% and 42.6%) with low Harm (5.9% and 4.8%), while MemOS loses most of its gain to Harm. Table 3 shows that this issue is more pronounced on Preservation Tasks, where NoMem already resolves 94.0% of tasks. Averaged over the five memory systems, Harm exceeds Rescue under Noisy (3.7% vs. 3.2%) and Repo history (3.2% vs. 2.6%), lowering Resolved below NoMem to 93.5% and 93.3%. Even modest regressions can thus erase the gains from memory.

Table 3: Results on 50 **Preservation Tasks**. Scenarios and reporting conventions follow Table 2.

	Resolved (%)			Rescue (%)			Harm (%)			Tok. In (M)			Tok. Out (K)			Steps		
System	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo
NoMem	94.0 ±2.0			–			–			3.08 ±0.24			34.9 ±3.3			130.5 ±7.1		
FullTraj	**97.3** ±1.2	92.7 ±2.3	90.0 ±2.0	4.0 0.7–8.0	2.9 0.4–6.2	2.7 0.7–5.3	**0.7** 0.0–2.0	4.2 1.6–7.3	6.7 3.1–10.4	7.22 ±0.43	11.80 ±0.25	22.99 ±1.03	**29.6** ±0.9	**24.4** ±3.1	**22.5** ±2.5	**95.1** ±5.6	**83.4** ±2.0	**81.2** ±3.6
A-Mem	95.3 ±3.1	**94.0** ±2.0	93.3 ±3.1	**4.4** 1.1–8.9	3.1 0.7–6.0	2.7 0.7–5.3	3.1 0.7–5.8	**3.1** 0.9–5.8	3.3 0.9–6.2	3.30 ±0.34	3.50 ±0.30	3.55 ±0.13	37.8 ±3.0	41.6 ±1.3	39.1 ±1.0	128.9 ±5.8	131.2 ±2.7	134.8 ±2.3
Mem0	96.7 ±1.2	**94.0** ±3.5	91.3 ±1.2	**4.4** 0.9–8.9	**4.0** 0.9–8.0	1.6 0.0–3.6	1.8 0.2–3.8	4.0 1.1–8.0	4.2 0.9–8.7	3.47 ±0.35	3.37 ±0.28	3.07 ±0.23	41.3 ±2.9	39.4 ±1.1	35.9 ±1.7	128.2 ±5.3	127.8 ±3.8	122.6 ±2.4
MemOS	95.3 ±2.3	92.7 ±5.8	94.0 ±0.0	3.3 0.4–7.1	2.9 0.4–6.2	2.4 0.2–5.8	2.0 0.2–4.2	4.2 1.1–8.0	2.4 0.4–5.1	3.48 ±0.31	**2.95** ±0.26	3.31 ±0.22	41.2 ±1.0	36.0 ±0.4	39.7 ±2.8	139.5 ±8.9	125.9 ±5.2	135.0 ±4.0
Supermemory	93.3 ±1.2	92.7 ±1.2	91.3 ±3.1	2.2 0.4–4.4	2.9 0.7–5.8	2.2 0.4–4.9	2.9 0.7–5.6	4.2 1.3–7.6	4.9 1.8–8.7	**3.19** ±0.09	3.35 ±0.29	3.33 ±0.25	38.2 ±0.7	39.6 ±5.0	38.6 ±1.0	127.5 ±3.3	131.4 ±4.5	130.8 ±5.1
Claude-Mem	94.0 ±2.0	**94.0** ±0.0	**96.7** ±1.2	2.7 0.2–6.2	3.1 0.7–6.5	**4.0** 0.9–8.0	2.7 0.4–5.6	**3.1** 0.7–6.0	**1.3** 0.0–3.1	**3.19** ±0.24	2.97 ±0.18	**3.04** ±0.29	36.7 ±3.5	35.5 ±1.9	36.2 ±4.0	129.8 ±3.4	127.9 ±3.4	126.7 ±5.1

History noise tends to increase regressions rather than suppress Rescue. Table 2 shows that, when averaged equally over the five structured memory systems, mean Rescue on Improvement Tasks remains nearly unchanged under Relevant, Noisy, and Repo history (11.7%, 11.5%, and 11.7%). This indicates useful history remains exploitable despite unrelated history. From Relevant to Noisy history, mean Harm tends to rise (7.1% to 8.0%) and mean Resolved tends to fall (39.9% to 38.8%). Preservation Tasks in Table 3 show the same trend: mean Rescue changes only slightly from 3.4% to 3.2%, whereas mean Harm increases from 2.5% to 3.7% and mean Resolved decreases from 94.9% to 93.5%. FullTraj is more sensitive, with Harm rising from 0.7% to 4.2%. Thus, history noise tends to introduce regressions rather than reducing the benefit of useful history.

Analysis of solving costs. Memory systems remain close to NoMem in solving cost, while FullTraj uses more input but fewer output tokens and steps. The higher input is expected from including raw historical trajectories; Section 4.4 analyzes the less obvious reductions in output and steps.

4.2 Attribution of Memory Gains

To understand the source of memory gains, we focus on the 68 Improvement Tasks that also fail in all three fresh NoMem runs. These tasks consistently fail without memory across independent runs, so improvements on these tasks can be more reliably attributed to memory. Table 4 reports File, Line, and Edit Recall, which measure agreement with the reference solution patch at the file, edit-location, and edit-content levels, together with F2P and P2P for functional correctness and behavior preservation. Appendices H.2 and H.3 give the definitions.

Table 4: Task-solving outcomes and repair metrics on 68 tasks where all three NoMem runs fail.

	Resolved (%)			File Recall (%)			Line Recall (%)			Edit Recall (%)			F2P (%)			P2P (%)		
System	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo	Rel.	Noi.	Repo
NoMem	0.00 ±0.00			74.22 ±0.37			18.41 ±2.19			11.02 ±0.58			18.10 ±1.01			97.39 ±0.22		
FullTraj	11.27 ±4.73	12.75 ±1.70	11.27 ±1.70	71.61 ±2.95	70.59 ±2.73	70.96 ±1.24	22.16 ±1.76	20.59 ±1.05	19.64 ±1.85	**14.77** ±1.69	**13.90** ±1.64	12.09 ±1.01	26.10 ±4.22	25.81 ±0.19	**27.53** ±3.16	97.45 ±0.28	98.12 ±0.72	97.42 ±0.16
A-Mem	10.29 ±3.89	10.78 ±4.25	**12.75** ±0.85	72.02 ±3.55	**74.59** ±2.09	70.42 ±0.67	17.65 ±1.23	18.67 ±1.70	18.82 ±2.85	11.83 ±0.77	12.38 ±0.83	11.21 ±1.59	25.55 ±2.97	26.99 ±3.55	27.29 ±0.55	97.28 ±0.32	97.37 ±0.29	97.54 ±0.16
Mem0	**12.75** ±0.85	8.82 ±3.89	7.84 ±1.70	72.59 ±4.22	71.41 ±3.36	**76.76** ±1.84	**22.54** ±1.20	18.21 ±1.84	**22.05** ±1.07	13.58 ±1.18	10.74 ±1.41	**13.11** ±1.52	26.74 ±2.15	23.93 ±3.00	25.45 ±1.54	97.64 ±0.03	97.54 ±0.10	97.28 ±0.00
MemOS	10.29 ±1.47	12.25 ±5.16	8.82 ±7.35	72.55 ±1.00	72.30 ±2.88	72.22 ±0.50	19.06 ±1.39	20.85 ±2.64	18.77 ±2.26	12.76 ±1.39	13.06 ±1.62	11.98 ±1.27	25.90 ±1.01	27.04 ±5.12	23.15 ±6.43	97.57 ±0.17	97.05 ±0.84	97.51 ±0.40
Supermemory	11.27 ±3.06	**13.73** ±6.63	11.76 ±0.00	71.98 ±1.42	73.53 ±0.77	73.08 ±3.00	17.77 ±1.14	**21.29** ±3.60	18.39 ±1.13	11.64 ±0.75	12.96 ±2.67	11.10 ±0.97	**27.71** ±4.24	**28.57** ±5.67	26.70 ±1.94	**98.32** ±0.57	**98.44** ±1.06	**98.27** ±0.73
Claude-Mem	11.76 ±4.41	9.31 ±4.25	8.82 ±1.47	**72.92** ±1.39	71.85 ±1.21	72.02 ±0.60	20.03 ±1.57	19.73 ±2.51	17.69 ±1.68	12.45 ±2.06	11.29 ±0.94	9.69 ±0.80	27.65 ±3.10	24.45 ±4.96	23.79 ±2.05	97.39 ±0.16	97.49 ±0.03	97.60 ±0.11

Memory mainly improves fine-grained code editing. NoMem already achieves 74.22% File Recall, and only 2 of the 18 non-NoMem settings exceed this baseline. In contrast, 13 settings exceed the 18.41% Line Recall baseline and 16 exceed the 11.02% Edit Recall baseline, with maximum gains of 4.13 and 3.75 pp, respectively. Memory thus rarely expands gold-file coverage but more consistently improves within-file localization and editing.

Memory shows larger gains in functional behavior. F2P increases from 18.10% for NoMem to 23.15%–28.57% across all 18 non-NoMem settings, while P2P remains high at 97.05%–98.44%. Notably, the gains in functional metrics are much larger than those in gold-patch alignment metrics.

This gap is expected because the gold patch represents only one valid solution, and a functionally correct solution need not reproduce its exact edits. Together with the Line and Edit Recall gains, the F2P improvements suggest that memory helps agents make more effective edits within relevant files, leading to better functional correctness.

4.3 Attribution of Memory-Induced Regressions

Section 4.2 analyzes memory gains on tasks that NoMem consistently fails. We examine the complementary case: 76 tasks solved in all three fresh NoMem runs. Because failures can still arise from run-to-run variation, we attribute a failure to memory only when memory visibly shapes the failing decision or edit. Manual inspection of failed trajectories from the five memory systems identifies 115 such cases. In these trajectories, **agents tend to follow memory as a solution rather than reassess it against the current task context.** The resulting failures fall into four recurring, non-exclusive failure modes:

1. **Compatibility mismatch.** Agents adopt memory whose assumptions do not fit the current task.
2. **Misapplied transfer.** Agents apply a historical solution to the wrong component or stage.
3. **Scope expansion.** Agents extend a history-derived solution beyond what the current task needs.
4. **Failure to revise.** Agents keep following memory-guided edits despite conflicting evidence.

Failure-mode analysis across history scenarios. As shown in Figure 4, misapplied transfer is the most common mode across all three history scenarios and is especially frequent under Noisy history. Trajectory analysis shows that it repeatedly occurs even after agents identify the correct files, indicating that adapting historical solutions to the current task remains difficult. Compatibility mismatch is most frequent under Relevant history, suggesting that relevance alone does not ensure reliable reuse because related history may contain incompatible assumptions. Scope expansion occurs at similar frequencies across scenarios, while failure to revise is least frequent but most common under Repo history, where more extensive history may increase reliance on memory-derived decisions and reduce revision when current-task feedback contradicts them.

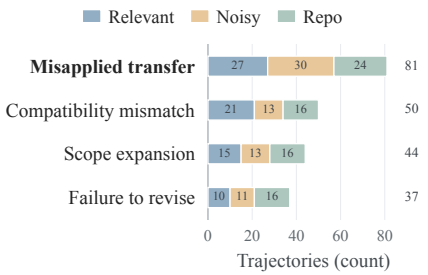
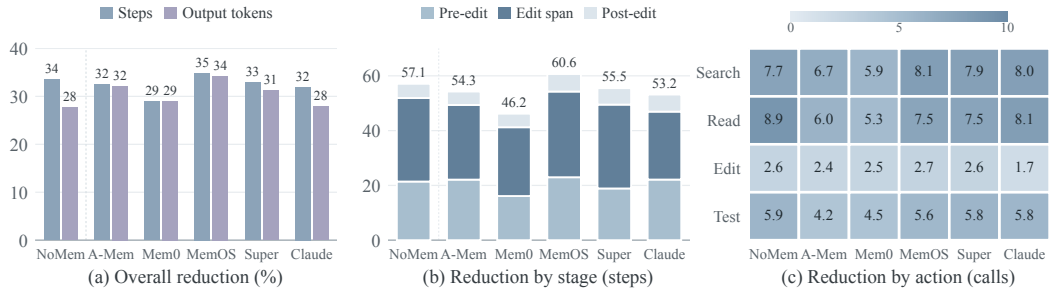


Figure 4: Counts of failure modes.

4.4 How FullTraj Changes the Solving Process

To examine the reductions noted in Section 4.1, we compare FullTraj with NoMem and the five memory systems across both subsets, examining where steps and action calls decrease.



(a) Overall reduction (%) (b) Reduction by stage (steps) (c) Reduction by action (calls)

Figure 5: **FullTraj reductions in steps, output tokens, and action calls.** Comparisons with NoMem and five memory systems use pooled means over 183 instances. Super and Claude denote Supermemory and Claude-Mem.

FullTraj substantially shortens the solving process. As shown in Figure 5a, FullTraj uses about 34% fewer steps and 28% fewer output tokens than NoMem, and 29%–35% fewer steps and 28%–34% fewer output tokens than the five memory systems. All evaluated trajectories finish within the interaction budget, so the observed step reductions are not caused by budget truncation. Figure 5b

decomposes the 46.2–60.6-step reduction into pre-edit, edit-span, and post-edit stages, with the largest decrease occurring during the edit span (24.8–31.3 steps). Figure 5c further shows that the shorter process involves less information gathering: FullTraj makes 5.9–8.1 fewer search and 5.3–8.9 fewer read calls per trajectory, with smaller reductions in test and edit calls.

Understanding FullTraj's lower interaction cost. The five memory systems compress or select historical information and provide the resulting notes or tool summaries through retrieval or context injection. See Appendix G for details. This abstraction can lose information about the sequence of code inspection, failed attempts, revisions, and test outcomes, whereas FullTraj retains these events in execution order. The results here show the potential benefit of preserving the historical solving process rather than only compressed summaries. Meanwhile, Section 4.3 shows that historical solutions can mislead the agent when their assumptions do not hold for the current task. Together, these results suggest that effective memory should preserve the solving process behind historical solutions while making their applicability explicit, a design implication discussed in Section 5.

