MEMTRACE: STATE-CONSISTENT MEMORY FOR LONG-HORIZON CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As coding agents take on long-horizon software evolution tasks spanning multiple files and stages, longer execution trajectories introduce two coupled challenges: (1) accumulated histories strain context budgets, and (2) repository changes can invalidate earlier execution evidence. Existing approaches address these challenges through techniques like larger context windows, compression, retrieval, or repository representations, but often fail to reconstruct a consistent task state after a context refresh or verify whether recalled evidence remains valid. Thus, we introduce **MemTrace**, a provenance-aware memory system that preserves execution history and aligns its reuse with the evolving task (e.g., iterative cross-file repair) and repository state. MemTrace stores history as immutable *Memory Traces* anchored to key information (e.g., files, symbols, tests), and organizes their execution order and dependencies in a *Memory Trace Graph*. When context is constrained, working memory retains only compact *Memory Anchors*, from which the agent can reconstruct the latest execution state and locate evidence relevant to its next action. Before restoring historical evidence, MemTrace checks its validity against the current repository state and retrieves only what the next action requires. Across three complementary long-horizon coding benchmarks, MemTrace consistently outperforms all fully evaluated baselines under the same backbone and harness, improving DeepSWE pass@1 by 21.2 points, SWE-EVO Resolved Rate by 4.4 points, and SWE-Milestone Score by 17.8 points under Codex CLI.

1 INTRODUCTION

Coding agents are moving from isolated bug fixing toward cross-file, multi-stage software maintenance and development, which requires iterative code inspection, coordinated edits, testing, and revision (Jiménez et al., 2024; Thai et al., 2025; Huang et al., 2026). As execution lengthens, growing interaction history increases context pressure, while ongoing repository changes can make earlier observations, test outcomes, and decisions no longer applicable to the current state. This challenge already appears in deployed coding agents: systems such as Codex CLI (Bolin, 2026) use context compaction to continue long-running tasks across context boundaries. Such mechanisms preserve execution continuity, but continuity alone does not determine whether recovered evidence still applies after the repository has changed. Long-horizon execution therefore requires the agent to maintain an accurate representation of task progress: what has been completed, what remains unresolved, and which prior evidence still applies. *Task-state drift* arises when this representation no longer matches the current task and repository state. The problem becomes particularly important after a context refresh, when the agent must reconstruct prior progress and recover evidence that remains applicable before continuing the unfinished task.

Existing approaches mitigate different parts of this problem. Longer context windows accommodate more interaction history, while prompt compression condenses earlier context and observation masking selectively removes prior observations and tool outputs. External-memory systems offload historical information for later retrieval (Packer et al., 2023; Lee et al., 2024; Chen et al., 2023), while repository graphs represent files, symbols, and dependencies for code localization (Liu et al., 2025b; Ouyang et al., 2025). These approaches improve history access and code localization, but access alone does not determine which historical evidence still supports the task state being resumed.

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

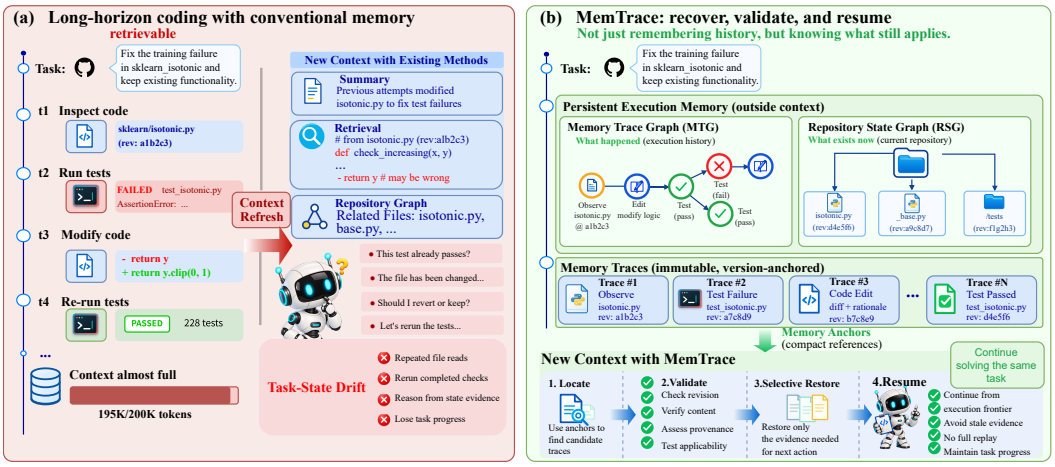


Figure 1: **Task-state recovery across context refreshes.** (a) Existing approaches use summaries, retrieved history, or repository structure, but may lose task progress or reuse stale evidence. (b) MemTrace reconstructs the latest execution state, validates historical evidence against the current repository, and restores only what is needed to resume the task.

The remaining challenge is to recover the current task state, not merely historical content. As illustrated in Figure 1(a), a refreshed context may retrieve an earlier test failure even after a repair and a passing rerun. Without the relation to these later events, the failure may be mistaken for an unresolved issue. We call this mismatch between recovered history and the current task and repository state *state-memory misalignment*. Addressing it requires reconstructing the current task state and identifying which prior evidence remains valid in the current repository and relevant to the next action.

To address this gap, we introduce **MemTrace**, a provenance-aware memory system that casts long-horizon coding memory as a state-alignment problem rather than mere history retention. Completed execution is sealed into immutable Memory Traces with explicit repository bindings and organized through two complementary views. A Memory Trace Graph (MTG) encodes *depend*, *verify*, *correct*, and *supersede* relations, enabling relation-aware traversal from the current execution frontier to evidence that establishes, revises, or invalidates its state. A Repository State Graph (RSG) maintains the current files, symbols, tests, and their structural relations. Trace-to-entity bindings project candidate traces onto this evolving state to determine whether their referenced evidence remains applicable. Under context pressure, trace bodies are offloaded while Memory Anchors preserve addressability. After a context refresh, MemTrace reconstructs the frontier through the MTG, checks candidate evidence against the RSG, filters stale or superseded traces, and loads only the validated evidence required for the next action back into working context (Figure 1(b)).

To assess MemTrace's effectiveness in long-horizon coding, we evaluate it on three complementary benchmarks covering repository-level engineering, multi-step software evolution, and continuous milestone-based development: DeepSWE, SWE-EVO, and SWE-Milestone. MemTrace consistently outperforms fully evaluated baselines across agent harnesses, while achieving a 46.4% overall score on SWE-Milestone under Codex. Ablations show that persistence alone is insufficient: both structured execution memory and repository-state grounding are important for maintaining state-consistent memory over long-horizon execution.

Contributions. This paper makes the following contributions.
(1) We introduce a state-bound execution-memory representation for long-horizon tasks that preserves historical evidence with its original repository bindings and records how later execution supports, revises, or supersedes earlier evidence.
(2) We present MemTrace, which links the MTG and RSG through shared repository references to reconstruct task state and selectively restore applicable historical evidence after a context refresh.
(3) On three long-horizon coding benchmarks, MemTrace consistently outperforms the evaluated baselines under matched backbone and agent harness, while ablations confirm the complementary roles of persistent storage, structured execution memory, and repository-state grounding.

2 Related Work

Context management for long-horizon agents. Long-horizon agents must manage growing interaction histories within a finite context window. LLMLingua and LongLLMLingua compress prompts (Jiang et al., 2023; 2024), while observation masking and SWE-Pruner reduce the tool content retained in context (Lindenbauer et al., 2025; Wang et al., 2026a). MemGPT manages information across working and external memory (Packer et al., 2023); ReadAgent, MemWalker, and RAPTOR use summaries and retrieval to access long documents (Lee et al., 2024; Chen et al., 2023; Sarthi et al., 2024). For agent trajectories, Context as a Tool and Context-Folding organize history around tasks or subtrajectories (Liu et al., 2025a; Sun et al., 2025), while Self-GC and Scroll retain recoverable history outside the active context (Hao et al., 2026; Lin et al., 2026). These approaches make longer histories manageable. However, recovering historical content does not establish whether it still supports the task after later edits or tests. MemTrace couples history retrieval with task-state recovery, using relations among execution records to determine what has been completed and what remains unresolved.

