CMB: A CODING MEMORY BENCHMARK FOR LONG-HORIZON AND INTERLEAVED CODING AGENTS

Jiazheng Kang* Tencent Hunyuan

Tianxiang Fei* Tencent Hunyuan

Mao Zheng Tencent Hunyuan

ABSTRACT

Coding agents increasingly operate in long sessions where tasks are interrupted, requirements evolve, branches diverge, and unfinished work must later be resumed. Memory must therefore support subsequent repository actions, not merely recall prior text. We introduce CodeMemoryBench (CMB), an execution-grounded benchmark for memory in long-session coding agents. CMB constructs controlled, interleaved task histories in real repositories, freezes interaction streams and repository checkpoints, and replays identical histories to every memory mechanism. It comprises 216 code-editing tasks and 750 diagnostic questions over 5,595 turns from nine repositories in four languages. Experiments with five existing memory mechanisms reveal substantial room for improvement: the strongest mechanism resolves only 36.6% of CMB tasks with the stronger of two agent backbones. To address this limitation, we propose a three-layer memory that links task facts, reusable experience, and concrete code diffs through provenance and evolution relations. It reaches 43.1% and 35.6% on CMB across the two backbones, compared with 36.6% and 28.2% for the strongest baselines. On SWE-ContextBench, it achieves 47.5% and 28.3%, versus 45.5% and 25.3% for the strongest competing memories. CMB connects diagnostic recall with executable outcomes, measuring whether memory enables agents to resume and complete coding work.

1 INTRODUCTION

A substantial coding session spans more than one self-contained request. A developer may ask an agent to implement a configuration option, investigate an unrelated failure, change an earlier requirement, and then continue the original task. By that point, the conversation contains useful knowledge absent from the current request: constraints already agreed upon, unsuccessful approaches, the reason for a design choice, and work that remains unfinished. A capable coding agent must retain and use this context throughout the session.

Memory plugins address context limits by storing and retrieving information outside the immediate prompt (Chhikara et al., 2025; Li et al., 2025; Packer et al., 2023). Yet remembering long coding sessions requires more than retrieving text about a related topic. A recalled fact can lose its task context; a useful procedure can be applied outside the conditions that made it work; and a summary of a change can omit the code needed to continue it. Interruptions, branch switches, and repeated edits amplify these problems. Version confusion is one concrete symptom of a broader failure to connect what happened, what was learned, and what changed in the repository.

This motivates an evaluation of memory through both recalled content and the subsequent coding action. Conversational benchmarks study temporal reasoning, knowledge updates, and cross-session recall (Maharana et al., 2024; Wu et al., 2024). Repository issue benchmarks provide execution-based correctness (Jimenez et al., 2024). Our focus is the memory needed to sustain a long developer–agent coding session, where task-relevant information is distributed across earlier interactions and must be used when work resumes.

We introduce CodeMemoryBench for this setting. We construct long sessions in real repositories, interleave coding tasks with noise turns, record the resulting work, and freeze the histories. Every

*Equal contribution.

memory mechanism ingests the same recorded stream before the agent answers a history-dependent request or edits the repository. The benchmark contains 216 executable tasks and 750 diagnostic questions over nine repositories and four languages, with 302–928 turns per repository. Code evolution supplies a demanding task family: the agent must retain decisions and implementation evidence despite later edits. Existing memory mechanisms solve at most 36.6% of the code tasks under the stronger backbone, demonstrating substantial room for improvement.

We propose TriLink, a coding memory with three connected layers. Fact Memory records task progress and outcomes. Experience Memory records useful approaches and their applicability. Diff Memory preserves concrete implementation changes. The layers evolve with the same interaction stream: updated facts retain experience links, experiences point back to supporting facts, and retrieved facts anchor relevant code histories. The agent can therefore combine the task's state, reusable lessons, and implementation evidence when continuing its work.

Our contributions are CMB, an execution-graded benchmark of memory in long coding sessions; TriLink, a method linking facts, experience, and code evolution; and an evaluation on both CMB and the external SWE-ContextBench suite under two backbones. The latter tests the usefulness of the combined memory for experience reuse across related coding tasks.