5 Design Directions

Our fine-grained analysis in Section 4.2 shows that memory improves line- and edit-level recall more consistently than file-level recall. FullTraj further suggests that retaining more process information in execution order reduces the need to reconstruct details omitted by compressed memory representations commonly used by the evaluated memory systems, lowering interaction cost notably during editing; Section 4.4 analyzes this effect. For code tasks, compressed memory is useful for high-level guidance but can be unreliable for fine-grained decisions: recovering omitted details requires extra search and read steps, while missing a critical condition can lead to incorrect localization or modification. Section 4.3 further shows that agents often continue following memory without sufficient reassessment, making such errors harder to correct. Together, these results suggest that memory should combine compact abstractions with finer process or code evidence for concrete edits, while agents should reassess historical guidance when current evidence no longer supports it.

Recent work has begun to address both sides. For memory representation and exposure, MCMA organizes experience across abstraction levels, MemHarness reconstructs retrieved experience for the current state, CoEvo-Mem co-adapts retrieval and memory through feedback, and CICL prioritizes evidence that can change the next action (Liang et al., 2026; Wu et al., 2026; Ye et al., 2026; Guan et al., 2026). These methods move beyond fixed memory replay, but still lack a mechanism to adaptively select historical evidence at the granularity required by each concrete decision. For memory reliance, MemHarness critiques experience before acting, BASM adds applicability boundaries and recovery guidance, MSCE associates skills with applicability, verification, and reliability signals, and Recuris uses working state and execution evidence for validation-gated memory evolution in a bounded recursive self-improvement (RSI) loop (Wu et al., 2026; Lin et al., 2026; Tang et al., 2026; Yu et al., 2026). These methods increasingly control whether history should guide the agent, but this remains separate from deciding what granularity and amount of evidence to expose.

The remaining challenge is therefore to couple memory exposure and memory reliance at decision time. Building on multi-level memory, future systems should determine the granularity and amount of historical evidence needed for each decision. The selection of historical evidence should be coupled with how strongly that evidence influences the agent, and both should adapt as the current state and feedback evolve. More broadly, effective memory should serve as adaptive decision support rather than fixed historical guidance for the agent to follow.

6 Conclusion and Limitations

Conclusion. Experiments on MemDev-Bench show that memory can improve task performance, but noisier histories tend to increase memory-induced regressions. Memory primarily improves within-file localization and fine-grained editing, while FullTraj shows the value of preserving process information, suggesting that applicability assessment and process preservation remain key challenges.

Limitations. Our evaluation is limited to the SWE-bench Verified subset of MemDev-Bench under a single agent–model configuration. We also do not systematically compare our construction pipeline with existing methods. Broader evaluation and direct pipeline comparisons remain future work.

REPRODUCIBILITY STATEMENT

We provide detailed specifications and anonymized implementations to support reproduction of MemDev-Bench and our experiments. Section 2.1 (*Candidate Historical PR Mining*) describes the historical-task mining procedure, while Section 2.2 (*Candidate Historical PR Annotation*) and Appendix C (*Historical PR Relevance Annotation*) specify the relevance annotation procedure and complete annotation protocol. Section 2.3 (*Executable Historical Task Construction*) and Appendix D (*Executable Historical Task Environment Construction*) describe executable task construction, environment preparation, and validation. The corresponding automated task-construction pipeline is available at `https://anonymous.4open.science/r/anonymous_task_builder-35E2/`. Section 3 (*Experimental Setup*) and Appendix E (*Experimental Setup Details*) document the evaluation protocol, with agent and model configurations detailed in Appendix E.1 (*Agent Configuration*) and information-isolation rules in Appendix E.2 (*Information Isolation Rules*). Appendix G (*Memory System Mechanisms*) describes the integration of the evaluated memory systems, and Appendix H (*Metric Definitions*) defines all reported metrics and statistical procedures. The MemDev-Bench evaluation framework and experiment configurations are available at `https://anonymous.4open.science/r/MemDev-Bench-0C0F/`.

ETHICS STATEMENT

Our benchmark is constructed from publicly available GitHub repositories, including issues, pull requests, commits, and code history, and is intended for research on code-agent evaluation. We use only information needed to construct and evaluate software-engineering tasks and do not intentionally collect sensitive personal information unrelated to the research objective. We preserve repository provenance and respect applicable licenses and terms when distributing benchmark artifacts. Human annotators assess only the technical relationships among development tasks. The study does not analyze annotators as research subjects or collect sensitive personal data from them.

AI USE STATEMENT

Generative AI tools were used to assist with literature retrieval and discovery, including identifying and summarizing relevant work; research ideation and execution, including experimental design, analysis, and presentation; and writing assistance, including translation, language editing, and improving clarity and concision. All AI-assisted outputs were reviewed and verified by the authors. The authors made the final decisions regarding the experiments, analysis, and presentation, and take full responsibility for the content of this work.

REFERENCES

AmarsDing. Aranea-Agents. GitHub repository, 2026. URL `https://github.com/AmarsDing/aranea-agents`. Accessed: 2026-08-24.

Anthropic. Claude Code. Anthropic, 2025. URL `https://www.anthropic.com/claude-code`. Accessed: 2026-08-24.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, Binyuan Hui, and Zhongxin Liu. SWE-Universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*, 2026.

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.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI, 2024. URL `https://openai.com/index/introducing-swe-bench-verified/`. Accessed: 2026-08-24.

Chusi-Truth. picoAgent. GitHub repository, 2026. URL https://github.com/Chusi-Truth/picoAgent. Accessed: 2026-08-24.

CodeWater517. Mem517. GitHub repository, 2026. URL https://github.com/CodeWater517/Mem517. Accessed: 2026-08-24.

cydd-1972. Hybrid memory search. GitHub repository, 2026. URL https://github.com/cydd-1972/hybrid_search. Accessed: 2026-08-24.

Soham Daga, Sreeram Sreedhar, and Dhravya Shah. Supermemory is the new state-of-the-art in agent memory. Supermemory Research, 2026. URL https://supermemory.ai/research/longmembench/. Technical report; accessed: 2026-08-24.

Zijie Dai, Shiyuan Deng, Sheng Guan, Yizhou Tian, Xin Yao, Xiao Yan, and James Cheng. RecMem: Recurrence-based memory consolidation for efficient and effective long-running LLM agents. In *Findings of ACL*, 2026.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Y. He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R. Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *ICML*, 2026.

FairmeHIT. memoria. GitHub repository, 2026. URL https://github.com/FairmeHIT/memoria. Accessed: 2026-08-24.

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

FantaSmallhamster. LDBD SQLite FTS Baseline. GitHub repository, 2026. URL https://github.com/FantaSmallhamster/ldbd-sqlite-fts-baseline. Accessed: 2026-08-24.

Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

Xinyu Guan, Qianyang Zhao, and Yuming Deng. Decision-aware memory cards: Counterfactual-inspired context selection and compression for tool-using LLM agents. *arXiv preprint arXiv:2606.08151*, 2026.

Chuanzhe Guo, Jingjing Wu, Sijun He, Yang Chen, Zhaoqi Kuang, Shilong Fan, Bingjin Chen, Siqi Bao, Jing Liu, Hua Wu, Qingfu Zhu, Wanxiang Che, and Haifeng Wang. MEnvAgent: Scalable polyglot environment construction for verifiable software engineering. In *ICML*, 2026a.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. SWE data construction, automatically! *Proceedings of the ACM on Software Engineering*, 3(FSE):525–546, 2026b.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. Software, 2026. URL https://doi.org/10.5281/zenodo.20953922. Version 0.3.0.

Chuanrui Hu, Xingze Gao, Zuyi Zhou, Dannong Xu, Yi Bai, Xintong Li, Hui Zhang, Tong Li, Chong Zhang, Lidong Bing, and Yafeng Deng. EverMemOS: A self-organizing memory operating system for structured long-horizon reasoning. In *ACL (Long Papers)*, 2026.

huangpengtao00-dotcom. MemFusion. GitHub repository, 2026. URL https://github.com/huangpengtao00-dotcom/memfusion-memory. Accessed: 2026-08-24.

Suozhao Ji, Baodong Wu, Zehao Wang, Lei Xia, Qingping Li, Ruisong Wang, Wenbo Ding, Zhenhua Zhu, Boxun Li, Guohao Dai, and Yu Wang. Infini Memory: Maintainable topic documents for long-term LLM agent memory. *arXiv preprint arXiv:2606.10677*, 2026.

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 *ICLR*, 2024.

Qirui Jin, Lingching Tung, Kenan Li, Qiyang Shi, Yushi She, Huanzhong Jia, Harrison Zhao, Kejing Xia, Zhenbang Du, Yikai Zhang, Jiaxin Pei, Zhenyu Zhang, Zhen Qi, Yuyan Duan, Wenke Lee, and Zijian Jin. ChainSWE: Benchmarking coding agents on multi-bug software maintenance. *arXiv preprint arXiv:2607.02606*, 2026.

JingxuanC. causal-memory. GitHub repository, 2026. URL `https://github.com/JingxuanC/causal-memory`. Accessed: 2026-08-24.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. SWE-Bench-CL: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Milla Jovovich and Ben Sigman. MemPalace. GitHub repository, 2026. URL `https://github.com/MemPalace/mempalace`. Accessed: 2026-08-24.

Adithya S. Kolavi. Repo2RLEnv. GitHub repository, 2026. URL `https://github.com/huggingface/Repo2RLEnv`. Accessed: 2026-09-18.

Christopher Latimer, Nicolò Boschi, Andrew Neeser, Chris Bartholomew, Gaurav Srivastava, Xuan Wang, and Naren Ramakrishnan. Hindsight: Structured agent memory that retains, recalls, and reflects. In *ACL (System Demonstrations)*, 2026.

Ruizhe Li, Licheng Zhang, Benfeng Xu, Mingxuan Du, Zheren Fu, and Weidong Chen. When your agent opens the chat app: Agent-controlled search over raw chat logs rivals structured memory. *arXiv preprint arXiv:2608.12888*, 2026.

Zhiyu Li, Chenyang Xi, Chunyu Li, Ding Chen, Boyu Chen, Shichao Song, Simin Niu, Hanyu Wang, Jiawei Yang, Chen Tang, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhen Tao, Huayi Lai, Hao Wu, Bo Tang, Zhengren Wang, Zhaoxin Fan, Ningyu Zhang, Linfeng Zhang, Junchi Yan, Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofen Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. MemOS: A memory OS for AI system. *arXiv preprint arXiv:2507.03724*, 2025.

Sirui Liang, Pengfei Cao, Jian Zhao, Wenhao Teng, Xiangwen Liao, Jun Zhao, and Kang Liu. Learning how to remember: A meta-cognitive management method for structured and transferable agent memory. In *Findings of ACL*, 2026.

Zihan Lin, Zhenyu Chen, Jiawen Wei, Xiaohan Wang, Jie Cao, Jiajun Chai, Wei Lin, Guojun Yin, and Ran He. When not to imitate: Boundary-aware skill memory for reliable tool-use LLM agents. *arXiv preprint arXiv:2608.22339*, 2026.

linxuhao. ActiveMemoryIndex. GitHub repository, 2026. URL `https://github.com/linxuhao/ActiveMemoryIndex`. Accessed: 2026-08-24.

Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. SimpleMem: Efficient lifelong memory for LLM agents. In *ICML*, 2026a.

Xule Liu, Hanlin Teng, Chao Li, Yanan Ni, Shuo Lu, Audrey Wang, Yijun Liu, Yunfei Wang, Xiaofeng Li, Xian Yi, Yuanfa Li, Kang Zhao, Jian Liang, Yuxuan Chen, Jinyuan Chen, Heng Qu, Kun Shao, and Jian Luan. Mi-Memory: A lifecycle memory framework for personal AI. *arXiv preprint arXiv:2607.18975*, 2026b.

Vasilije Markovic, Lazar Obradovic, Laszlo Hajdu, and Jovan Pavlovic. Optimizing the interface between knowledge graphs and LLMs for complex reasoning. *arXiv preprint arXiv:2505.24478*, 2025.

mhw-mathcode. TraceMem: Temporal evidence-ledger memory service for the agent memory challenge. GitHub repository, 2026. URL `https://github.com/mhw-mathcode/TraceMem`. Accessed: 2026-08-24.

Alex Newman. Claude-Mem. GitHub repository, 2025. URL https://github.com/thedotmack/claude-mem. Accessed: 2026-08-24.

openProustian. openProustian Agent Memory. GitHub repository, 2026. URL https://github.com/openProustian/openProustian-agent-memory. Accessed: 2026-08-24.

simple-boy. Chronicle Memory. GitHub repository, 2026. URL https://github.com/simple-boy/Chronicle-Memory. Accessed: 2026-08-24.

Sarthak Singh. DreamBench-SWE: A multi-session memory-hygiene benchmark for software agents. *arXiv preprint arXiv:2608.20664*, 2026.

Bo Tang, Yang Zhang, Guomian Zhuang, Wenqiang Wei, Gaoyang Zheng, Lindong Xie, Yanchao Tan, Feiyu Xiong, Qingyu Yang, Edward Chung, and Zhiyu Li. From memory to skills: Evidence-grounded co-evolution governance for long-horizon LLM agents. *arXiv preprint arXiv:2607.16621*, 2026.

Tencent Cloud. TencentDB Agent Memory. GitHub repository, 2026. URL https://github.com/TencentCloud/tencentdb-agent-memory. Accessed: 2026-08-24.

Tin11Mn. ChronoHybridMem. GitHub repository, 2026. URL https://github.com/Tin11Mn/chrono-hybrid-mem. Accessed: 2026-08-24.

Bo Wang, Yuqian Yao, Enxi Wang, Luozhijie Jin, Yang Liu, Yiran Suo, Yuxuan Cai, Enyu Zhou, Yufei Gao, Honglin Guo, Tianyu Huai, Li Ji, Zhikai Lei, Bufan Li, Lizhi Lin, Jinxiu Liu, Jie Yang, Jiazheng Zhou, Maosen Zhou, Pengfang Qian, Shichun Liu, Guanshan Liu, Hao Zheng, Yunhao Yu, Hang Yan, Jihua Kang, Xinchi Chen, and Xipeng Qiu. ContextWeave: A real-world workflow benchmark. *arXiv preprint arXiv:2608.04830*, 2026a.