Agent memory and execution-state management. Agent-memory systems retain experiences and reflections for later decisions (Park et al., 2023; Shinn et al., 2023). Mem0 updates salient memories, A-MEM links related memory notes, and Zep tracks the temporal validity of facts while retaining superseded information (Chhikara et al., 2025; Xu et al., 2025; Rasmussen et al., 2025). Beyond storing reusable knowledge, recent systems also use memory to maintain an explicit view of ongoing task progress. MAGE maintains a hierarchical state tree with summary validation and rollback (Chen et al., 2026), while Ledger uses change counters to track observation freshness and reduce redundant actions (Wang et al., 2026b). These mechanisms help preserve execution state across long trajectories. In coding, however, maintaining task progress alone is insufficient: stored observations and test results may become stale as the repository changes. Recovering offloaded evidence therefore requires not only reconstructing what happened later, but also checking whether the evidence still applies to the current repository. MemTrace addresses this by binding immutable execution traces to repository state, recording verification, correction, and supersession relations in its Memory Trace Graph (MTG), and validating evidence before restoration.

Repository-level coding and code graphs. Repository-level coding systems coordinate localization, editing, and validation through agent–environment interaction or structured repair pipelines (Yang et al., 2024; Zhang et al., 2024; Xia et al., 2024). CodePlan further uses dependency analysis to plan coordinated changes across files (Bairi et al., 2024). CodexGraph, RepoGraph,

Table 1: Comparison of representative methods discussed in Related Work. ✓, △, and × denote explicit, partial, and no support under the criteria below.

Method	Context management	Execution state		Repository alignment	
	Recoverable history	Structured task state	Revision history	Code graph	Repo-aware evidence reuse
Observation Masking	×	×	×	×	×
MemGPT	✓	×	×	×	×
Context as a Tool	△	△	×	×	×
Self-GC	✓	×	×	×	×
Scroll	✓	△	△	×	×
Zep	✓	×	✓	×	×
MAGE	✓	✓	✓	×	×
Ledger	×	✓	△	×	✓
RepoGraph	×	×	×	✓	×
MemTrace	✓	✓	✓	✓	✓

Note. History: ✓ original records remain recoverable after leaving context; △ only summaries remain. Task state: △ implicit progress. Revisions: △ event order or staleness without explicit correction. Repo-aware reuse checks current code; Ledger requires still-visible outputs.

and LocAgent represent code entities and their relations to support repository navigation and localization (Liu et al., 2025b; Ouyang et al., 2025; Chen et al., 2025). These representations expose structural dependencies that are difficult to recover from isolated code snippets. However, code structure alone does not record which observations supported an edit, which tests verified it, or whether later execution replaced an earlier conclusion. MemTrace links its Repository State Graph (RSG) to the MTG through shared repository entities, allowing relevant execution evidence to be located and checked against the evolving codebase. Table 1 summarizes these differences across history retention, execution and repository structure, and evidence validity.

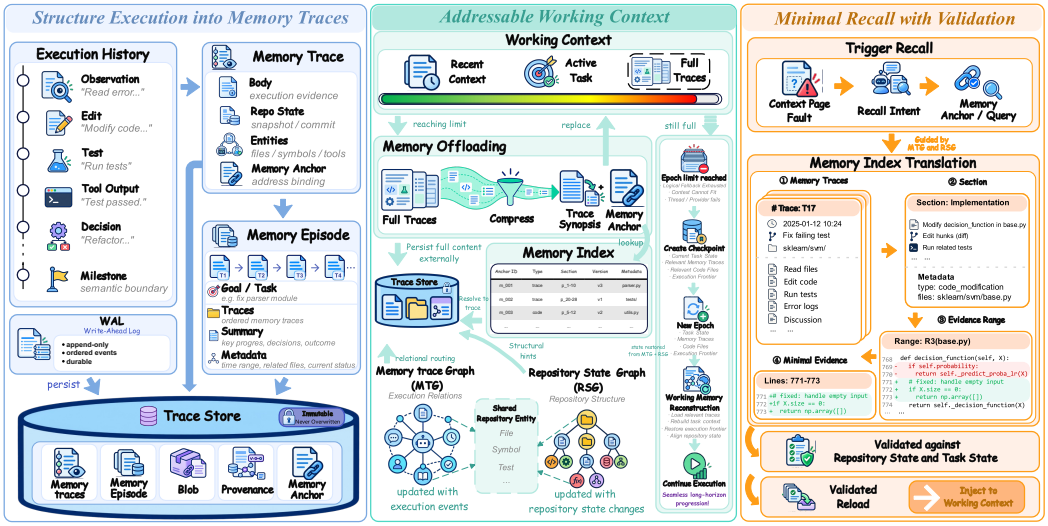


Figure 2: **Overview of MemTrace.** From left to right, MemTrace structures execution into state-bound traces, maintains an addressable working context, and restores only validated evidence needed to resume execution.

3 METHOD

MemTrace equips coding agents with persistent execution memory that survives context refreshes and supports evidence reuse as the repository evolves. It stores completed units of work as immutable *Memory Traces* and records their execution order and dependencies in a *Memory Trace Graph* (MTG). A complementary *Repository State Graph* (RSG) represents the current files, symbols, tests, and their structural relations. Linking traces to repository entities allows MemTrace to check whether historical evidence remains applicable. When working memory approaches its context limit, MemTrace offloads completed trace bodies, retaining compact *Memory Anchors* and lightweight *Trace Synopses* that reference the persistent store. After a context refresh, it reconstructs the execution frontier, validates historical evidence against the current repository, and restores only what the next action requires.

Workflow. Figure 2 demonstrates the three stages of MemTrace. At step t, MemTrace maintains the state $S_t = (R_t, m_t, \mathcal{H}_t, G_t^{\mathrm{M}}, G_t^{\mathrm{R}}, \Pi_t, C_t, \mathcal{A}_t)$. Here R_t is the current repository state, including working-tree edits, m_t records the execution frontier and unresolved work, and $\mathcal{H}_t$ is the set of immutable traces in the persistent Trace Store. G_t^{M} and G_t^{R} are the MTG and RSG; Π_t associates traces with repository entities; C_t is working memory; and $\mathcal{A}_t$ contains the anchors currently visible in it. The usable input-context budget is B, with space reserved for the next response. Instructions, task text, tool descriptions, checkpoint metadata, and restored evidence all count toward this budget. We next present state-bound Memory Traces, the MTG–RSG alignment through shared repository entities, and validated restoration after a context refresh.

3.1 STATE-BOUND MEMORY TRACES

Long execution histories consume context, while repository changes can invalidate earlier observations. To support compact retention and reliable reuse, MemTrace organizes history into immutable

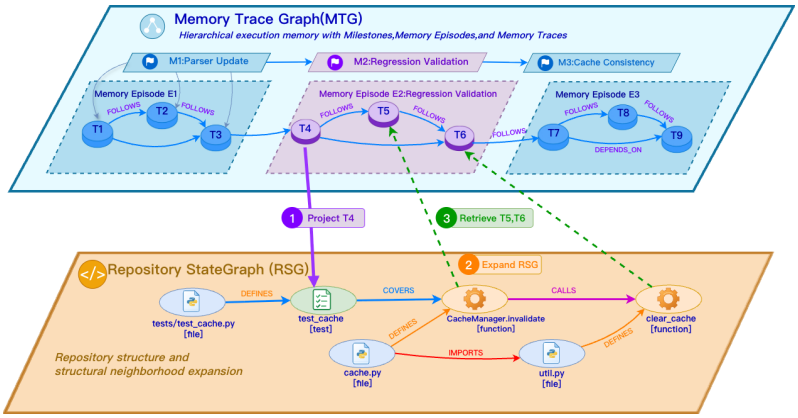


Figure 3: **MTG–RSG Projection and candidate localization.** Repository bindings connect execution traces to current repository entities, allowing MTG–RSG traversal to locate relevant traces for validated restoration.