Shu Wang, Edwin Yu, Oscar Love, Tom Zhang, Tom Wong, Steve Scargall, and Charles Fan. MemMachine: A ground-truth-preserving memory system for personalized AI agents. *arXiv preprint arXiv:2604.04853*, 2026b.

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 *ICLR*, 2025.

Deliang Wen, Ke Sun, and Yu Wang. Memory Bear AI Memory Science Engine for Multimodal Affective Intelligence: A technical report. *arXiv preprint arXiv:2603.22306*, 2026.

Rong Wu, Daocheng Fu, Licheng Wen, Xuemeng Yang, Shu Zou, Jianbiao Mei, Yuxin Wang, Hairong Zhang, Yu Yang, Tao Hu, Cong Zhang, Botian Shi, and Pinlong Cai. MemHarness: Memory is reconstructed, not replayed. *arXiv preprint arXiv:2607.28272*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *NeurIPS*, 2025.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

Bowen Ye, Yongchao Xu, Zhijian Li, Xiang Yin, Junkai Ma, and Wenzhao Li. CoEvo-Mem: Co-evolving retrieval policy and memory bank for LLM agents. *arXiv preprint arXiv:2608.01739*, 2026.

Zhaochen Yu, Yingcheng Wu, Zhenfei Yin, Kaiyuan Chen, Zhe Zhao, Mengdi Wang, Shuicheng Yan, and Ling Yang. Recursive experiential–working memory evolution for long-horizon agent harnesses. *arXiv preprint arXiv:2608.24876*, 2026.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a.

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! In *NeurIPS*, 2025.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026b.

Jiayuan Zhu, Junde Wu, Minhao Hu, Shengda Zhu, Jiazhen Pan, Weixiang Shen, Yijun Yang, Fenglin Liu, Jianye Hao, Yueming Jin, Qirong Ho, and Min Xu. SWE Context Bench: A benchmark for context learning in coding. *arXiv preprint arXiv:2602.08316*, 2026.

A Related Work

Code agents and cross-task experience. Code agents such as SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Claude Code (Anthropic, 2025) combine repository navigation, code editing, and tool execution to address software development tasks. Their interaction with the environment provides feedback for revising plans and code changes. However, retaining context during one execution is different from maintaining reusable experience across tasks. For evolving repositories, this experience may include code structure, implementation constraints, and test outcomes, whose usefulness depends on the current task and repository state. Memory systems provide persistent representations and retrieval mechanisms for such reuse (Xu et al., 2025; Chhikara et al., 2025; Li et al., 2025; Daga et al., 2026; Newman, 2025). Our work studies how these systems affect performance on subsequent development tasks.

Memory systems for agents. Existing memory systems differ in what they retain and how experience is organized, updated, and exposed. A-Mem (Xu et al., 2025) represents experience as structured notes with contextual attributes and links, and evolves existing notes as new memories arrive. Mem0 (Chhikara et al., 2025) extracts and consolidates salient information into persistent memories, with an additional graph-based variant for relational information. MemOS (Li et al., 2025) treats memory as a managed resource, using MemCubes to support organization and lifecycle operations across memory forms. Supermemory (Daga et al., 2026) combines source documents with extracted memories and graph relationships, supporting retrieval of both derived information and its underlying context. Claude-Mem (Newman, 2025) captures tool-use observations and session summaries, and exposes them through search and chronological retrieval. These approaches offer different balances between abstraction and retained execution detail. Appendix G describes their integrations in our evaluation.

Recent methods increasingly learn memory management and condition reuse on current evidence. MCMA learns to manage memories across abstraction levels (Liang et al., 2026), while MemSkill learns and evolves reusable operations for extracting, consolidating, and pruning memories (Zhang et al., 2026a). For retrieval, MemRL combines semantic matching with utility learned from execution feedback (Zhang et al., 2026b), and CoEvo-Mem jointly adapts the retrieval policy and memory bank (Ye et al., 2026). CICL selects and compresses evidence according to its estimated influence on the next action (Guan et al., 2026). For applicability, MemHarness reconstructs retrieved experience for the current state (Wu et al., 2026), while BASM records applicability boundaries, avoidance rules, and recovery guidance (Lin et al., 2026). MSCE associates reusable skills with supporting evidence, applicability conditions, and verification signals (Tang et al., 2026). Recuris uses working memory and execution evidence to guide localized, validation-gated updates to experiential memory (Yu et al., 2026). These advances motivate evaluating not only what is remembered, but also when historical guidance helps or misleads code agents.

Benchmarks for cross-task memory in development. Evaluating cross-task reuse requires earlier task experience to be available when an agent tackles a subsequent task. SWE-bench and its Verified and Pro variants provide executable repository-level tasks, but their standard evaluation treats each task independently rather than specifying how memory accumulates across tasks (Jimenez et al., 2024; Chowdhury et al., 2024; Deng et al., 2026). DreamBench-SWE (Singh, 2026) uses synthetic multi-session software tasks with executable oracles to test the retention and appropriate

use of historical evidence. ContextWeave (Wang et al., 2026a) reconstructs task streams from real longitudinal workflows, including coding, and evaluates memory from controlled workspace states with diagnostics for relevance and misleading recall. These benchmarks address complementary aspects of cross-session memory but differ from benchmarks built from historical issue–PR tasks in versioned GitHub repositories.

Among repository-based benchmarks, SWE-ContextBench builds an experience pool from agent trajectories and evaluates retrieval on tasks linked through GitHub issues and PRs, including comparisons of memory frameworks (Zhu et al., 2026). Its reference-based construction permits tasks that share a resolving PR and does not explicitly enforce experience-before-target ordering in repository history. SWE-Bench-CL (Joshi et al., 2025) constructs repository-specific sequences from SWE-bench Verified using timestamps and a difficulty curriculum, and records file-overlap dependencies. Its proposed protocol resets each task to its own base commit while memory accumulates. ChainSWE (Jin et al., 2026) instead forms chains with overlapping code changes and carries agent-modified repository state across tasks, with additional comparisons of retained versus reset conversation history. MemDev-Bench emphasizes human-verified, graded relevance of historical tasks to a base task and constructs Relevant, Noisy, and Repo histories with different compositions and lengths. Historical tasks generate memory in chronological order, while each base-task evaluation starts from a fixed repository state with frozen memory. This supports systematic comparisons of memory systems through task improvements, memory-induced regressions, fine-grained editing, and execution cost under a shared agent configuration.

Executable historical tasks also require reliable environment construction. RepoLaunch in SWE-bench-Live and Repo2RLEnv automate repository environment setup (Zhang et al., 2025; Kolavi, 2026). SWE-Factory and MEnvAgent coordinate setup, testing, and verification, including reuse of previously validated configurations (Guo et al., 2026b;a). SWE-Universe further scales executable task construction through an agent with iterative self-verification (Chen et al., 2026). Our pipeline focuses on repository-aware test–solution patch splitting and separate preparation of pre- and post-fix environments, and uses deterministic rules and validated experience before invoking an agent. These mechanisms support the reconstruction and validation of historical tasks used for memory evaluation.

B Fine-grained Relation Types

On top of the three relevance levels Strong Related, Weak Related, and Unrelated, we define 20 fine-grained relation types: eight for Strong Related, six for Weak Related, and six for Unrelated. Each type describes the specific form of the relationship between a historical PR and a base task. This finer annotation separates the different kinds of help or distraction that historical information can provide, and lets us analyze how a memory system behaves under different types of history. Table 5 gives the complete definitions.

Beyond the coarse relevance levels, these fine-grained types capture the specific ways in which historical PRs can help, contextualize, or distract from a base task. Strong Related types characterize directly transferable implementation or design information, Weak Related types capture useful but indirect context, and Unrelated types distinguish different forms of non-useful or misleading overlap. Figure 6 shows the distribution of these relation types within each relevance level for the Verified and Pro subsets.

C Historical PR Relevance Annotation

This appendix describes how candidate historical PRs are annotated for relevance in MemDev-Bench. It covers the annotation objective, the model instructions, and the input and output formats.

C.1 Annotation Setup

For each base task, we judge how much a candidate historical PR that precedes it can help a code agent solve the current task when used as historical development experience. Relevance here is not textual or topical similarity. It is whether the historical PR carries information of actual value for the current task. We use three levels:

Table 5: **Fine-grained relation types between a historical PR and a base task.** Each relevance level has its own set of types, and an annotation must pick a type that belongs to the level it assigns.

Relation type	Definition
STRONG RELATED	
reusable implementation pattern	The historical PR contains an implementation pattern, code structure, algorithm, or repair strategy that can be reused directly, or applied to the base task after local adaptation.
same code path	The historical PR involves the same critical code path as the base task, or one directly connected to it, so its changes help localize where and how to fix it.
introducing change	The historical PR introduced or substantially modified the behavior, assumption, or implementation that later gave rise to the base task's problem, making it an important source for understanding the current issue.
same problem	The historical PR addresses essentially the same problem or requirement as the base task, possibly in a different version, component, or usage scenario.
same root cause	The surface symptoms of the historical PR and the base task differ, but the two share the same underlying technical cause, so the historical analysis or fix transfers directly to the current task.
design authority	The historical PR establishes a design decision, expected behavior, or system constraint that a correct implementation of the base task must respect.
same test surface	The historical PR contains tests that directly cover the behavior of the base task; its assertions and boundary conditions can guide the fix or its validation.
prerequisite implementation	The historical PR introduced functionality, an API, or infrastructure that the solution to the base task directly depends on or extends.
WEAK RELATED	
same module	The historical PR and the base task involve the same module or component, which gives local implementation background, but its concrete changes do not transfer.
same file, different issue	The historical PR and the base task modify the same file but address different problems; its value lies mainly in conveying the structure or implementation context of that file.
same feature area	The historical PR and the base task belong to the same feature area but involve different sub-features, execution paths, or specific problems.
related API	The historical PR demonstrates the usage, semantics, or evolution of an API relevant to the base task, but does not provide the solution itself.
contextual precedent	The historical PR provides a design, implementation, or maintenance precedent that helps in understanding the codebase, but does not determine the specific fix.
adjacent test	The historical PR contains tests for adjacent or related behavior, giving a reference for testing style and expected behavior, without covering the base task.
UNRELATED	
no overlap	The historical PR and the base task have no substantive relationship in functionality, implementation path, root cause, or design context.
different subsystem	The historical PR belongs to a different subsystem or component, and its changes provide no useful information for understanding or solving the current task.
coincidental file overlap	The historical PR and the base task modify the same file, but involve different code regions, functionality, or development goals, so the overlap is only surface-level.
dependency only	The historical PR and the base task share only a dependency, library, package, or infrastructure component, while the change itself carries no task-relevant information.
docs/CI only	The historical PR mainly concerns documentation, CI, formatting, release configuration, or other auxiliary changes, and offers no substantive help for the base task.
mass refactor	The historical PR touches code relevant to the base task only as a side effect of a large-scale refactor, migration, or repository-wide change.

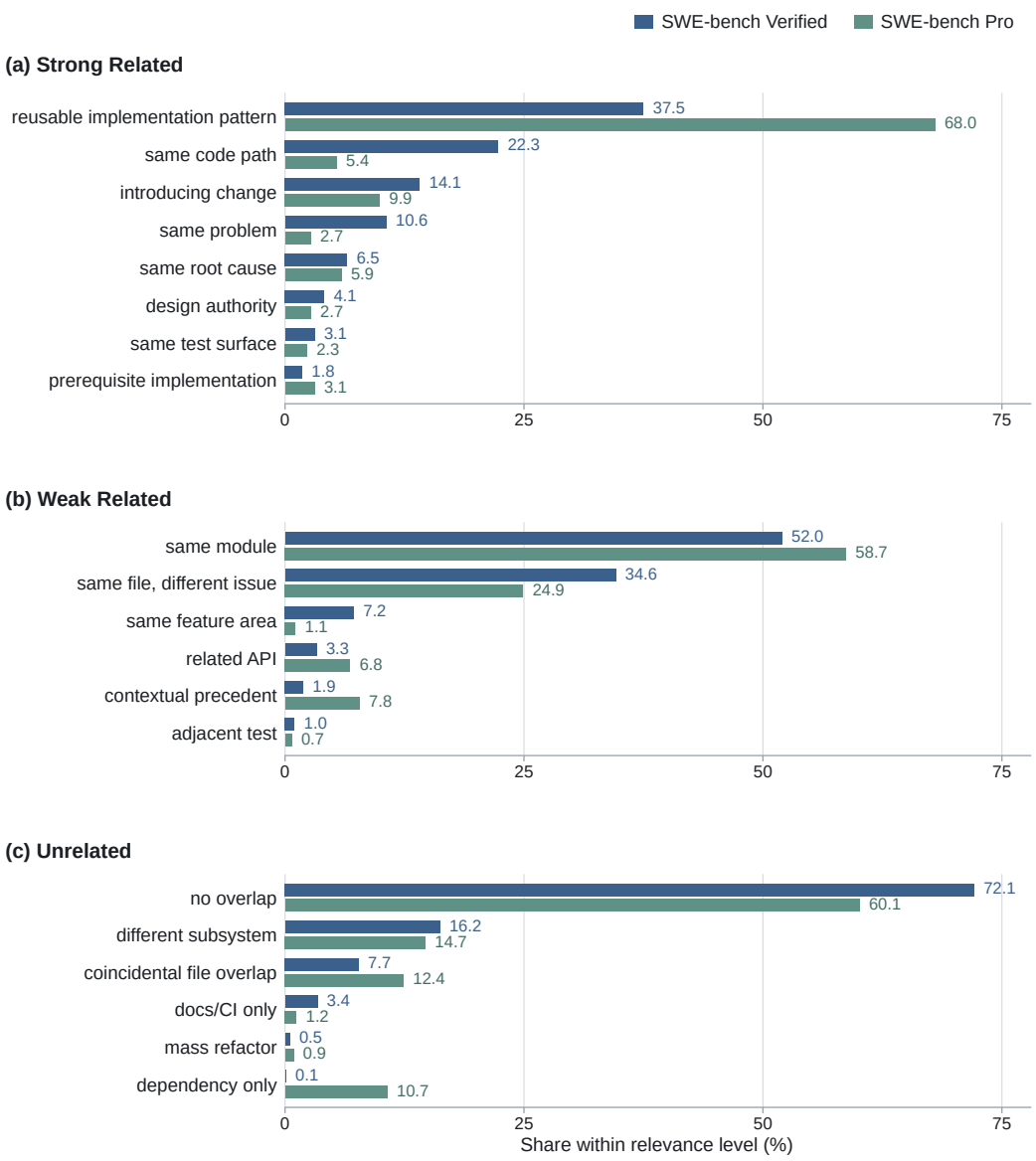


Figure 6: **Distribution of fine-grained relation types within each relevance level.**

- **Strong Related.** The historical PR gives direct and actionable information, such as the key modification site, a reusable implementation, the underlying root cause, a design constraint, or directly relevant test behavior.
- **Weak Related.** The historical PR gives clearly useful module, API, design, or test context, but not enough to determine the solution to the current task.
- **Unrelated.** The historical PR gives no substantive help, even when the two overlap on the surface in keywords, files, or dependencies.

Beyond the level, the annotation model also selects one of the 20 fine-grained relation types defined in Table 5. The selected type must belong to the selected level.

C.2 System Prompt

We reproduce the full system prompt below. It has four parts: the annotation objective and the three relevance levels (Box C.1), the Strong relation types (Box C.2), the Weak and Unrelated relation types (Box C.3), and the decision procedure, boundary rules, and output contract (Box C.4).

Box C.1: System prompt (1/4): objective and relevance levels

You are a relevance annotator for software engineering tasks in a code-agent memory benchmark. Given a **BASE task** that must be solved now and a **HISTORICAL PR** that occurred earlier, your job is to judge:

> *If the code agent could read this HISTORICAL PR as historical development memory while solving the BASE task, what useful information would it obtain, and to what extent would that information help solve the current task?*

The judgment is not whether the two tasks are similar in text, topic, files, or keywords. It is whether the HISTORICAL PR contains historical development experience that helps the agent understand, localize, implement, or validate the BASE task.

You must make two related judgments:

1. Label the HISTORICAL PR as **Strong**, **Weak**, or **Unrelated**.
2. Select, from the relation types associated with that relevance level, the one that best describes the technical relationship between the BASE task and the HISTORICAL PR.

Relevance levels

- **Strong**: the HISTORICAL PR provides direct and actionable information that lets the agent substantially narrow the solution space of the BASE task. Typically, the historical PR alone is already enough to say fairly concretely which code paths should be inspected or modified, which implementation approach to take, which root cause to focus on, which design constraints to follow, or how to construct and validate the relevant tests.
- **Weak**: the HISTORICAL PR provides clear and valuable related context that helps the agent better understand the module, file, feature, API, design, or test behavior involved in the current task, but this information alone is not enough to determine how the current task should be implemented or fixed.
- **Unrelated**: the HISTORICAL PR provides no substantive help for understanding or solving the BASE task, or the two overlap only superficially. Adding this PR to the agent context mainly introduces noise rather than useful historical development experience.

Box C.2: System prompt (2/4): Strong relation types

If the HISTORICAL PR can directly help solve the BASE task, select the most appropriate of the following:

- **reusable implementation pattern**: the HISTORICAL PR contains an implementation pattern, code structure, algorithm, or repair strategy that can be reused directly, or applied to the BASE task after local adaptation.
- **same code path**: the HISTORICAL PR involves the same critical code path as the BASE task, or one directly connected to it, and its changes directly help localize where to modify or determine how to implement the fix.
- **introducing change**: the HISTORICAL PR introduced or substantially modified the behavior, assumption, or implementation that later gave rise to the problem addressed by the BASE task, and is therefore an important historical source for understanding the current issue.
- **same problem**: the HISTORICAL PR addresses essentially the same problem or requirement as the BASE task without directly revealing its solution. The problem may occur in a different version or usage scenario.
- **same root cause**: the surface symptoms of the HISTORICAL PR and the BASE task may differ, but the two share the same underlying technical cause, so the historical analysis or fix transfers directly to the current task.
- **design authority**: the HISTORICAL PR explicitly establishes, modifies, or explains a design decision, API contract, expected behavior, or system invariant that is directly relevant to the BASE task, and a correct implementation of the current task must respect these constraints.
- **same test surface**: the tests in the HISTORICAL PR directly cover the behavior involved in the BASE task, and their inputs, assertions, boundary conditions, or expected behavior can directly guide the implementation or validation of the current task.
- **prerequisite implementation**: the HISTORICAL PR introduced functionality, an API, infrastructure, or a key implementation that the solution to the BASE task directly depends on, calls, or extends.

The core criterion for Strong is **actionability**. The following question can be used as a test:

Box C.2 (continued)

> *Based on this HISTORICAL PR alone, can I already tell the agent fairly concretely where to look or modify, and what kind of modification to make?*

If the answer is yes, Strong is normally the right choice.

Box C.3: System prompt (3/4): Weak and Unrelated relation types

Weak relation types

If the HISTORICAL PR cannot determine the solution to the BASE task but does provide clear and valuable related context, select from the following:

- **same module**: the HISTORICAL PR and the BASE task involve the same module or component, which supplies local implementation background, but its concrete changes do not transfer directly to the current task.
- **same file, different issue**: the HISTORICAL PR and the BASE task involve the same file but address different problems, functions, or code regions; the value lies mainly in understanding the structure or historical implementation of that file.
- **same feature area**: the HISTORICAL PR and the BASE task belong to the same feature area but involve different sub-features, execution paths, or specific problems.
- **related API**: the HISTORICAL PR demonstrates the usage, semantics, constraints, or evolution of an API relevant to the BASE task, but does not directly provide the solution to the current task.
- **contextual precedent**: the HISTORICAL PR provides a design, implementation, or maintenance precedent related to the BASE task, which helps in understanding design intent or development background in the codebase, but does not determine how the current task should be modified.
- **adjacent test**: the HISTORICAL PR contains tests for adjacent or conceptually related behavior, offering a reference for testing patterns, expected behavior, or boundary conditions, but does not directly cover the behavior involved in the BASE task.

Weak must carry clear positive value. Coming from the same repository, occurring at a nearby time, sharing common keywords, or happening to modify the same frequently changed file is not sufficient to justify Weak.

Unrelated relation types

If the HISTORICAL PR provides neither effective implementation information nor valuable context for the BASE task, select from the following:

- **no overlap**: the HISTORICAL PR and the BASE task have no substantive relationship in functionality, implementation path, root cause, or design context that could help solve the current task.
- **different subsystem**: the HISTORICAL PR belongs to a different subsystem or component, and its changes provide no useful information for understanding or solving the BASE task.
- **coincidental file overlap**: the HISTORICAL PR and the BASE task modify the same file, but they involve different code regions, functionality, or development goals, so the file overlap is only a surface-level connection. Be especially careful with frequently modified files such as shared configuration files, shared utilities, fixtures, and `__init__.py`.
- **dependency only**: the HISTORICAL PR and the BASE task share only a dependency, library, package, or infrastructure component, while the historical modification itself provides no implementation or design information relevant to the current task.
- **docs/CI only**: the HISTORICAL PR mainly concerns documentation, CI, formatting, release configuration, licensing, or other auxiliary changes, and contains no code, design, or test information that would help implement or fix the BASE task.
- **mass refactor**: the HISTORICAL PR touches files or code relevant to the BASE task only as a side effect of a large-scale refactor, rename, migration, formatting pass, import sweep, or repository-wide change, and contains no reusable task-relevant information.

Box C.4: System prompt (4/4): decision procedure, constraints, and output

Decision procedure

Proceed in the following order:

1. First check whether the HISTORICAL PR provides information that can directly guide the solution of the BASE task, including but not limited to:
   - the files, functions, or critical code paths that the current task should inspect or modify;
   - an implementation approach that can be reused directly or adapted locally;
   - the underlying root cause of the current problem;
   - a design decision, API contract, or invariant that the current implementation must follow;
   - test logic, boundary conditions, or expected behavior that apply directly to the current task;
   - a historical implementation on which the current solution directly depends.

   If at least one such clear and actionable item exists, select **Strong**.
2. If the solution cannot be guided directly, check whether the HISTORICAL PR clearly improves the agent's understanding of the module, file, feature, API, design, or test behavior involved in the current task. If concrete and valuable context exists but is insufficient to determine the solution, select **Weak**.
3. If the historical PR neither guides the solution directly nor provides clear positive context, select **Unrelated**.

Boundary cases

Judge boundary cases conservatively:

- If both Strong and Weak are defensible, but it cannot be shown that the historical information is sufficient to guide the current modification directly, select **Weak**.
- If both Weak and Unrelated are defensible, but no concrete and valuable context added by the historical PR can be identified, select **Unrelated**.
- If the evidence for relevance comes mainly from keywords, file names, the repository, temporal proximity, or other surface signals, and no actual technical relationship exists, select **Unrelated**.

Important constraints

- Do not use information about the retrieval, mining, ranking, or recall method through which the candidate historical PR was obtained. Relevance must be judged entirely from the content of the BASE task and the HISTORICAL PR.
- Do not assign Strong or Weak merely because similar wording appears in the title, body, problem statement, or commit messages.
- Do not assume relevance merely because the two tasks come from the same repository.
- Do not assume a useful relationship merely because the two tasks modified the same file. Inspect the code regions, functionality, and behavior actually involved.
- The gold patch and the historical diff may be used to understand the real implementation relationship between the tasks, but do not judge by mechanical overlap of code lines; assess whether the changes convey development knowledge useful for the current task.
- Weak is not the default category for uncertain samples. If no concrete positive contribution of the historical PR to the BASE task can be identified, select Unrelated.

Confidence

Set the confidence according to how clear the evidence behind the judgment is:

- **high**: the evidence is direct, explicit, and essentially unambiguous, and independent reasonable annotators would normally reach the same conclusion.
- **medium**: the evidence is sufficient, but the judgment requires combining several signals or applying the Strong/Weak or Weak/Unrelated boundary rules.
- **low**: the judgment relies on weak evidence or substantial inference, and different reasonable annotators might disagree.

Output

Output a single JSON object and nothing else outside it. The JSON contains the following fields:

Box C.4 (continued)

- `label`: the relevance level, restricted to `strong`, `weak`, or `unrelated`.
- `relation_type`: one fine-grained relation type belonging to the selected relevance level.
- `relationship`: one or two short sentences that objectively describe the concrete technical relationship between the BASE task and the HISTORICAL PR. Name specific files, functions, APIs, tests, issues, behaviors, or implementations wherever possible, rather than writing generic statements such as "they are related to the same module."
- `confidence`: restricted to `high`, `medium`, or `low`.
- `brief_reason`: one to three short sentences explaining why the stated relationship implies the selected relevance level. For Strong and Weak, state what concrete help the historical PR offers the agent; for Unrelated, explain why the existing surface overlap does not amount to valuable historical experience.
- `evidence_spans`: one to three verbatim text or code spans from the BASE task or HISTORICAL PR input that directly support the judgment. Do not fabricate content that is absent from the input.

C.3 User Message and Input Format

We issue one independent model call per candidate historical PR. The user message adds no natural language task description. It provides the base task and the candidate historical PR as structured JSON:

User message: input format

```
{
  "base_task": {
    "problem_statement": "<problem statement>",
    "changed_files": ["<file 1>", "<file 2>"],
    "patch": "<gold patch>"
  },
  "historical_pr": {
    "title": "<PR title>",
    "body": "<PR body>",
    "changed_files": ["<file 1>", "<file 2>"],
    "diff_hunks": {
      "<path/to/file_1>": "<diff hunks>",
      "<path/to/file_2>": "<diff hunks>"
    },
    "commit_messages": ["<message 1>", "<message 2>"]
  }
}
```

This input gives the model three complementary signals.

- **Development records.** The problem statement, the PR title and body, and the commit messages carry task intent, the problem description, and what the developers wrote down. They support relations that rest on the same problem, on design decisions, on explicit historical background, or on API semantics.
- **Code structure.** The changed files on both sides show where the two tasks touch the repository, so the model can judge whether they involve the same module, file, or feature area. File overlap alone does not imply useful relevance, so this signal has to be read together with the concrete code changes.
- **Code changes.** The gold patch of the base task and the diff hunks of the historical PR give the actual modifications, so the model compares implementation paths instead of surface wording. This matters most for Strong relations such as *reusable implementation pattern*, *same code path*, *introducing change*, *same root cause*, *same test surface*, and *prerequisite implementation*.

The annotation therefore combines development records, code structure, and actual code changes to decide whether historical development experience supplies reusable information for the current task.

We deliberately leave repository names, PR state, merge status, and GitHub labels out of the input. None of them carries evidence about the technical relationship between the two tasks. Merge status is a further risk: we also use it when we assemble the historical tasks, so exposing it to the model would offer a shortcut that bypasses the judgment we want to measure.

C.4 Output Format

For each base task–historical PR pair, the annotation model returns the following JSON:

Model response: output format

```
{
  "label": "strong | weak | unrelated",
  "relation_type": "<one fine-grained relation type>",
  "relationship": "<concrete technical relationship between the base task and the
  historical PR>",
  "confidence": "high | medium | low",
  "brief_reason": "<why this relationship implies the selected relevance level
  and what information the historical PR provides>",
  "evidence_spans": ["<evidence span 1>", "<evidence span 2>"]
}
```

The relation_type must belong to the selected relevance level. For example, *same root cause* can only accompany Strong, *related API* can only accompany Weak, and *coincidental file overlap* can only accompany Unrelated.

We keep relationship and brief_reason as two separate fields. The first asks the model to describe the technical relationship that objectively holds between the two tasks. The second asks why that relationship implies the selected level and what information the historical PR supplies to the agent. This split stops the model from writing a generic justification for a label it has already committed to, and it lets human verifiers check relationship identification and relevance judgment separately.

The evidence_spans field asks the model to quote the text or code that supports its judgment, so verifiers can quickly check whether the level and the relation type are grounded in the input.

D Executable Historical Task Environment Construction

D.1 Pipeline Overview

As illustrated in Figure 7, we convert mined and relevance-annotated historical PRs into executable historical tasks. Given the pre-PR source state and its code changes, the pipeline sequentially performs patch splitting, base-environment construction, side-specific environment preparation, and repeated testing, producing a frozen task definition, restorable execution environments, and per-test-case outcomes. Base construction and side preparation share the same environment resolver, allowing a unified environment parsing, plan generation, and execution validation procedure to operate on both the original and patched source states.