Memory Traces, each binding a completed unit of execution evidence to the repository state in which it was produced. Trace construction and offloading proceed as follows.

Identify and seal reusable evidence. MemTrace seals a unit of work when it is complete and provides evidence that can support subsequent actions, such as an edit, test outcome, failure diagnosis, or correction. Each trace is represented as

$$\tau = \big(x_\tau,\ \rho_\tau,\ \mathcal{E}(\tau),\ \alpha_\tau\big),$$

where x_τ contains the execution evidence and its provenance; ρ_τ records the repository state at evidence creation; $\mathcal{E}(\tau)$ identifies the files, symbols, and tests referenced in the body; and α_τ is a compact *Memory Anchor* that provides a stable reference to the trace. Sealing adds τ to the persistent Trace Store $\mathcal{H}_t$ and its anchor to $\mathcal{A}_t$. A unit that exceeds the size of a single trace body is split into multiple traces, each with its own anchor.

Preserve provenance as the repository evolves. Once sealed, a trace is never modified. Subsequent edits or corrections produce new traces and relations, preserving the original evidence, repository state, and entity bindings. These records make the conditions under which evidence was produced available for later validation. Persistence alone, however, does not establish current validity: a recorded test pass may no longer apply after the tested code changes. MemTrace therefore retains historical traces for provenance but reuses their evidence only when it remains applicable. Section 3.3 describes how ρ_τ and $\mathcal{E}(\tau)$ are checked against the current repository state R_t and task state m_t before restoration.

Offload bodies while retaining access. When working memory C_t approaches the context budget B, *Memory Offloading* removes selected completed trace bodies from the active context while preserving their full records in $\mathcal{H}_t$. Compact *Trace Synopses* summarize the offloaded content, and Memory Anchors provide stable references to it. Both count toward B, so only a budgeted subset remains in C_t. Anchors omitted from the active context remain stored with their traces and are recoverable through the persistent index. This separates persistent retention from in-context use: evidence remains accessible, while its restoration depends on the validation procedure in Section 3.3.

3.2 MTG–RSG Projection

MemTrace links execution history to the evolving codebase through two complementary graphs: the *Memory Trace Graph* (MTG) records task progress, and the *Repository State Graph* (RSG) represents current repository structure. Their nodes and edges remain separate; the projection Π_t connects traces to repository entities.

Execution and repository graphs. The MTG G_t^{M} contains one node per trace in $\mathcal{H}_t$, with relations recording execution order, dependencies, verification, correction, and supersession. For example, a repair trace may depend on a failure diagnosis and supersede an earlier edit, while a passing-test trace verifies the repair. These relations identify evidence that supports or revises the frontier m_t.

The RSG G_t^{R} connects task-relevant files, symbols, and tests through *defines*, *calls*, *imports*, and *covers* relations. Here, *covers* denotes a test-to-code association for localization, not measured coverage or a passing result. MemTrace incrementally updates the graph around the execution frontier as R_t changes. Its structural relations guide retrieval without establishing the validity of historical evidence.

Trace–repository projection. Each entity reference identifies a file, symbol, or test within its repository scope; symbols include their file path and qualified name. Sealing a trace records its bindings in

$$\Pi_t = \big\{(\tau, e) \mid \tau \in \mathcal{H}_t,\ e \in \mathcal{E}(\tau)\big\}. \tag{1}$$

An anchor resolves to one trace, whereas an entity lookup may return several. In Figure 3, a cache-invalidation query starts from T_4, which references `test_cache`. RSG traversal follows *covers* to `CacheManager.invalidate` and *calls* to `clear_cache`; lookups through Π_t then retrieve the associated traces T_5 and T_6.

Candidate retrieval. Retrieval starts from anchored traces, traces supporting m_t, or repository references. RSG expansion follows a bounded, request-specific sequence of relations; the figure's two-relation path is illustrative, not a global limit. Candidates may then expand by at most two MTG hops along dependency, verification, correction, and supersession relations. Traversal inspects incoming and outgoing edges without changing their meaning, allowing a failure trace to locate a later repair. Direct candidates precede expanded neighbors, and duplicates are removed before validation.

3.3 Validated Restoration

MemTrace restores evidence only when it is relevant to the next action, applicable to the current execution state, and within the context budget.

Applicability check. A trace is applicable when

$$\mathrm{applies}(\tau; R_t, m_t) \iff \mathrm{aligned}(\tau, R_t) \ \wedge\ \neg\mathrm{superseded}(\tau, m_t). \tag{2}$$

The predicate aligned checks the evidence's recorded source conditions against R_t, using ρ_τ, $\mathcal{E}(\tau)$, and provenance in x_τ. It does not require the entire repository to remain unchanged. For a test result, however, both the test and its recorded dependencies must be checked; an unchanged target function alone is insufficient. Changed or unverifiable conditions prevent automatic reuse. The predicate superseded requires an explicit MTG relation indicating replacement for the current frontier; temporal order alone is insufficient.

Selective restoration. MemTrace inspects retrieved candidates and, if needed, expands them as described in Section 3.2. Relevant traces satisfying Eq. (2) are restored with their source labels and anchors until the request is satisfied. Blocks that exceed the remaining budget are deferred rather than truncated. Explicit historical queries may retrieve older or superseded traces labeled as historical evidence. If applicable evidence is unavailable, the agent reads the current file or symbol, or reruns the required test. Budget-deferred evidence is tracked separately from unavailable or invalid evidence.

Recovery after a context refresh. Before refreshing the context, MemTrace offloads completed bodies and checkpoints m_t and the anchors for active work; the Trace Store, both graphs, and Π_t persist. Recovery then proceeds in three steps: (1) reconstruct task progress from MTG traces supporting or revising the frontier; (2) use Π_t and the RSG to locate repository entities relevant to continuation; and (3) restore relevant candidates that pass validation and fit within B. The admitted evidence, original task, recovered frontier, and current repository scope form the refreshed working memory. Algorithm 1 summarizes this procedure.

Algorithm 1 MTG–RSG context recovery

Require: State S_t, recovery request q, context budget B
Ensure: Reconstructed memory C_t^+
1: Checkpoint m_t and $\mathcal{A}_t$; offload completed bodies
2: $Q \leftarrow$ anchored or frontier-related traces
3: Expand Q by up to two MTG hops
4: Add q-relevant traces via the RSG and Π_t
5: Initialize C_t^+ with the task, checkpoint, and repository scope
6: **for** each q-relevant trace $\tau \in Q$ **do**
7: **if** $\text{applies}(\tau; R_t, m_t)$ and its body and anchor fit the remaining budget **then**
8: Restore x_τ and α_τ into C_t^+
9: **return** C_t^+

4 EXPERIMENTS

We evaluate MEMTRACE on three long-horizon coding benchmarks and two agent harnesses. We compare task performance (Section 4.2) and analyze component contributions and memory behavior (Section 4.3). Wall-clock results are in Appendix A.2.

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on three long-horizon software engineering benchmarks covering complementary task settings. DeepSWE (Huang et al., 2026) contains 113 tasks across 91 repositories and five programming languages. SWE-EVO (Thai et al., 2025) comprises 48 software-evolution tasks constructed from release histories of seven Python projects. Two instances already pass all FAIL_TO_PASS tests without code changes, so we evaluate the remaining 46 tasks. SWE-Milestone (Deng et al., 2026) contains 98 human-verified milestones across seven continuous software-evolution itineraries spanning five languages.

Backbone, harnesses, and baselines. We use `deepseek-v4-flash-0731-high` for all experiments, with a 200K-token context window and no explicit step limit. Experiments are conducted under Codex CLI v0.144.4 (Bolin, 2026) and mini-swe-agent v2.4.6 (SWE-agent, 2026). We compare against three baselines. Native uses the default context management of each harness. Observation Masking (Lindenbauer et al., 2025) masks older tool observations outside a rolling window while retaining the reasoning and action history. RepoGraph (Ouyang et al., 2025) augments the agent with repository-level structural retrieval and is evaluated only on Python tasks, following its implementation scope.

Metrics. We report benchmark-specific effectiveness metrics together with wall-clock time. For DeepSWE, we report pass@1 and the average wall-clock time per task, where pass@1 measures the fraction of tasks successfully resolved. For SWE-EVO, we report Resolved Rate, Fix Rate, and Apply, measuring complete resolution, partial functional progress, and successful patch application, respectively (Thai et al., 2025); we additionally report the average wall-clock time per task. For SWE-Milestone, we report Score, Precision, Recall, and Resolved Rate. Score is the harmonic mean of Precision and Recall, which measure functional correctness with respect to regressions and the fraction of required functionality completed, respectively (Deng et al., 2026).

4.2 MAIN RESULTS

Table 2 compares MEMTRACE with the baselines across three benchmarks and two agent harnesses. Beyond the overall gains, the results reveal three notable patterns: strong harness sensitivity among the baselines, persistent gains in Python-only settings, and a clear distinction between partial progress and complete task resolution.

(i) Baseline effectiveness depends strongly on the harness, whereas MEMTRACE remains effective across both. MEMTRACE achieves the best primary metric in every benchmark–harness setting. In contrast, the relative performance of the baselines changes substantially across harnesses.

Table 2: Main results (%) on DeepSWE, SWE-EVO, and SWE-Milestone. Full-benchmark results are reported with Python subsets in parentheses; SWE-EVO is Python-only. **Bold** denotes MEMTRACE, and underlining denotes the best baseline. Δ_{best} reports the percentage-point difference from the best baseline, computed directly from the reported values.

	DeepSWE	SWE-EVO			SWE-Milestone			
Method	pass@1	Resolved	Fix	Apply	Score	Prec.	Recall	Resolved
Harness: Codex CLI								
Native	35.4 (32.4)	47.8	53.4	97.8	28.6 (30.7)	26.9 (26.4)	41.9 (84.5)	7.1 (8.3)
Obs. Masking	34.5 (38.2)	65.2	72.1	97.8	24.4 (30.0)	24.2 (26.0)	37.1 (84.9)	9.2 (8.3)
RepoGraph	– (41.2)	58.7	68.8	97.8	– (45.2)	– (36.7)	– (91.5)	– (8.3)
MEMTRACE	**56.6 (67.6)**	**69.6**	**70.6**	**100.0**	**46.4 (46.1)**	**43.6 (37.8)**	**60.9 (91.7)**	**16.3 (16.7)**
Δ_{best}	**+21.2 (+26.4)**	**+4.4**	**-1.5**	**+2.2**	**+17.8 (+0.9)**	**+16.7 (+1.1)**	**+19.0 (+0.2)**	**+7.1 (+8.4)**
Harness: mini-swe-agent								
Native	48.7 (47.1)	52.2	54.1	95.7	34.5 (17.6)	34.7 (19.6)	45.6 (54.2)	11.2 (0.0)
Obs. Masking	26.6 (20.6)	23.9	33.6	97.8	24.9 (21.9)	25.1 (21.4)	29.9 (22.5)	8.2 (8.3)
RepoGraph	– (32.4)	30.4	37.9	97.8	– (30.8)	– (25.5)	– (69.1)	– (0.0)
MEMTRACE	**66.4 (64.7)**	**63.0**	**68.7**	**93.5**	**48.3 (37.3)**	**46.9 (32.6)**	**71.2 (82.2)**	**19.4 (16.7)**
Δ_{best}	**+17.7 (+17.6)**	**+10.8**	**+14.6**	**-4.3**	**+13.8 (+6.5)**	**+12.2 (+7.1)**	**+25.6 (+13.1)**	**+8.2 (+8.4)**

On SWE-EVO, for example, Observation Masking improves over Native by 17.4 points under Codex CLI, but falls 28.3 points below Native under mini-swe-agent; RepoGraph shows a similar reversal. This suggests that context reduction and structural retrieval are sensitive to the surrounding agent execution policy, while the benefit of MEMTRACE is more stable across harnesses.

(ii) The gain persists on Python, where RepoGraph is directly applicable. On the DeepSWE Python subset, MEMTRACE exceeds RepoGraph by more than 26 points under Codex CLI and 32 points under mini-swe-agent. A different pattern appears on SWE-Milestone: under Codex CLI, MEMTRACE and RepoGraph are nearly tied on Python Score, Precision, and Recall, yet the strict Resolved Rate increases from 8.3% to 16.7%. Thus, the advantage is not explained by language coverage alone; on sequential evolution tasks, it is particularly visible in converting partial progress into complete solutions.

(iii) Higher partial-progress metrics do not necessarily translate into higher task resolution. On SWE-EVO under Codex CLI, Observation Masking attains a slightly higher Fix Rate than MEMTRACE (72.1% vs. 70.6%), but a lower Resolved Rate (65.2% vs. 69.6%). Under mini-swe-agent, MEMTRACE has a lower Apply rate than the best baseline (93.5% vs. 97.8%) while achieving a substantially higher Resolved Rate (63.0% vs. 52.2%). The improvement therefore appears primarily in end-to-end task completion, rather than in patch applicability or partial repair alone.

4.3 ANALYSIS

Component ablation. Adding the MTG to the Trace Store improves DeepSWE from 32.1% to 44.2%, showing that execution relations help connect earlier edits, tests, and corrections across a long trajectory. Adding the RSG further raises performance to 56.6%, indicating the value of tracking relations among files, symbols, and tests when task progress spans a broad repository context. This effect is much smaller on SWE-EVO, where removing the RSG leaves performance unchanged at 69.6%. One likely reason is the difference in task scope: DeepSWE covers repository-level tasks across many repositories and languages, whereas SWE-EVO follows release-level evolution within a smaller set of projects. As a result, explicit repository grounding appears more useful when recovery must reconnect evidence across a broader and more heterogeneous codebase.

State–memory alignment. We examine how later repository changes affect stored execution evidence. Among 9,226 eligible file-bound Memory Traces, 57.0% encounter a later change to at least one bound file. Exposure is more frequent in tasks with more repository revisions (Spearman $\rho = 0.49$; Figure 4(a)). The estimated probability that bound files remain unchanged falls to about

Table 3: Component ablation under Codex CLI. Trace Store Only replaces MTG and RSG routing with flat retrieval over Trace Synopses and bodies; w/o MTG removes execution-order and dependency edges; w/o RSG removes repository-structure candidate expansion. Cells report Resolved Rate for SWE-EVO and pass@1 for DeepSWE.