Patch splitting and task freezing (Fig. 7a). Let c denote the base commit before the PR, and let T and S denote the test and solution patches, respectively. We classify PR changes based on repository test configuration, AST structures before and after the change, statically resolvable code relations, and task-role information. Splitting is performed at the granularity of complete file patches, with tests and supporting changes required for task evaluation assigned to T and implementation changes under evaluation assigned to S. After checking patch applicability and composition order, we freeze c, T, S and their provenance as fixed inputs to subsequent environment construction and testing.

Base-environment construction (Fig. 7b). We first invoke the shared resolver on the original commit c to infer its runtime, dependencies, and build requirements, after which the corresponding environment plan is executed. Neither T nor S is applied at this stage, ensuring that environment changes introduced by the PR are excluded from the base environment. After validation, the re-

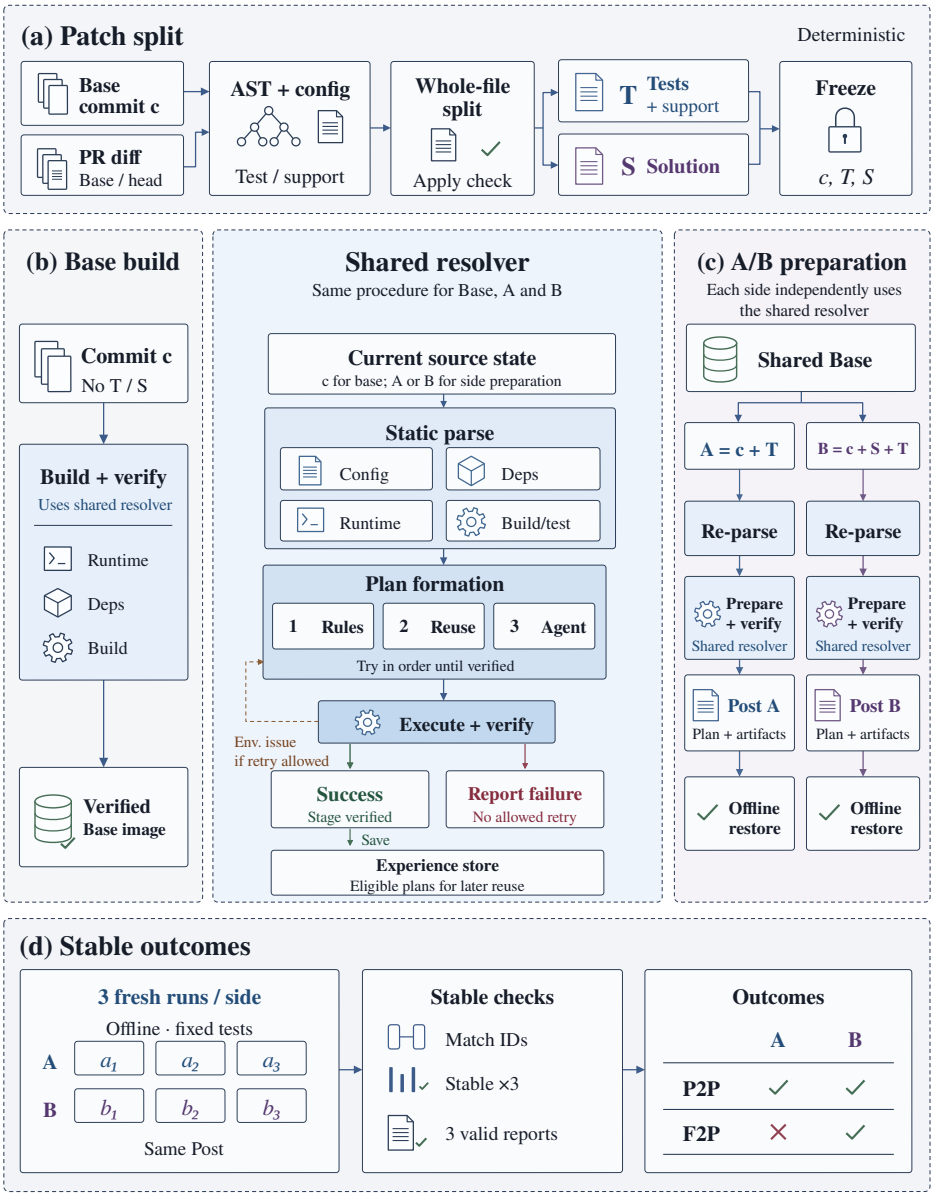


Figure 7: **Executable historical task construction pipeline.** (a) PR changes are split into test patch T and solution patch S, and the task inputs are frozen. (b) The original commit c is resolved and validated to construct a shared Base image. (c) The two patched source states independently invoke the shared resolver, and their plans and environment artifacts are frozen as Post A and Post B for offline restoration. (d) Stable outcomes are obtained from three fresh runs on each side.

sulting environment is frozen as the **Base image**, which provides the common starting point for side-specific preparation and subsequent testing.

Shared environment resolution and validation. The shared resolver operates on the source state available at each invocation. During Base construction, it analyzes the repository at the original commit c; during side-specific preparation, it analyzes the corresponding source state after the side's patch has been applied. It first parses repository configuration, dependency declarations, runtime constraints, and build and test settings, then constructs an environment plan following the priority **Rules** $\rightarrow$ **Reuse** $\rightarrow$ **Agent**. Deterministic rules first resolve configurations that can be derived directly from the repository. Previously validated solutions are reused only when their recorded

applicability conditions match the current source state, and are then revalidated. The environment Agent completes any remaining unresolved parts of the plan. All plans follow the same execution and validation procedure. Validated solutions and their applicability conditions are recorded for reuse in subsequent tasks.

Side-specific preparation and offline restoration (Fig. 7c). Starting from the same Base, we construct two source states:

$$A = c + T, \qquad B = c + S + T,$$

where addition denotes ordered patch application. After the full patch sequence is applied, the shared resolver independently re-parses the environment requirements from the resulting source states A and B. Thus, dependency, runtime, or build changes introduced by S affect only the preparation of B. The validated environment plans and artifacts are then frozen as **Post A** and **Post B**. Finally, we instantiate fresh containers from the Base image and restore each side using its frozen Post state, verifying that both environments can be reconstructed independently.

Repeated testing and stable outcomes (Fig. 7d). Both sides are evaluated through three independent runs. Each run starts from a fresh offline container instantiated from the Base image, applies the corresponding patch, and restores the same frozen Post state for that side. Runs share no writable state and do not reinstall dependencies. We then align test outcomes between the two sides using the exact test-case identifiers recorded during execution. A test case is considered stable only if all three runs contain complete records and produce the same outcome. Cases that consistently fail on A and pass on B are labeled **F2P**, while those that consistently pass on both sides are labeled **P2P**.

D.2 Comparison with Prior Work

We compare prior historical-task construction methods along three dimensions: patch partitioning, patched-environment preparation and validation, and reuse of construction experience.

D.2.1 Patch Partitioning with Repository and Task Context

SWE-bench (Jimenez et al., 2024), SWE-bench-Live (Zhang et al., 2025), and MEnvAgent (Guo et al., 2026a) identify test changes mainly from test-related keywords in file paths. SWE-Factory (Guo et al., 2026b) further considers path tokens, naming conventions, and file types, while Repo2RLEnv (Kolavi, 2026) uses directory components, filename patterns, and exclusion rules. Although these methods employ different heuristics, patch assignment still relies largely on file location and naming. These signals alone cannot reliably determine whether a change defines the test criteria or implements the solution being evaluated.

Our partitioner combines repository-specific test configuration with AST-based analysis, statically resolvable code relations, and the role of each change in the task. For example, a helper located under a test directory may also be used by non-test code. If the task repairs this helper, the change should be treated as part of the solution; assigning it to the test patch would cause the pre-fix evaluation state to contain part of the intended repair. Likewise, when the task itself is to correct an erroneous assertion, the assertion update is part of the solution rather than a test change shared across the pre- and post-fix states. Repository-specific test configuration captures custom test roots and naming conventions that generic path-based rules may miss. Our patch assignment therefore accounts for both repository-specific testing conventions and the functional role of each change in the task.

SWE-Universe (Chen et al., 2026) also examines modification content, using an agent to distinguish test changes from solution changes. In contrast, our partitioner combines repository configuration, code structure, statically resolvable usage relations, and task roles through deterministic rules, without invoking an agent during patch partitioning.

D.2.2 Preparation and Validation of Patched Environments

Existing systems already validate tasks across repository states before and after applying the reference patch. SWE-bench (Jimenez et al., 2024) compares test behavior across these states and provides reusable environment configurations, with some configurations reinstalling the project after

patching. SWE-bench-Live (Zhang et al., 2025) rebuilds both states, evaluating the pre-patch state once and the post-patch state three times in fresh containers. Repo2RLEnv (Kolavi, 2026) compares individual test cases while executing both states within a shared sandbox. SWE-Factory (Guo et al., 2026b) generates and repairs environment and evaluation scripts, while its outer F2P classifier relies on script-level exit codes. MEnvAgent and SWE-Universe likewise use execution feedback to validate and adapt constructed environments (Guo et al., 2026a; Chen et al., 2026). However, these methods do not always re-parse environment requirements from each complete patched source state. As a result, patch-induced changes to runtimes, dependencies, or build requirements may be missed, causing an otherwise valid patch to fail under an environment that no longer matches its source state and confounding task validation with environment mismatch.

To address this mismatch, our design independently re-parses the runtime, dependency, build, and test requirements from each complete patched source state. As described in Section D.1, the resulting environment artifacts are frozen, restored in fresh offline containers, and reused across all three runs for that state. This ties each observation to a specific source state and its prepared environment, while applying the same repetition criterion to both pre- and post-patch execution.

D.2.3 Validated Reuse of Construction Experience

Prior work reuses construction knowledge in several forms. SWE-bench reuses environment recipes and images (Jimenez et al., 2024). RepoLaunch, which supports SWE-bench-Live, can reuse historical environments and commands after execution checks (Zhang et al., 2025). Repo2RLEnv can reuse cached bootstrap results without invoking its construction agent (Kolavi, 2026). SWE-Factory retrieves successful Dockerfiles and test scripts as references for model-based generation (Guo et al., 2026b). MEnvAgent retrieves and validates historical environments, repairing them when needed (Guo et al., 2026a), while SWE-Universe trains its environment-construction model on successful construction trajectories (Chen et al., 2026).

Our pipeline integrates validated experience into the **Rules** $\rightarrow$ **Reuse** $\rightarrow$ **Agent** hierarchy described in Section D.1. Deterministic rules first resolve requirements that can be derived directly from the repository. When unresolved requirements remain, the pipeline retrieves a previously validated construction plan with compatible applicability conditions, executes it in the current environment, and revalidates the result. An agent is invoked only when neither rules nor reusable experience succeeds. Successful agent-generated plans are then stored with their applicability conditions as executable experience for compatible future tasks.

Task-independent Base repairs that repeatedly succeed across independent tasks can further be incorporated into reusable parent-image capabilities after rebuilding and revalidation. This allows construction experience to accumulate from task-level plans into stable Base capabilities, reducing repeated agent planning and exploratory execution while retaining validation on the current task.

E Experimental Setup Details

E.1 Agent Configuration

We use Claude Code as the software-engineering agent and DeepSeek-V4-Flash as the base model. To keep the six evaluation conditions comparable, the agent and model configuration, the interaction budget, and the per-trial execution settings are held fixed. Table 6 lists them. FullTraj retains the complete trajectories generated by the same agent while solving the assigned historical tasks without memory, independent of task outcome. NoMem results come from three fresh runs that are independent of the runs used to define the task subsets. This avoids reusing the runs whose outcomes determine subset membership as the evaluation baseline. Runs interrupted by infrastructure errors, such as API failures or container crashes, are rerun under the same configuration, and the same rule applies to every condition.

E.2 Information Isolation Rules

We apply a unified closed-book information isolation protocol while the agent solves both historical tasks and the base task. The protocol prevents access to task solutions and repository history while preserving the local tools, dependency installation, and task-required connectivity needed for normal

Table 6: Shared agent and execution configuration. Every setting is identical across all evaluated systems and all six evaluation conditions.

Setting	Configuration
Agent and base model	
Agent	Claude Code 2.1.170
Base model	DeepSeek-V4-Flash
Model routing	The primary route, all Claude Code model aliases, and the subagent route use DeepSeek-V4-Flash
Reasoning effort	High
Interaction budget	At most 200 turns per task
Execution environment	
Evaluation framework	Harbor 0.3.0 (Harbor Framework Team, 2026)
Task environment	Docker containers
Resources	1 CPU, 4 GB memory, and 10 GB storage per trial
Time limits	3,000 s for the agent, 3,000 s for verification, and 3,600 s for each environment-start attempt

software development. The agent can use the historical trajectories provided for the current instance and information retrieved from its designated memory store. The memory store is read-only while the agent solves the base task. Each base task uses an independent memory store, so memory never carries across base tasks. Claude Code's built-in persistent memory is disabled in all conditions.

E.2.1 Closed-Book System Prompt

During both historical- and base-task solving, we append the following policy to Claude Code's system prompt.

Box E.1: Closed-book system prompt

You are participating in a strict closed-book software-engineering benchmark.

PERMITTED EVIDENCE

Your primary obligation is to solve the task using your own reasoning. You may use only:

1. The task instruction provided to you.
2. Files in the current working-tree checkout.
3. Output produced by local builds, tests, linters, type checkers, debuggers, and reproduction scripts.
4. Task-provided runtime services that are necessary to reproduce or test the requested behavior.
5. Historical trajectories supplied for the current instance and information retrieved from its designated memory store, when available.

Use only the historical information provided for the current task. You may read the supplied trajectories and query the designated memory store through its provided interface, including network access when required. The restrictions below on external retrieval and repository history do not prohibit this access. During base-task solving, the history is frozen and the memory store is read-only.

NETWORK POLICY

Normal network connectivity may be available because some tasks require external services or installation of missing third-party dependencies. Network access is an execution transport only; it must never be used as an information, research, retrieval, or answer source.