Variant	Component			SWE-EVO Resolved Rate (%)	DeepSWE pass@1 (%)
	Trace Store	MTG	RSG		
Native	–	–	–	47.8	35.4
Trace Store Only	✓	–	–	65.2	32.1
MEMTRACE w/o MTG	✓	–	✓	65.2	34.5
MEMTRACE w/o RSG	✓	✓	–	**69.6**	44.2
MEMTRACE	✓	✓	✓	**69.6**	**56.6**

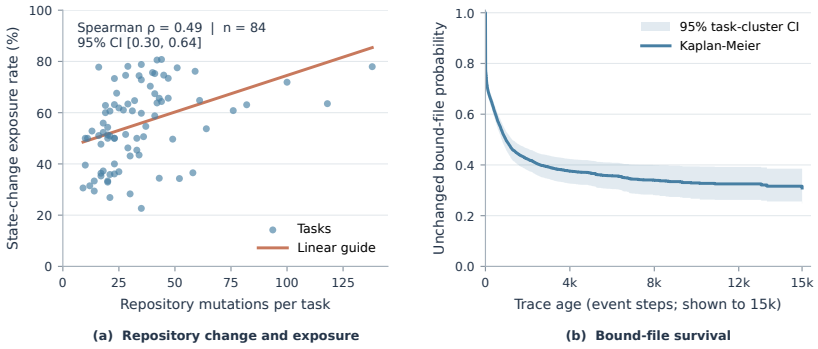


Figure 4: State–memory alignment under repository evolution. (a) Each point is one task, showing the relation between repository revisions and the fraction of Memory Traces whose bound files are later modified. (b) The probability that a trace's bound files remain unchanged as the trace becomes older. The shaded region shows the 95% task-cluster bootstrap confidence interval. Details are in Appendix A.3.

0.50 after 1,000 logged event steps (Figure 4(b)). These observations motivate checking current applicability before reusing historical evidence. Definitions and statistics are in Appendix A.3.

Long-horizon behavior. We group DeepSWE tasks into short, medium, and long execution horizons using response counts from an independent Native run (Appendix A.4). Across the three groups, median stored Memory Traces increase as $79 \rightarrow 102.5 \rightarrow 129$, repository revisions as $21 \rightarrow 30 \rightarrow 41$, and semantic evidence invalidations as $122 \rightarrow 142 \rightarrow 223$ (Figure 5(a–c)). Longer execution therefore corresponds to more accumulated history and repository evolution, increasing state-aware memory maintenance. The exposure rate, however, shows no clear monotonic increase with execution length (Figure 5(d)), indicating that the added maintenance load is driven mainly by more stored history and repository changes rather than a higher per-trace exposure rate.

5 Conclusion and Future Work

We introduced MEMTRACE, a state-consistent memory system for long-horizon coding agents that preserves execution evidence with repository provenance and validates it before reuse. Across long-horizon benchmarks, MemTrace consistently improves the primary task-level metric over fully evaluated baselines across two agent harnesses, while our analyses show the complementary value of structured execution memory and repository-state grounding. These results suggest that long-horizon memory should maintain not only historical information, but also its applicability to the current execution state. Future work will evaluate MemTrace with additional backbone models and context-window sizes, and extend it to persistent development settings spanning multiple tasks and sessions within the same repository.

AI USE STATEMENT

In this work, we have not used generative AI tools for any of the tasks requiring disclosure: generative AI was not used to develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to design research methodology or experiments, to implement the proposed methods, or to interpret results; the remaining required-disclosure tasks (e.g., generating synthetic datasets, formulating or proving mathematical claims, and dataset cleaning) are not applicable to this work. Additionally, we used generative AI tools to draft and debug auxiliary code (e.g., experiment scripts and tooling) and to edit the paper to improve readability. We have reviewed all AI-assisted work: all AI-generated code was verified and tested for correctness by the authors, and all AI-edited text was checked by the authors to ensure it preserves the intended meaning. The core concepts and innovations of this work were developed entirely by the authors without AI involvement. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. No human-subject or animal experimentation was involved. All benchmarks used in this study—DeepSWE, SWE-EVO, and SWE-Milestone—are publicly released and derived from open-source software repositories. We use these benchmarks solely for evaluating repository-level coding agents and do not use or analyze personally identifiable information as part of our evaluation. The experiments were conducted in controlled execution environments and did not involve deployment against real-world production systems or private repositories. Because the agents studied in this work can modify repository contents and execute commands, practical deployments should retain appropriate sandboxing, access controls, and human review of generated changes.

REPRODUCIBILITY STATEMENT

We have made every effort to ensure that the results presented in this paper are reproducible. Section 4.1 specifies the benchmarks, backbone model and configuration, agent harnesses, baselines, evaluation metrics, and experimental settings; no model training is involved. Section 3 defines the MemTrace memory representation, while Section 3.3, Eq. (2), and Algorithm 1 specify the applicability check and bounded recovery procedure. Appendix A.2 reports the resolved-task counts underlying the main results and the corresponding wall-clock statistics. Appendices A.3 and A.4 define the trajectory-level analyses, their evaluation populations and statistics, and the bootstrap procedure with fixed seeds. All three benchmarks are publicly available (Huang et al., 2026; Thai et al., 2025; Deng et al., 2026). We will release the anonymized implementation, run configurations, and evaluation logs as supplementary material to facilitate independent replication and verification.

REFERENCES

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/25f7be9694d7b32d5cc670927b8091e1-Paper-Conference.pdf.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. volume 1, pp. 675–698, 2024. doi: 10.1145/3643757. URL https://doi.org/10.1145/3643757.

Michael Bolin. Unrolling the codex agent loop. OpenAI Engineering, 2026. URL https://openai.com/index/unrolling-the-codex-agent-loop/.

Howard Chen, Ramakanth Pasunuru, Jason Weston, and Asli Celikyilmaz. Walking down the memory maze: Beyond context limit through interactive reading. *arXiv preprint arXiv:2310.05029*, 2023. URL https://arxiv.org/abs/2310.05029.

Yaoqi Chen, Haibin Lai, Yuru Feng, Chuyu Han, Qianxi Zhang, Baotong Lu, Menghao Li, Xinjiang Wang, Zhirui Wang, Shusen Xu, Zengzhong Li, Zewen Jin, Hao Wu, Cheng Li, and Qi Chen. Beyond semantic organization: Memory as execution state management for long-horizon agents. *arXiv preprint arXiv:2606.06090*, 2026. URL https://arxiv.org/abs/2606.06090.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor K. Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 8697–8727. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.ACL-LONG.426. URL https://doi.org/10.18653/v1/2025.acl-long.426.

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

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. SWE-Milestone: Evaluating AI agents on continuous software evolution. *arXiv preprint arXiv:2603.13428*, 2026. URL https://arxiv.org/abs/2603.13428.

Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5): 323–333, 1968. URL https://denninginstitute.com/pjd/PUBS/WSModel_1968.pdf.

Darren Edge et al. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024. URL https://arxiv.org/abs/2404.16130.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.14831.

Xubin Hao, Hongjin Meng, Xin Yin, Jiawei Zhu, and Chenpeng Cao. Self-GC: Self-governing context for long-horizon LLM agents. *arXiv preprint arXiv:2607.00692*, 2026. URL https://arxiv.org/abs/2607.00692.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026. URL https://arxiv.org/abs/2607.07946.

Dongming Jiang, Yi Li, Guanpeng Li, and Bingzhe Li. MAGMA: A multi-graph based agentic memory architecture for AI agents. *arXiv preprint arXiv:2601.03236*, 2026. URL https://arxiv.org/abs/2601.03236.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of EMNLP*, 2023. URL https://aclanthology.org/2023.emnlp-main.825/.

Huiqiang Jiang et al. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of ACL*, 2024. URL https://arxiv.org/abs/2310.06839.

Carlos E. Jiménez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts. In *Proceedings of ICML*, 2024. URL https://arxiv.org/abs/2402.09727.

Yin Lin, Elaine Ang, Erkang Zhu, Bolin Ding, and Jingren Zhou. Context as an environment: Programmatic context management for long-horizon agents. *arXiv preprint arXiv:2608.21690*, 2026. URL https://arxiv.org/abs/2608.21690.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management. *CoRR*, abs/2508.21433, 2025. doi: 10.48550/ARXIV.2508.21433. URL https://doi.org/10.48550/arXiv.2508.21433.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon SWE-agents. *arXiv preprint arXiv:2512.22087*, 2025a. URL https://arxiv.org/abs/2512.22087.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code graph databases. pp. 142–160, 2025b. doi: 10.18653/V1/2025.NAACL-LONG.7. URL https://doi.org/10.18653/v1/2025.naacl-long.7.

Siru Ouyang et al. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4a4a3c197deac042461c677219efd36c-Paper-Conference.pdf.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park et al. Generative agents: Interactive simulacra of human behavior. In *Proceedings of UIST*, 2023. URL https://arxiv.org/abs/2304.03442.

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

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. RAPTOR: Recursive abstractive processing for tree-organized retrieval. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/8a2acd174940dbca361a6398a4f9df91-Paper-Conference.pdf.

Noah Shinn et al. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.11366.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*, 2023. URL https://arxiv.org/abs/2309.02427.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon LLM agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025. URL https://arxiv.org/abs/2510.11967.

SWE-agent. mini-swe-agent, version 2.4.6. Software release, 2026. URL https://github.com/SWE-agent/mini-swe-agent/releases/tag/v2.4.6.

Minh V. T. Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: benchmarking coding agents in long-horizon software evolution scenarios. *CoRR*, abs/2512.18470, 2025. doi: 10.48550/ARXIV.2512.18470. URL https://doi.org/10.48550/arXiv.2512.18470.

Guanzhi Wang et al. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Yuhang Wang et al. SWE-Pruner: Self-adaptive context pruning for coding agents. *arXiv preprint arXiv:2601.16746*, 2026a. URL https://arxiv.org/abs/2601.16746.

Zehao Wang, Yisen Xu, Chenglin Li, Chao Peng, Bram Adams, Ahmed E. Hassan, and Tse-Hsun Chen. Turning interaction history into execution state: A runtime layer for long-horizon coding agents. *arXiv preprint arXiv:2608.00808*, 2026b. URL https://arxiv.org/abs/2608.00808.

Zora Zhiruo Wang et al. Agent workflow memory. In *Proceedings of ICML*, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/d813d324dbf0598bbdc9c8e79740ed01-Paper-Conference.pdf.

Chunqiu Steven Xia et al. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. *arXiv preprint arXiv:2502.12110*, 2025. URL https://arxiv.org/abs/2502.12110.

John Yang et al. SWE-agent: Agent–computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. G-Memory: Tracing hierarchical memory for multi-agent systems. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2506.07398.

Yuntong Zhang et al. AutoCodeRover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024. URL https://arxiv.org/abs/2404.05427.

A Appendix

A.1 Benchmark Composition

Table 4 summarizes the official benchmark composition. DeepSWE contains 113 tasks from 91 repositories (Huang et al., 2026); SWE-EVO contains 48 tasks from seven Python projects (Thai et al., 2025). SWE-Milestone comprises seven repository-level itineraries and 98 graded milestones (Deng et al., 2026). Milestones are graded stages within an itinerary.

Table 4: Official task distribution by repository language (Huang et al., 2026; Thai et al., 2025; Deng et al., 2026). SWE-Milestone reports both evolution itineraries and graded milestones.

	DeepSWE	**SWE-EVO**	**SWE-Milestone**	
Language	Tasks	Tasks	Itineraries	Graded milestones
Python	34	48	1	12
TypeScript	35	0	1	18
Go	34	0	2	32
JavaScript	5	0	0	0
Rust	5	0	2	24
Java	0	0	1	12
Total	**113**	**48**	**7**	**98**

Counting convention. Graded milestone counts are go-zero (23), Navidrome (9), element-web (18), Nushell (13), ripgrep (11), Dubbo (12), and scikit-learn (12). Language groups follow the official catalogue, which classifies Navidrome under Go (Deng et al., 2026). Resolved-rate tables below use these official sets, with two stated exclusions. SWE-EVO rates use 46 tasks: `pydantic` v2.7.1 to v2.7.2 and `scikit-learn` 0.21.1 to 0.21.2 fail benchmark validation and are removed before scoring. They are not unresolved agent attempts. Every DeepSWE rate in Table 5 uses all 113 tasks.

A.2 Additional Results and Wall-Clock Time

Table 5 gives the resolved-task counts underlying the main results. RepoGraph is reported for Python tasks only. SWE-Milestone uses its official itinerary-level aggregation. The DeepSWE Trace Store Only ablation was scheduled on the same 113 tasks. `httpx-streaming-json-iteration` has no official score (`MISSING_RESULT`; the attempt exited in about 33 seconds) and is excluded from both the numerator and the denominator. The scored result is 36/112 (32.1%). That task is not counted as a resolved success or as a model failure.

Implementation scope. Configurations share the backbone within each harness setup. Component ablations use Codex CLI. The audited RSG provides symbol-level relations for Python and file-level relations for the other languages.

Table 5: Resolved-task counts for Table 2.

		DeepSWE			**SWE-EVO**
Harness	**Method**	All (113)	Python (34)	Non-Python (79)	All (46)
Codex CLI	Native	40	11	29	22
	Obs. Masking	39	13	26	30
	RepoGraph	–	14	–	27
	MemTrace	64	23	41	32
mini-swe-agent	Native	55	16	39	24
	Obs. Masking	30	7	23	11
	RepoGraph	–	11	–	14
	MemTrace	75	22	53	29

Wall-clock time. Table 6 reports mean elapsed time per task in minutes, rounded to two decimal places. Times are compared within each harness and benchmark.

Table 6: Mean wall-clock time in minutes per task.

	DeepSWE		SWE-EVO	
Method	Codex CLI	mini-swe-agent	Codex CLI	mini-swe-agent
Native	42.02	103.77	30.00	170.97
Obs. Masking	44.32	120.98	52.72	170.02
RepoGraph	34.67†	70.43†	34.22	203.28
MEMTRACE	54.17	75.30	82.05	114.03

Note. † DeepSWE RepoGraph means use the 34 Python tasks. Other DeepSWE means use 113 tasks.

A.3 STATE–MEMORY ALIGNMENT

Analysis population. These diagnostics use archived DeepSWE mini-swe-agent trajectories, separately from the performance runs in Table 2. Exposure analysis includes 9,226 Memory Traces from 84 trajectories, each with a recorded sealing event and nonempty file bindings. Revision, invalidation, and Context Refresh counts use 85 instrumented trajectories (Table 7). Memory Trace identifiers and semantic evidence records are counted separately.

Table 7: Analysis population and event counts for the state–memory diagnostics.

Quantity	**Count**
Event-instrumented trajectories	85
Trajectories with eligible file-bound Memory Traces	84
Eligible time-aligned, file-bound Memory Traces	9,226
Memory Traces with a later bound-file change	5,261
Repository revision events	2,897
Semantic evidence invalidations	16,197
Explicit context-refresh events	258

Event clock and repository revisions. For task q, let u_q be its last recorded event step and $\mathcal{V}_q$ its revision events. Each revision r has a step s_r and modified-file set ΔF_r. The sealing step c_τ is the recorded finalization of Memory Trace τ, matched by its identifier. Repository revisions count `WORKSPACE_REVISION_ADVANCED` events:

$$M_q = |\mathcal{V}_q|. \tag{3}$$

The figure label "repository mutations" refers to the same count. Event steps are log positions, distinct from model responses and elapsed time.