You may use the network only to:

- Install a missing third-party build, runtime, or test dependency.
- Communicate with an endpoint explicitly required to reproduce or test the task.
- Exercise HTTP, API, OAuth, webhook, proxy, package-manager, or other network behavior required by the task.

You must not use the network to:

Box E.1 (continued)

- Search for the task, issue, error message, expected patch, or solution.
- Read documentation, tutorials, blog posts, Q&A pages, release notes, changelogs, or external examples to help solve the task.
- Access search engines, public code hosts, issue trackers, pull requests, commits, patches, benchmark records, leaderboards, or hidden tests.
- Download, install, or inspect another copy or version of the target project.
- Query another model, assistant, search API, retrieval service, or external knowledge service.
- Use curl, wget, Python, package managers, Git, MCP tools, subagents, browsers, or any other mechanism to bypass these restrictions.

Do not use WebSearch or WebFetch. If the task instruction contains a URL, issue number, pull request number, or commit reference, treat it only as text supplied by the task. Do not open or retrieve it.

SOURCE AND GIT POLICY

Work only with the current checkout.

The only permitted Git commands are:

- git status
- git diff limited to the current working tree or index
- git diff HEAD limited to the current working tree or index

Do not use any other Git command, including log, show, blame, reflog, rev-list, cat-file, fetch, pull, clone, checkout, switch, restore, reset, cherry-pick, merge, rebase, archive, branch, tag, or worktree.

Do not inspect .git directly. Do not access Git objects, refs, logs, alternate object databases, remote references, stashes, or historical revisions. Do not reconstruct historical content by any indirect method.

Do not read the target project from site-packages, dist-packages, package-manager caches, another checkout, an installed wheel, a source archive, or any location outside the current checkout. You may inspect genuinely separate third-party dependencies when necessary to understand their local interfaces or runtime behavior.

BENCHMARK PRIVACY POLICY

Do not access benchmark-private /tests or /solution paths, gold patches, verifier implementation details, hidden tests, expected outputs, grading metadata, or results from other agents.

Repository-owned tests inside the current checkout are allowed. You may add a focused reproduction test when useful, but do not delete, skip, weaken, rewrite, or manipulate tests merely to make them pass.

TOOL POLICY

You may use local file-reading, file-editing, shell, build, test, debugging, and static-analysis tools. Subagents, if available, must follow exactly the same closed-book restrictions and may use only the same permitted evidence.

Do not use a permitted tool to simulate or replace a prohibited tool. Do not encode, disguise, split, delegate, or indirectly execute a prohibited operation.

WORKING METHOD

Investigate the current code, reproduce the problem locally when possible, identify the root cause, implement the smallest correct fix, and validate it with relevant local tests.

Base conclusions on the permitted evidence. If information is missing, reason from that evidence rather than using prohibited external sources.

If an operation is blocked or prohibited, do not retry it through another command, tool, process, subagent, encoding, or indirect path. Continue solving from the permitted evidence.

At completion, provide a concise summary of the completed task and the work performed.

Solve the task yourself. Do not obtain the answer, implementation, patch, or decisive technical guidance from any source outside the permitted evidence.

E.2.2 TOOL-LEVEL ENFORCEMENT

The system prompt states the closed-book information boundary; two layers of programmatic control enforce it during execution. First, the Claude Code CLI turns off the tools whose only purpose is external retrieval or work outside the current checkout. Second, a PreToolUse guard validates every request to the remaining tools before execution and rejects commands or arguments that would violate the protocol. Table 7 summarizes tool availability under this scheme. The supplied trajectories and designated memory interface remain accessible under both enforcement layers.

Table 7: Tool availability under the information-isolation protocol. Tools that remain available are checked request by request rather than turned off, so normal local development is preserved.

Tool category	**Availability**	**Enforcement**
Bash	Available	The guard validates each request before execution and inspects the complete command string.
Read, Write, Edit, NotebookEdit, Glob, Grep	Available	The guard validates each request for protected paths, Git metadata, and alternate copies of the target project.
WebSearch, WebFetch	Disabled	Turned off in the CLI and additionally rejected by the guard if invoked, removing the direct route to external information.
EnterWorktree, ExitWorktree	Disabled	Turned off in the CLI and additionally rejected by the guard if invoked, keeping the agent inside the current checkout.

For the tools that remain available, enforcement happens at the request level rather than by turning the tool off. The guard inspects commands, paths, and relevant arguments, and blocks access to external solutions, repository history, benchmark-private information, and alternate copies of the target project, while leaving the operations needed for normal task solving intact. Table 8 lists the main rule families and the behavior each one preserves.

Table 8: Main request-level checks enforced by the PreToolUse guard. Each rule family blocks a route to external answers or repository history while keeping the corresponding development operation available.

Rule family	**Restricted behavior**	**Preserved behavior**
Target-project copies	Inspecting the target project under `site-packages` or `dist-packages`, package-metadata queries that locate such copies, and imports of the target project from outside the current checkout.	Genuinely separate third-party dependencies may still be inspected.
Shell and package acquisition	Remote-access and source-acquisition tools (`gh`, `ssh`, `scp`, `sftp`, `ftp`, `netcat`, `rsync`); `pip download`/`wheel`; `--target`/`-t` installation; and URL- or VCS-based installation.	Ordinary installation of named third-party dependencies.
Git and metadata	Git history, any Git operation other than the permitted commands, direct access to `.git`, Git plumbing utilities, and Git-metadata environment variables.	`git status` and `git diff` on the current working tree or index, optionally against `HEAD`.
Benchmark-private paths	Access to the benchmark-private `/tests` and `/solution` paths.	Repository-owned source files and tests inside the current checkout.
Explicit network access	Explicit URLs in `curl`, `wget`, and recognized inline-Python calls (`requests`, `httpx`, `aiohttp`, `urllib`, `http.client`, `socket`) that target code hosts (GitHub and `githubusercontent.com`, GitLab, Bitbucket), Q&A sites (Stack Overflow and Stack Exchange), search engines (Google, Bing, DuckDuckGo, Brave Search), or SWE-bench.	Networking needed for ordinary dependency installation and task execution.

Continued on the next page

Table 8 continued from the previous page

Rule family	Restricted behavior	Preserved behavior
Guard bypass	Attempts to unset or override `LD_PRELOAD` or network-guard endpoint variables, or to bypass the execution guard by any other means.	Requests that satisfy the isolation checks proceed.
File and search requests	Read, Write, Edit, NotebookEdit, Glob, and Grep requests that touch Git metadata, benchmark-private paths, or alternate installed copies of the target project.	Requests confined to permitted files in the current checkout.

Overall, the protocol implements a closed-book environment rather than a fully offline sandbox: the connectivity required for dependency installation and task execution is preserved, while access to external information and repository history is restricted programmatically.

F Broad Evaluation of Memory Systems

Section 3 studies five representative memory systems across all six evaluation conditions. To cover a wider range of memory-system designs at a manageable evaluation cost, we further evaluate 29 additional systems under the Relevant History scenario on a fixed subset of 150 base tasks. The subset draws on both Improvement Tasks and Preservation Tasks so that the comparison covers memory-induced gains and potential regressions while keeping the evaluation scale bounded, comprising 120 of the 133 Improvement Tasks and 30 of the 50 Preservation Tasks. The same 150 tasks are used for every system. Table 9 reports the complete results, with NoMem included as a reference baseline. To control the cost of this large-scale evaluation, each of the 29 systems is evaluated with a single run, whereas the NoMem baseline is averaged over three runs and reported as mean and sample standard deviation.

The remaining metrics follow the definitions in Appendix H. Resolved and File Recall are task-level rates, F2P and P2P are micro-averaged over their corresponding test cases, and the Tokens and Steps columns report per-task means over the same tasks. Systems are ranked primarily by Resolved, which measures final task-solving success. Ties are broken in turn by F2P, File Recall, and P2P: F2P captures partial repair progress on issue-specific tests, File Recall reflects whether the agent reaches the reference files, and P2P measures preservation of pre-existing passing behavior. If two systems tie on all four performance metrics, we prefer the one with lower solving cost. We compare input tokens, output tokens, and agent steps in that order.

Table 9: Evaluation of 29 memory systems under the Relevant History scenario on the same 150 base tasks.

Memory system	Task performance				Solving cost		
	Resolved (%)	File Recall (%)	F2P (%)	P2P (%)	Tokens In (M)	Tokens Out (K)	Steps
causal-memory (JingxuanC, 2026)	52.67	81.56	79.84	99.47	6.18	56.50	186.35
ChronoHybridMem (Tin11Mn, 2026)	52.00	83.34	81.29	99.42	6.62	59.31	186.68
Memory Bear (Wen et al., 2026)	52.00	79.90	80.50	99.46	6.62	55.95	180.39
TencentDB Agent Memory (Tencent Cloud, 2026)	51.33	82.23	80.76	99.45	5.86	55.94	184.69
TraceMem (mhw-mathcode, 2026)	51.33	81.40	80.63	99.44	6.52	56.16	179.07
LightMem v2.0 (Fang et al., 2026)	51.33	82.79	80.50	99.44	5.44	51.16	180.77
picoAgent (Chusi-Truth, 2026)	51.33	81.73	80.24	99.44	5.91	53.42	172.89
memoria (FairmeHIT, 2026)	51.33	83.51	79.97	99.44	6.46	56.57	186.88
Hindsight (Latimer et al., 2026)	51.33	81.87	79.91	99.44	6.52	54.13	178.22
Cognee (Markovic et al., 2025)	50.67	84.12	80.11	99.42	6.54	58.77	189.65

Continued on the next page

Table 9 continued from the previous page

Memory system	Task performance				Solving cost		
	Resolved (%)	File Recall (%)	F2P (%)	P2P (%)	Tokens In (M)	Tokens Out (K)	Steps
Aranea-Agents (AmarsDing, 2026)	50.67	83.20	80.11	99.25	6.55	54.73	181.82
Infini Memory (Ji et al., 2026)	50.67	82.84	80.11	99.42	6.32	59.09	185.58
SQLite FTS Baseline (FantaSmallhamster, 2026)	50.00	81.81	80.37	99.43	5.29	50.85	176.49
ReFind (Li et al., 2026)	50.00	81.67	80.37	99.43	6.74	58.85	194.23
LightMem (Fang et al., 2026)	50.00	83.90	79.71	99.45	5.87	54.84	182.04
MemPalace (Jovovich & Sigman, 2026)	49.33	83.62	79.97	99.46	7.04	58.91	189.95
TencentDB Agent Memory (4o-mini) (Tencent Cloud, 2026)	49.33	83.09	79.71	99.43	5.81	55.30	185.96
EverMemOS (Hu et al., 2026)	48.67	80.79	79.45	99.46	6.95	59.42	185.35
Chronicle Memory (simple-boy, 2026)	48.00	80.51	79.84	99.44	5.84	54.11	182.47
openProustian Agent Memory (openProustian, 2026)	48.00	81.95	79.45	99.41	6.36	56.11	180.93
RecMem (Dai et al., 2026)	48.00	80.40	79.31	99.44	7.01	62.34	194.99
SimpleMem (Liu et al., 2026a)	48.00	82.34	79.05	99.47	6.53	54.72	176.94
NoMem (3 runs)	**47.78** $\pm$**1.54**	**84.03** $\pm$**2.30**	**79.67** $\pm$**0.53**	**99.47** $\pm$**0.02**	**5.92** $\pm$**0.14**	**54.78** $\pm$**3.09**	**176.93** $\pm$**3.73**
Mem517 (CodeWater517, 2026)	47.33	81.51	80.11	99.43	6.69	59.78	188.01
MemFusion (huangpengtao00-dotcom, 2026)	47.33	82.73	79.71	99.36	6.36	57.93	183.64
MemMachine (Wang et al., 2026b)	47.33	81.23	79.18	99.37	6.62	57.68	191.59
Hybrid Memory Search (cydd-1972, 2026)	47.33	82.56	78.79	99.43	6.43	58.39	193.92
EverOS (Hu et al., 2026)	46.67	80.09	78.79	99.38	6.40	55.39	184.16
ActiveMemoryIndex (linxuhao, 2026)	46.67	79.95	78.52	99.47	6.08	54.77	180.20
Mi-Memory (Liu et al., 2026b)	46.00	78.95	78.79	99.44	5.47	55.29	177.15

G MEMORY SYSTEM MECHANISMS

This appendix compares how five memory systems represent an agent's solving history and retrieve relevant memories as context for Claude Code on later tasks. Table 10 summarizes the implementations, internal models, and retrieval configurations used in our evaluation.

Example task. The agent is asked to add a keep_attrs option to xarray.where, allowing users to choose whether to preserve array metadata such as units and descriptions. We use task pydata__xarray-4687 from xarray, a data-processing library. Its where operation retains or replaces array values according to a condition. At the time of this task, the where methods on arrays and datasets (DataArray.where and Dataset.where) automatically preserved metadata. However, the standalone function xarray.where offered no option to control metadata preservation.

Solving trajectory. Figure 8 shows a segment of the agent's trajectory while solving this task, comprising five events:

- e_1: **Inspect the existing behavior.** The agent inspects the existing where methods and confirms that they automatically preserve metadata.
- e_2: **Add the option and modify the implementation.** While adding the new option, the agent also modifies the existing methods' internal implementation so that metadata preservation depends on the supplied keep_attrs value.
- e_3: **Detect a regression through testing.** The initial change breaks the existing default behavior. When keep_attrs is left unspecified, metadata is unexpectedly lost, causing a test to fail.
- e_4: **Correct the default handling.** The agent corrects how an unspecified keep_attrs value is handled, restoring metadata preservation by default.

Table 10: Memory-system configurations used in our evaluation.

System	Implementation	Memory LLM	Embedding Model	Retrieval Configuration
A-Mem	`ceffb860f0712`; MCP 0.2.6 (`dce0ac62cba5`)	DeepSeek-V4-Flash	all-MiniLM-L6-v2 (SentenceTransformers, Chroma)	`k=5` for standard and agentic search; `k=10` for temporal search
Mem0	Plugin 0.2.9 (`366945965df4`); official API	Provider-managed	Provider-managed	Official API defaults
MemOS	MCP 1.1.2; official API	Provider-managed	Provider-managed	`memory_limit_number=6`; retry with `relativity=0` when no result is returned
Supermemory	Plugin 0.0.13 (`193290b5ccba`); official API	Provider-managed	Provider-managed	`limit=5`; `searchMode=hybrid`; `threshold=0.6`
Claude-Mem	v12.1.1 (`0b6997e3d03e`)	DeepSeek-V4-Flash	all-MiniLM-L6-v2 (ONNX, 384-d)	`search limit=20`; 100 semantic candidates

Provider-managed denotes components managed by the official API for which no fixed model identifier is exposed to the client.

e_5: **Verify the repair.** The agent reruns the same set of where tests, and all five tests pass.