Repository-state exposure. Let F_τ be the files bound to Memory Trace τ at sealing, and $\mathcal{T}_q^*$ the traces with a recorded sealing step and nonempty F_τ. Exposure records a later change to at least one bound file:

$$x_\tau = \mathbf{1}[\exists r \in \mathcal{V}_q : c_\tau < s_r \le u_q \ \wedge \ F_\tau \cap \Delta F_r \neq \varnothing]\,. \tag{4}$$

The task-level and pooled rates are

$$E_q = \frac{\sum_{\tau \in \mathcal{T}_q^*} x_\tau}{|\mathcal{T}_q^*|}, \qquad E = \frac{\sum_q \sum_{\tau \in \mathcal{T}_q^*} x_\tau}{\sum_q |\mathcal{T}_q^*|}. \tag{5}$$

The pooled rate weights traces equally; Figure 4(a) uses one (M_q, E_q) pair per eligible task. The pooled rate is $5{,}261/9{,}226 = 57.0\%$ (95% CI: 52.8–61.3%). Exposure measures file changes; Trace Validation assesses whether historical evidence remains applicable.

Semantic evidence invalidations. `SemanticStore.advance_workspace_revision` in `semantic_memory/store.py` counts evidence identifiers selected for closure at each revision. Let $\mathcal{C}_{r^-}$ contain records marked `CURRENT` with an open validity interval (`valid_to_cursor IS NULL`) before revision r. Write $\nu(e)$ for a record's revision, ν_r for the new revision, and $A_r(e)$ for the affected-entity predicate. For global fact types $\mathcal{G}$ and entity-scoped types $\mathcal{D}$, the selected identifiers are

$$\mathcal{I}_r = \{\mathrm{id}(e) : e \in \mathcal{C}_{r^-},\ \nu(e) \neq \nu_r, \\ \mathrm{type}(e) \in \mathcal{G}\ \vee\ [\mathrm{type}(e) \in \mathcal{D} \wedge A_r(e)]\}. \tag{6}$$

Per-revision and cumulative per-task counts are

$$I_r = |\mathcal{I}_r|, \qquad I_q = \sum_{r \in \mathcal{V}_q} I_r. \tag{7}$$

Global types are `TOOL_RESULT`, `TEST_RESULT`, `TEST_FAILURE`, `VERIFIER_RESULT`, and `REQUIREMENT_REVIEW`; they close on a revision regardless of modified-file overlap. Entity-scoped types, `CODE_CHANGE` and `CODE_OBSERVATION`, also require `_entity_affected` to match a file, symbol, or test binding. Closure records the validity endpoint and an `INVALIDATED_BY` relation while retaining the Memory Trace. The 2,897 revisions yield 16,197 recorded invalidations. Thus, I_q measures revision-triggered maintenance under this policy, not independently verified semantic errors.

Statistical procedure. We report task-level Spearman correlations and 95% percentile bootstrap intervals from 5,000 resamples. Trace-level rates and survival resample tasks with all their eligible traces; correlations resample task pairs; Long–Short median differences resample within each group. The scripts use base seed 20260926 with fixed per-analysis offsets. Table 8 reports the task-level associations.

Table 8: Task-level associations with 95% task-bootstrap confidence intervals. Pearson correlation is a secondary descriptive statistic.

Variables	N	**Spearman ρ [95% CI]**	**Pearson r**
Revisions vs. exposure	84	$0.487\ [0.305, 0.642]$	0.394
Revisions vs. invalidations	85	$0.837\ [0.741, 0.899]$	0.881

Time to the first bound-file change. For an eligible Memory Trace, define

$$T_\tau = \min\{s_r - c_\tau : r \in \mathcal{V}_q,\ s_r > c_\tau,\ F_\tau \cap \Delta F_r \neq \varnothing\}, \tag{8}$$

with $T_\tau = \infty$ when no change is observed. Follow-up ends at $C_\tau = u_q - c_\tau$. We observe $Y_\tau = \min(T_\tau, C_\tau)$ and $\delta_\tau = \mathbf{1}[T_\tau \leq C_\tau]$; traces without a recorded change are right-censored at the run endpoint. At event age a_j, let d_j count first changes and n_j count traces still observed and unchanged immediately beforehand. The Kaplan–Meier estimate is

$$\widehat{S}(a) = \prod_{a_j \leq a} \left(1 - \frac{d_j}{n_j}\right). \tag{9}$$

Figure 4(b) estimates the probability that no bound file has yet changed, using the internal event clock and pointwise task-cluster bootstrap intervals. Table 9 gives selected values.

Table 9: Bound-file survival at selected event-step ages. Intervals are 95% task-cluster bootstrap intervals; at-risk counts are measured immediately before each age.

Event-step age	$\widehat{S}(a)$ **[95% CI]**	**Memory Traces at risk**
97	$0.703\ [0.682, 0.723]$	6,380
1,001	$0.495\ [0.456, 0.530]$	3,654
2,012	$0.422\ [0.379, 0.464]$	2,404

Grouping traces by total observed age C_τ gives exposure rates of 27.4%, 60.8%, 69.9%, and 70.1% across quartiles. These are descriptive summaries with different follow-up lengths; the time-to-change estimate accounts for right censoring.

A.4 LONG-HORIZON BEHAVIOR

Execution-length groups. For task q, execution length is the number of distinct model responses in an independent Native mini-swe-agent trajectory:

$$H_q = \left|\{\mathrm{id}(z) : z \text{ is a model response in the Native trajectory for } q\}\right|. \tag{10}$$

The extractor counts unique identifiers with `object == "response"`. H_q measures observed execution length, not intrinsic task difficulty. Native outcome labels are not used for grouping or performance comparison. We sort by $(H_q, \mathrm{task_id})$ and form three near-equal groups, keeping each task's group fixed across metrics. Table 10 gives the ranges and group sizes.

Table 10: Execution-length groups defined from independent Native response counts.

Group	Native responses	Median	Tasks
Short	10–197	145	37
Medium	198–290	248	38
Long	291–526	343	38

Measures and distributions. Stored Memory Traces are counted from their manifests. Revisions M_q, invalidations I_q, and exposure E_q follow Appendix A.3. Context Refreshes count `NEW_EPOCH_MODEL_OBSERVED_AND_ACTIVATED` events. The 258 events across 85 trajectories match the recorded epoch counts. Figure 5 shows task-level distributions; Table 11 gives medians and interquartile ranges.

Table 11: Memory behavior by execution-length group: median [25th, 75th percentile]. Group-specific sample sizes are given in Figure 5; Context Refresh counts use the same tasks as revision counts.

Metric	Short	Medium	Long
Stored Memory Traces	79 [61,112]	102.5 [74.25,160.25]	129 [92.5,218.5]
Repository revisions	21 [17,27]	30 [20,37]	41 [26,49]
Evidence invalidations	122 [66,177]	142 [79,231]	223 [142.5,350.5]
Context Refreshes	2 [1,2]	2 [2,4]	3 [2,4.5]
Exposure (%)	54.4 [50.0,64.4]	49.7 [36.5,60.7]	64.4 [50.0,75.5]

Association with execution length. From Short to Long, median Memory Trace counts rise from 79 to 129, revisions from 21 to 41, and evidence invalidations from 122 to 223. Their correlations with Native response count are $\rho = 0.385$, 0.466, and 0.403, respectively (Table 12). Context Refresh count is also positively associated with response count. Exposure has no clear monotonic association ($\rho = 0.137$; 95% CI: $[-0.098, 0.361]$). Longer measured horizons are therefore associated with more stored history and revision-triggered maintenance, rather than a consistently higher exposure fraction.

Table 12: Association with Native response count and the Long–Short median difference. Intervals are 95% task-bootstrap intervals. Exposure differences are percentage points; all other differences are counts.