Later retrieval. When Claude Code encounters a related metadata-preservation issue in a later task, this history can provide the earlier failure's cause, repair, and validation. To illustrate memory storage and retrieval, we use a shared example query, q: "How should where preserve attributes?" The following five system figures show possible memory representations of this history. They also illustrate how each system retrieves relevant context for Claude Code in response to q. The e_i labels identify source events. A memory may integrate information from multiple events.

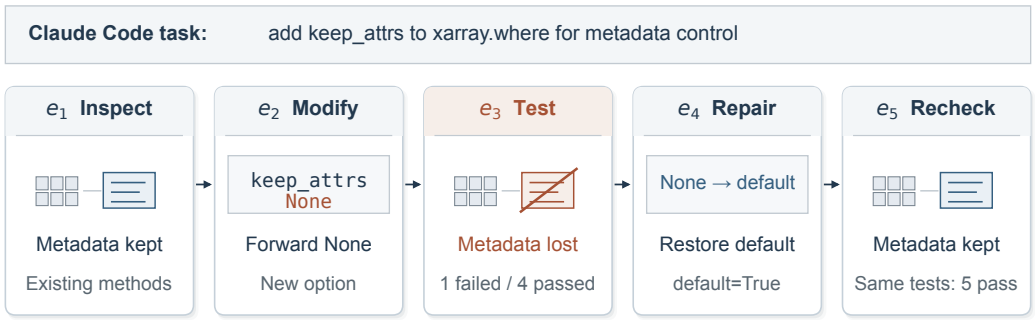


Figure 8: **Shared trajectory.** An initial change breaks default metadata preservation in the existing methods. The same five tests expose the regression and pass after the repair.

G.1 A-Mem: Evolving Note Graph

Memory construction. A-Mem (Xu et al., 2025) organizes reusable knowledge from Claude Code as linked notes. In this integration, a session-start hook injects instructions to search memory before each task and save new findings through MCP during code exploration, debugging, or repair. The add_memory_note description covers architectural relations, code locations, errors and solutions, recurring implementation patterns, and configuration requirements. It excludes trivial or obvious information, findings already covered by memory, and observations without reusable insights.

In Step 1 of Figure 9, Claude selects new findings from its current task context and summarizes them as a self-contained note body. It submits this text as the tool's content argument. Claude authors the body from its exploration, edits, and test results. A-Mem receives this text rather than extracting it from the full trajectory. Here, a note can combine metadata loss (e_3), repair (e_4), and a passing recheck (e_5) to record the failure conditions, solution, and validation.

In Step 2, A-Mem creates a note containing this body, a unique identifier, and a timestamp. Its LLM extracts keywords for key concepts, assigns semantic tags, and generates a one-sentence context description of the topic, main points, and purpose. These fields form the note's semantic attributes. A text encoder embeds the concatenated body and attributes for indexing in the vector database.

In Step 3, A-Mem encodes the new note's body with the same text encoder. It compares this query vector with stored embeddings of note bodies and semantic attributes to retrieve the nearest candidates. An LLM compares the candidates' bodies and attributes with the new note to select links. A-Mem records the selected identifiers as links. It can also update existing context descriptions and tags while retaining the original bodies. Here, the repair note can link to the note from e_1. The earlier note's description can then explain how to handle an unspecified keep_attrs value.

Memory retrieval. The hook prompts Claude to retrieve memory, but does not query the database itself. Claude formulates query q from the current task and submits it through search_memories. In Step 4, A-Mem encodes q with the same encoder and returns the top-k nearest notes as lightweight previews: identifiers, context, keywords, tags, and distance scores. Claude assesses their relevance and calls read_memory_note to read selected bodies and links by identifier. For broader context, search_memories_agentic extends the retrieval pool with linked neighbors before returning results for further reading. This combination of semantic search and selective reading lets Claude locate prior experience, then inspect its failure conditions, repair, and validation.

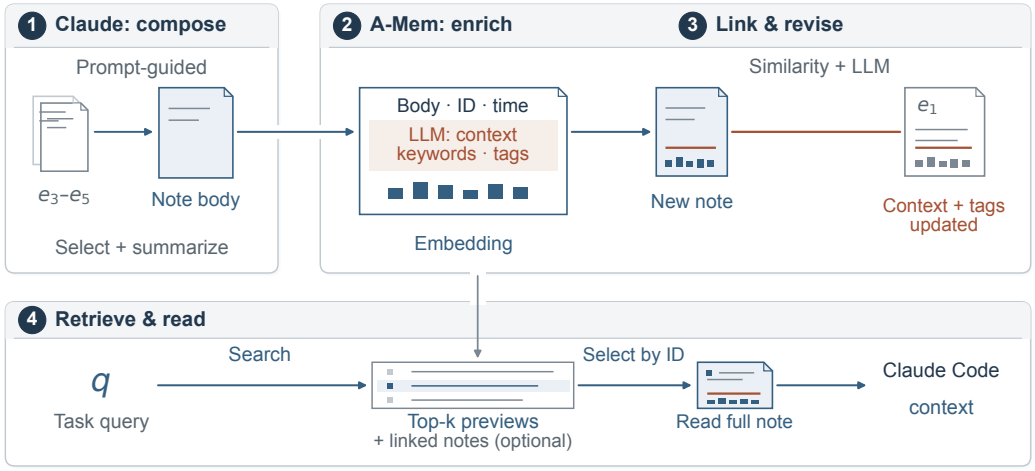


Figure 9: **A-Mem with Claude Code.** Claude composes a note from e_3–e_5, which A-Mem enriches and links. Retrieval combines semantic previews, selective reading, and optional link expansion.

G.2 Mem0: Persistent Repository Memory

Memory construction. Mem0 (Chhikara et al., 2025) distills an agent's interaction history into persistent memory for retrieval across sessions. In Step 1 of Figure 10, Claude Code lifecycle hooks apply predefined rules to collect user prompts, agent responses, and selected execution evidence, including modified file paths and bounded diagnostics for failed commands. The plugin accumulates this material locally, and a background worker submits it to Mem0 in message batches. The evidence payload excludes full file contents and general tool output.

Step 2 illustrates the current open-source text-memory extraction pipeline. The submitted user and assistant messages, together with the selected evidence, provide the primary input. Recent messages within the session scope supply context and help resolve references. Before extraction, Mem0 uses the current messages to retrieve semantically related records within the same scope, helping the LLM distinguish new information from existing knowledge. A dedicated extraction prompt then guides an LLM to retain established repository behavior, relevant implementation locations, important constraints, and reusable solutions. It excludes transient exploration, unaccepted proposals, and standalone test results. The LLM produces concise memory entries, or none when no useful in-

formation has been established. For e_3–e_5, an entry can state that where should preserve attributes when keep_attrs is unspecified, resolving the default through _get_keep_attrs(default=True).

Step 3 persists the extracted memories. Mem0 computes their embeddings, filters duplicate entries, and stores each record with its text, identifier, timestamps, scope, and metadata. The where rule thus becomes a record associated with the xarray repository. Its text captures reusable repository knowledge, its embedding supports similarity search, and its scope limits retrieval and reuse to relevant tasks.

Memory retrieval. Step 4 returns persistent memories to Claude Code during subsequent tasks. The first substantive user prompt in a new session initiates a search within the applicable repository and user scope. During later execution, the agent can also call search_memories with a natural-language query about repository behavior, an error, or an implementation detail. Given q, the platform combines similarity between query and memory embeddings with keyword and entity matching to rank eligible records. Temporal metadata can further restrict results through filters. The search returns the highest-ranked memories with relevance scores and metadata, and the plugin provides selected memory text to the agent as compact context.

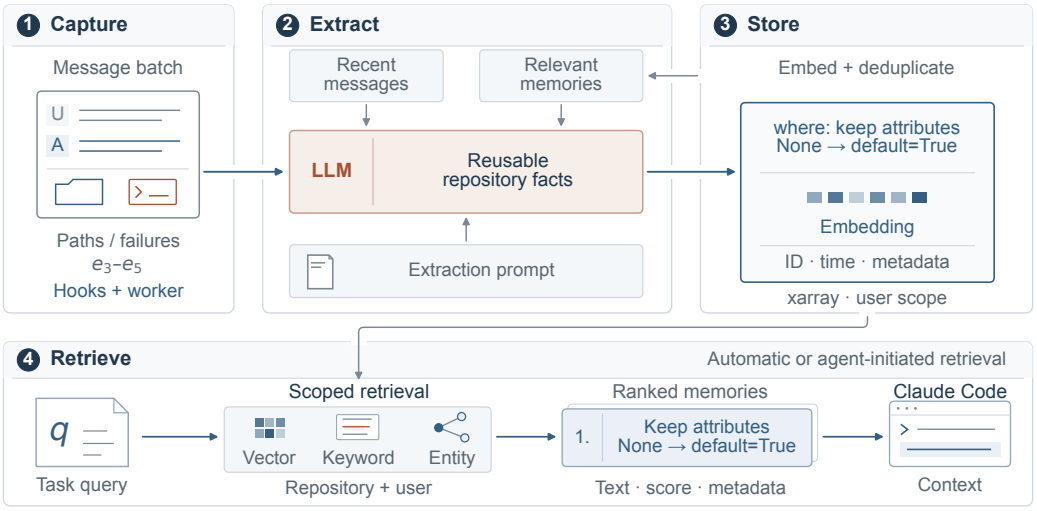


Figure 10: **Mem0 with Claude Code.** Selected interaction evidence is distilled into persistent memories for scoped retrieval.

G.3 MemOS: Text Memories in MemCube

Memory construction. MemCube is the unified memory abstraction proposed by MemOS and implemented in its open-source framework as a modular container for textual, activation, and parameter memories (Li et al., 2025). It supports the organization and persistence of these memory types. Figure 11 follows the textual path, which converts Claude Code's past interactions into structured records for storage and later reuse.

In Step 1, Claude Code submits past interactions through the MCP tool add_message. The interface associates an ordered messages array with a conversation_id before forwarding it to the service. Each message carries role and content, with chat_time supplied by the caller or filled in by the interface. User and assistant exchanges associated with e_3–e_5 thus provide the input to memory extraction.

Step 2 follows MemReader's LLM extraction path in the open-source framework. Consecutive messages are grouped into context windows bounded by a token limit. Adjacent windows may overlap to preserve context across turns. For each window, an LLM extracts reusable facts, decisions, and experience as structured memory items, including a memory statement (value), a short identifying key, a memory type, and tags. Window background and supporting messages are retained. For e_3–e_5,

the extracted statement can specify that the existing where methods should resolve keep_attrs=None using _get_keep_attrs(default=True) to preserve attributes by default.

Step 3 shows how these items are represented and persisted in MemCube's text store. When preparing each record, the embedding model encodes only value for semantic retrieval. The key, memory type, tags, background, and source messages are retained as metadata, together with user, session, and time information. The record therefore preserves both the text used for semantic matching and the contextual attributes stored alongside it.

Memory retrieval. In subsequent tasks, Claude Code formulates q from the current problem and calls search_memory. Step 4 illustrates query encoding, semantic matching against embeddings derived from the stored value text, and candidate filtering and ranking. Metadata filters constrain the eligible records, while semantic relevance guides selection. When attribute handling in where becomes relevant again, the experience extracted from e_3–e_5 can be returned to Claude Code as context, providing the earlier behavioral constraint and implementation rule.

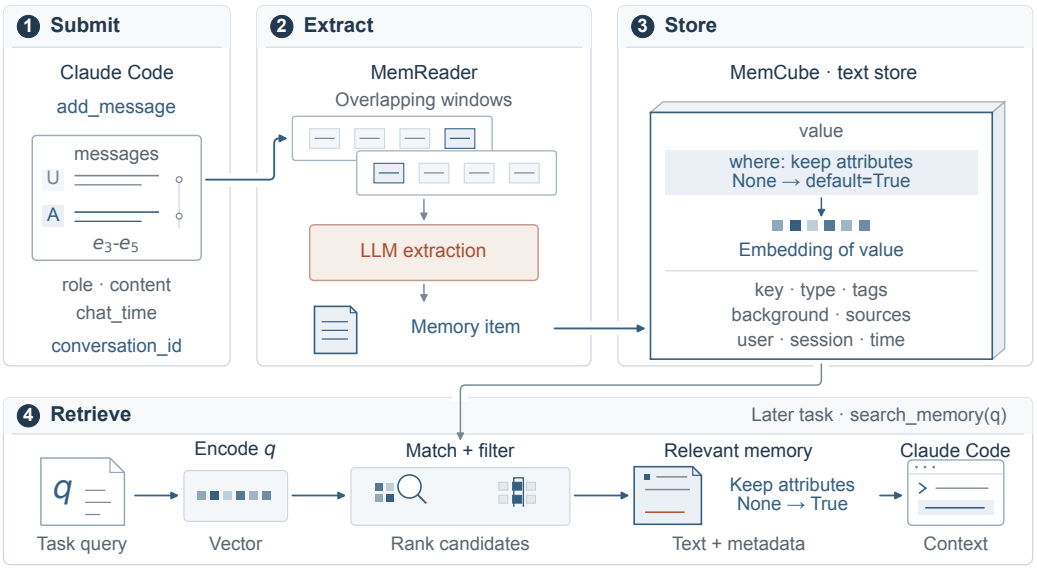


Figure 11: **MemOS with Claude Code.** MemReader extracts textual memories into MemCube for later semantic retrieval.

G.4 SUPERMEMORY: DOCUMENTS, MEMORY GRAPHS, AND PROFILES

Memory construction. Supermemory stores Claude Code interactions as source documents and distills them into retrievable atomic memories (Daga et al., 2026). In Step 1 of Figure 12, the Stop hook reads the session transcript after Claude completes a response, selecting entries added since the last successful submission. It preserves the order of user messages, assistant responses, and configured tool calls and results, then submits this increment as document content with project, session, and time information.

In Step 2, Supermemory partitions documents into coherent chunks and indexes their embeddings. Extraction instructions supplied by the plugin prioritize reusable decisions, conventions, workflows, and lessons. Atomic memories isolate individual facts while resolving the objects, operations, and parameters needed to interpret them. Each memory retains links to its supporting chunks and an embedding for retrieval. For example, e_4 can yield the rule for resolving keep_attrs=None in the existing where methods, while e_5 records the corresponding focused-test result. Memory generation may continue after chunks become searchable.

In Step 3, these facts join the memory graph with provenance and temporal metadata. An *updates* relation makes a corrected fact current while retaining the earlier version as history. An *extends* relation adds compatible details while both memories remain valid. A *derives* relation links an

inferred conclusion to its premises. Thus, e_4 can extend e_1's attribute-preservation rule, and e_4 together with e_5 can support the conclusion that the repair passed the selected checks. The service also maintains a profile for the project, whose static component summarizes stable facts and whose dynamic component captures recent activity and evolving context.

Memory retrieval. At session start, the SessionStart hook retrieves the project's static and dynamic profile components and supplies them to Claude Code. For subsequent substantive prompts, UserPromptSubmit requests matching content using the current prompt as q and the project's containerTag. In Step 4, the query is embedded for semantic matching. The service can search memory embeddings, document-chunk embeddings, or both, with metadata constraints and optional relation expansion or reranking. The hook filters low-scoring matches, removes text already recalled during the session, and injects concise context into Claude Code. For q, matching memory or source text can expose e_4's default-resolution rule and e_5's focused verification.

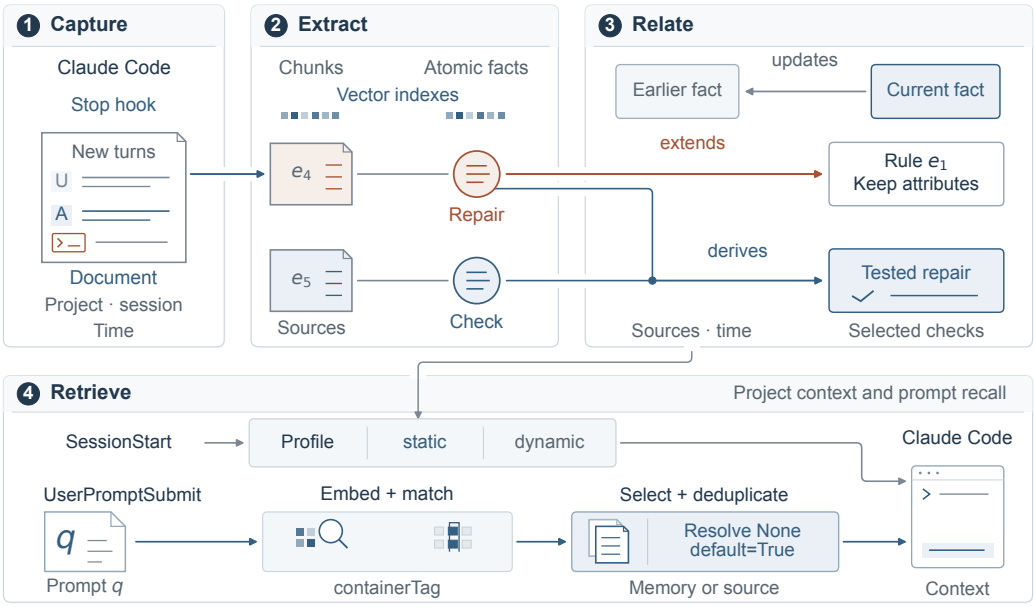


Figure 12: **Supermemory with Claude Code.** Documents yield linked atomic memories for project profiles and subsequent retrieval.

G.5 Claude-Mem: Observations and Progress Summaries

Memory construction. Claude-Mem captures coding activity through Claude Code's lifecycle hooks and compresses it into observations and progress summaries in a background worker (Newman, 2025). In Step 1 of Figure 13, the user request establishes task context. After each tool call, PostToolUse forwards the tool name, arguments, result, and working directory to a background queue. Code reads, edits, and test runs thus supply raw activity records for subsequent extraction.

In Step 2, the worker's Claude Agent SDK backend invokes an LLM with the user request as context and the current tool's name, input, and result as evidence. The extraction prompt permits skipping uninformative activity or producing multiple observations. Each observation contains a type, title, facts, an explanatory narrative, concepts, and files read or modified. For example, observations can record e_3's metadata loss, e_4's default-resolution repair, and e_5's focused-test result.

Step 3 maintains a complementary session-level record. The Stop hook requests a progress summary using the worker's accumulated task context and Claude's latest response. The LLM summarizes the request, investigation, lessons, completed work, next steps, and notes, allowing successive checkpoints within one session. Observations and summaries are stored in SQLite with project, session, and time metadata. When Chroma indexing is enabled, an observation's narrative and individual facts receive separate vector representations linked to its SQLite ID.

Memory retrieval. At session start, SessionStart uses recent observations and summaries from the project's SQLite records to construct a compact chronological overview for Claude Code. Selection follows project, time, and configured context filters without requiring semantic search. The overview exposes earlier work and progress before the agent requests further details.

In Step 4, Claude can query history through MCP tools. For observation retrieval, search returns compact candidates with IDs, titles, types, and timestamps. With Chroma enabled, query similarity matches indexed narratives and facts, while metadata filters constrain project, type, concepts, files, or date. Claude can then use get_observations to read complete facts, narratives, and file information for selected IDs, or timeline to inspect neighboring events around an anchor ID or query. For q, this can locate e_4's repair and place it alongside the preceding failure and e_5's verification.

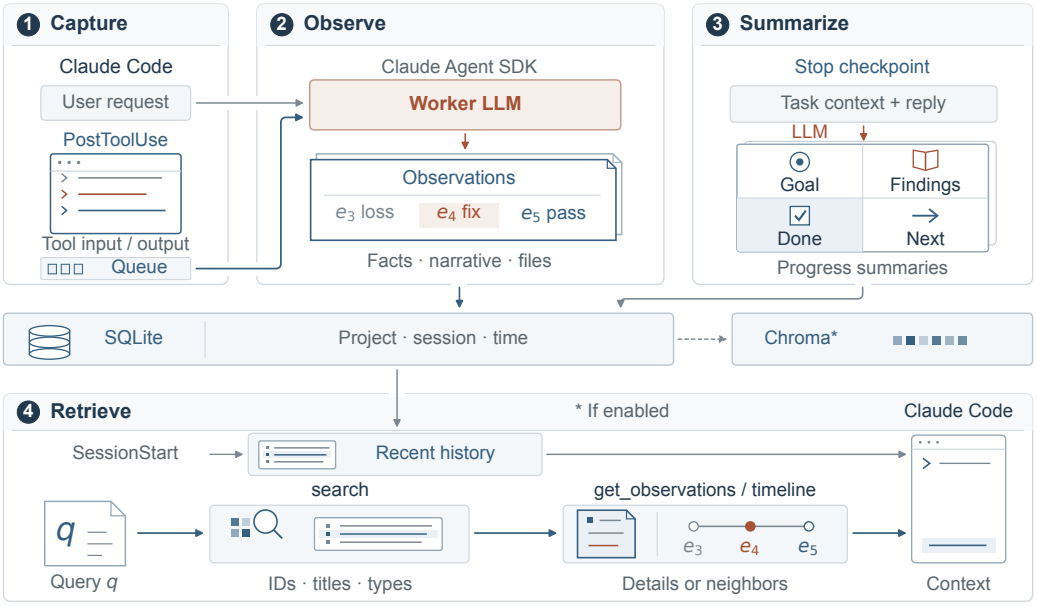


Figure 13: **Claude-Mem with Claude Code.** Observations and progress summaries support startup context and selective retrieval.

H Metric Definitions

This appendix gives the full definitions of the evaluation metrics used in the paper. Let N denote the number of base tasks in an evaluation condition.

H.1 Main Evaluation Metrics

Resolved. Resolved measures the fraction of base tasks solved according to the official task verifier. For a single run,

$$\text{Resolved} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}[V_i = 1], \tag{1}$$

where $V_i = 1$ if the agent's final patch for task i passes the verifier and $V_i = 0$ otherwise. Resolved is therefore an all-or-nothing measure of task-solving success.

Rescue and Harm. The change in Resolved relative to NoMem captures only the net effect of memory. Rescue and Harm separate this change into positive and negative effects: Rescue captures cases where NoMem fails but the condition with memory succeeds, whereas Harm captures the reverse.

Each condition is evaluated with three independent runs. For task i, let $k_i \in \{0, 1, 2, 3\}$ denote the number of successful NoMem runs and $m_i \in \{0, 1, 2, 3\}$ the number of successful runs under the

memory condition. Because runs across conditions have no natural one-to-one correspondence, we consider all 3×3 cross-condition run pairs for each task. Among these nine pairs, $(3 - k_i)m_i$ have a failed NoMem run and a successful memory run, while $k_i(3 - m_i)$ exhibit the reverse. Rescue and Harm are therefore defined as

$$\text{Rescue} = \frac{1}{9N}\sum_{i=1}^{N}(3 - k_i)m_i, \qquad \text{Harm} = \frac{1}{9N}\sum_{i=1}^{N}k_i(3 - m_i). \tag{2}$$

Their difference exactly recovers the change in Resolved relative to NoMem:

$$\text{Rescue} - \text{Harm} = \frac{1}{3N}\sum_{i=1}^{N}(m_i - k_i) = \text{Resolved}_{\text{system}} - \text{Resolved}_{\text{NoMem}}. \tag{3}$$

For Rescue and Harm, we compute 95% confidence intervals by bootstrapping over tasks. Each bootstrap replicate samples N tasks with replacement while retaining all three outcomes under each condition, and recomputes Rescue and Harm using the same 3×3 cross-condition pairing. We repeat this process $10{,}000$ times and report the 2.5th and 97.5th percentiles of the resulting bootstrap distributions.

Input Tokens and Output Tokens. Input Tokens and Output Tokens measure the token cost of base-task solving. They exclude the tokens consumed while memory systems build memory from historical tasks. For task i, let T_i^{in} and T_i^{out} denote the total numbers of input and output tokens consumed during solving. We compute the per-task means

$$\text{InputTokens} = \frac{1}{N}\sum_{i=1}^{N}T_i^{\text{in}}, \qquad \text{OutputTokens} = \frac{1}{N}\sum_{i=1}^{N}T_i^{\text{out}}. \tag{4}$$

Agent Steps. Agent Steps measures the number of model-response turns produced by the agent during base-task solving. Each model response is counted as one step and may contain reasoning, textual output, or tool calls; multiple tool calls within the same response are counted as a single step. Let S_i denote the total number of steps for task i. We compute

$$\text{AgentSteps} = \frac{1}{N}\sum_{i=1}^{N}S_i. \tag{5}$$

H.2 Partial Repair Metrics

F2P and P2P. Resolved credits a task only when all required tests pass and therefore does not capture partial repair progress. For task i, let F_i denote its fail-to-pass tests, which fail on the original repository state and should pass after a correct repair, and let P_i denote its pass-to-pass tests, which pass on the original state and should remain passing. Both sets come from the official task specification. Let $f_i \le |F_i|$ denote the number of fail-to-pass tests that pass after the agent's final patch is applied, and let $p_i \le |P_i|$ denote the number of pass-to-pass tests that remain passing. We micro-average over tests:

$$\text{F2P} = \frac{\sum_{i=1}^{N} f_i}{\sum_{i=1}^{N} |F_i|}, \qquad \text{P2P} = \frac{\sum_{i=1}^{N} p_i}{\sum_{i=1}^{N} |P_i|}. \tag{6}$$

F2P measures how much of the requested behavior is implemented, whereas P2P measures how well previously passing behavior is preserved.

H.3 Gold-Patch Alignment Metrics

Resolved, F2P, and P2P measure the agent's functional performance through test outcomes. As a complement, we further compare the agent's final patch with the reference solution patch, measuring their agreement at the file, edit-location, and edit-content levels.

Patch Sets. For task i, let G_i denote the set of files modified by the reference solution patch, excluding the test patch, and let C_i^r denote the set of files modified by the agent's final patch in run

r. For all three metrics, we first average over tasks within each run and then report the mean and sample standard deviation across three runs.

File Recall. File Recall measures how many files modified by the reference solution patch are covered by the agent. It is defined as the fraction of reference files that also appear in the agent's modified-file set:

$$\mathrm{FileRecall}_i^r = \frac{|C_i^r \cap G_i|}{|G_i|}. \tag{7}$$

Line Recall. Line Recall further measures whether the agent edits the locations modified by the reference solution patch. We treat each added or deleted line in a unified diff as one edit operation and ignore context lines; a line replacement corresponds to one deletion and one addition. A deletion is represented as

$$(\mathrm{file}, \texttt{delete}, \ell, 0),$$

where ℓ is the line number in the pre-edit file. An addition is represented as

$$(\mathrm{file}, \texttt{add}, a, o),$$

where a is its insertion anchor in the pre-edit file and o is its offset among additions sharing that anchor.

Let L_i and A_i^r denote the operation sets of the reference solution patch and the agent's final patch, respectively. Line Recall is defined as the fraction of reference operations whose locations are matched by the agent:

$$\mathrm{LineRecall}_i^r = \frac{|A_i^r \cap L_i|}{|L_i|}. \tag{8}$$

An operation is matched only when the file, edit direction, location, and offset all agree. Line Recall does not compare the changed text.

Edit Recall. Edit Recall further requires the changed text to match at the same location. We extend each operation to

$$(\mathrm{file}, \texttt{add}/\texttt{delete}, \text{anchor or line}, \text{offset}, \mathrm{rstrip}(t)),$$

where t is the changed-line text and rstrip removes trailing whitespace only, preserving leading indentation, internal whitespace, and case, with no other normalization.

Let E_i and D_i^r denote the resulting operation sets for the reference solution patch and the agent's final patch, respectively. Edit Recall is defined as the fraction of reference operations whose location and text are both matched exactly:

$$\mathrm{EditRecall}_i^r = \frac{|D_i^r \cap E_i|}{|E_i|}. \tag{9}$$

An operation is matched only when the entire tuple agrees exactly.