Metric	N	Spearman ρ [95% CI]	Long–Short [95% CI]
Stored Memory Traces	113	$0.385\ [0.206, 0.539]$	$50\ [27, 78]$
Repository revisions	85	$0.466\ [0.285, 0.614]$	$20\ [9, 25]$
Evidence invalidations	85	$0.403\ [0.205, 0.566]$	$101\ [33, 206]$
Context Refreshes	85	$0.382\ [0.171, 0.560]$	$1\ [0, 2]$
Exposure (pp)	84	$0.137\ [-0.098, 0.361]$	$10.0\ [-1.5, 19.2]$



Figure 5: Memory behavior across short-, medium-, and long-horizon tasks. (a) Stored Memory Traces. (b) Repository revisions. (c) Stored observations or facts closed by MemTrace's revision policy (semantic evidence invalidations; Appendix A.3). (d) Fraction of file-bound Memory Traces whose bound files are later modified. Points are tasks; boxes show medians and interquartile ranges. Whiskers extend to observations within 1.5 interquartile ranges. Group sizes are 37/38/38 in (a), 25/29/31 in (b–c), and 24/29/31 in (d). Groups are defined by response counts from an independent Native run.

Alternative execution-length measures. Native response count correlates with tool calls ($\rho = 0.980$), wall-clock time ($\rho = 0.677$), and output tokens ($\rho = 0.508$). Using tool calls or output tokens for grouping preserves 97/113 (85.8%) or 52/113 (46.0%) of the response-based assignments, respectively. Memory Trace counts and invalidations are positively associated with all three length measures (Table 13). Revisions and Context Refreshes show stronger associations with responses and tool calls than with output tokens.

Table 13: Spearman correlations under alternative Native execution-length measures, with 95% task-bootstrap intervals. Sample sizes follow Table 12.

Metric	Responses	Tool calls	Output tokens
Stored Memory Traces	0.385 [0.206, 0.539]	0.384 [0.209, 0.538]	0.200 [0.014, 0.367]
Repository revisions	0.466 [0.285, 0.614]	0.463 [0.278, 0.616]	0.199 [−0.002, 0.380]
Evidence invalidations	0.403 [0.205, 0.566]	0.401 [0.201, 0.570]	0.256 [0.052, 0.440]
Context Refreshes	0.382 [0.171, 0.560]	0.385 [0.175, 0.564]	0.187 [−0.039, 0.392]
Exposure	0.137 [−0.098, 0.361]	0.126 [−0.106, 0.348]	0.214 [−0.011, 0.432]

Within-run accumulation. We split each instrumented trajectory into four equal event-step intervals and count traces and events accumulated by each interval endpoint. Median cumulative Memory Trace counts are 23, 52, 75, and 102; median cumulative invalidations are 33, 80, 136, and 158. Figure 6 separates Context Refreshes from the larger trace and invalidation counts.

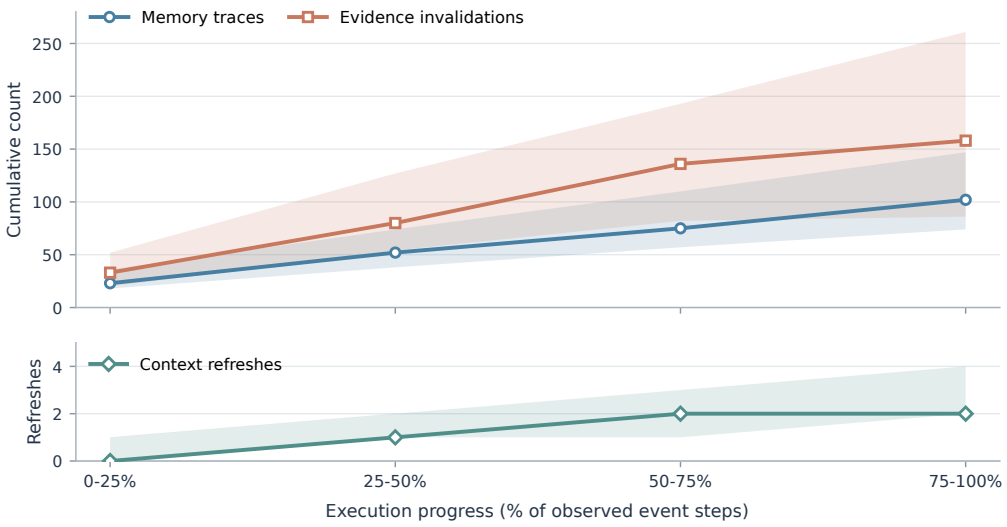


Figure 6: Cumulative counts across execution progress in 85 instrumented trajectories. Lines show task medians and shaded regions show interquartile ranges. Progress uses internal event steps. The upper panel shows Memory Traces and evidence invalidations; the lower panel shows Context Refreshes on a separate scale.

A.5 Case Study: Repository Evolution Across a Context Refresh

Task and selection. We examine the successful archived mini-swe-agent trajectory for `scriggo-method-declarations`. The task adds method declarations to Scriggo, including value and pointer receivers, method expressions, and interface dispatch (Huang et al., 2026). Its records link a sealed Memory Trace to a later bound-file revision, a Context Refresh, and subsequent verification.

Trace and repository evolution. At event step 1,061, Memory Trace τ_A is sealed with a binding to `internal/compiler/parser_test.go`. At step 17,152, that file is modified in a revision recording 15 semantic evidence invalidations. The shared file path and event order establish exposure for τ_A under Eq. (4). The invalidation count applies to the revision, not specifically to this trace.

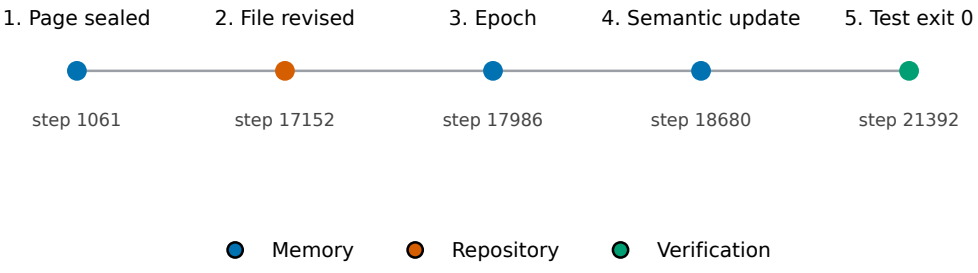


Figure 7: Selected events from the Scriggo trajectory. Events show Memory Trace sealing, a bound-file revision, Context Refresh, a semantic update, and successful in-run verification. Positions show event order, not elapsed-time spacing.

Continuation and verification. A new context epoch is activated at step 17,986. At step 18,680, two semantic updates are accepted for Execution Milestone `M003`. At step 21,392, `go test`

`./compare/ -count=1` returns exit code 0 for criterion `M003.C001`. The separate archived score summary records Harbor reward 1. Table 14 summarizes the logged sequence.

Table 14: Recorded events in the Scriggo case. File bindings come from the Memory Trace manifest; step numbers use the internal event clock.

Step	Record	Observed information
1,061	Memory Trace sealing	τ_{A} is sealed; its manifest includes the compiler parser-test file.
17,152	Repository revision	The bound file is modified; the revision records 15 evidence invalidations.
17,986	Context Refresh	A new context epoch is activated after model observation.
18,680	Semantic update	Two semantic updates are accepted for `M003`.
21,392	Runtime verification	The comparison test command exits with code 0; criterion `M003.C001` is marked successful.

Interpretation. The case links historical storage, a later change to a bound file, and successful continuation across a Context Refresh. The file binding identifies where current applicability must be checked; the later test supplies evidence about the updated execution state. This illustrates the distinction between preserving history and establishing its current applicability.