2 Related Work

Memory for agents. Recent surveys organize agent memory by its theories, taxonomies, and evaluations (Bai et al.). Generative Agents retrieves observations and synthesizes reflections, while MemGPT manages memory tiers (Park et al., 2023; Packer et al., 2023). Mem0 consolidates conversational content; MemoryOS organizes short-, mid-, and long-term storage (Chhikara et al., 2025; Kang et al., 2025). MemOS manages memory lifecycles (Li et al., 2025), and OpenViking[1] organizes context hierarchically. DCPM goes beyond retrieval with a dual-process design that models belief evolution, latent intentions, and cross-domain abstractions (Fei et al., 2026b). A-Mem updates interconnected notes, while MAGE preserves execution dependencies in a state tree (Xu et al., 2026; Chen et al., 2026). Procedural memory emphasizes reuse: MemP distills and revises instructions from trajectories; Reflexion retains feedback-derived reflections for later trials (Fang et al., 2026; Shinn et al., 2023). These works motivate evolving memory; TriLink links task facts and reusable experience to concrete code changes.

Memory for software engineering. RepoCoder alternates retrieval and completion; GraphCoder uses control and data dependencies to retrieve repository context (Zhang et al., 2023; Liu et al., 2024). Repository memory reuses commits, linked issues, and module summaries for localization (Wang et al., 2026a). CodeMem maintains AST-guided repository context and session memory for iterative development, including checks for forgetting (Wang et al., 2026b). CodeDelegator instead separates persistent planning from ephemeral code execution, keeping debugging traces and failed attempts out of the planner's context (Fei et al., 2026a). These methods already address code structure and changing context. TriLink keeps task progress, reusable solutions, and ordered patch evidence connected as development continues, so retrieval can support interrupted work.

Memory and coding benchmarks. LoCoMo evaluates recall and temporal reasoning over dialogues, while LongMemEval additionally targets knowledge updates and abstention (Maharana et al., 2024; Wu et al., 2024). MemoryBench studies learning from user feedback, and MemoryArena tests memory use across dependent agentic sessions (Ai et al., 2025; He et al., 2026). In software engineering, SWE-bench evaluates issue resolution, CodeIF-Bench tests instruction retention during iterative coding, and SWE-Bench-CL examines learning and forgetting across chronological issues (Jimenez et al., 2024; Wang et al., 2025; Joshi et al., 2025). SWE-ContextBench measures reuse of related coding experience, while ContextBench evaluates retrieval using annotated repository contexts (Zhu et al., 2026; Li et al., 2026). CMB fixes an interleaved coding history for every compared mechanism and evaluates both QA and executable edits, connecting recalled decisions with their use in continued development. Table 1 summarizes the settings: long session denotes a single extended agent session; interleaving denotes interrupted and resumed coding tasks; memory denotes an explicit retention or reuse objective.

[1] https://github.com/volcengine/OpenViking

Table 1: Benchmark settings and evaluation targets. ✓: explicitly covered; ×: not a target of the protocol.

Benchmark	Long session	Coding	Interleaving	Memory	Repo tests
LoCoMo	×	×	×	✓	×
LongMemEval	×	×	×	✓	×
MemoryArena	×	×	×	✓	×
SWE-bench	×	✓	×	×	✓
CodeIF-Bench	×	✓	×	✓	✓
SWE-ContextBench	×	✓	×	✓	✓
CMB (ours)	✓	✓	✓	✓	✓

3 CodeMemoryBench: Memory in Long Coding Sessions

3.1 Evaluation unit and task construction

A CMB instance is a repository-level long session with multiple coding tasks. Figure 1 gives the construction-to-evaluation flow. For task i, the source artifact is $\tau_i = (R_i, b_i, F_i, p_i^\star, t_i^\star, T_i^+, T_i^=)$: repository, base commit, editable files, reference patch, test patch, F2P tests, and P2P tests. The reference patch and test specifications are used for construction and grading, not inserted as answers into memory. A point is one technical decision within a task; a turn is one developer instruction and the resulting agent/tool trace. Several point-level edits contribute to one task-level executable result.

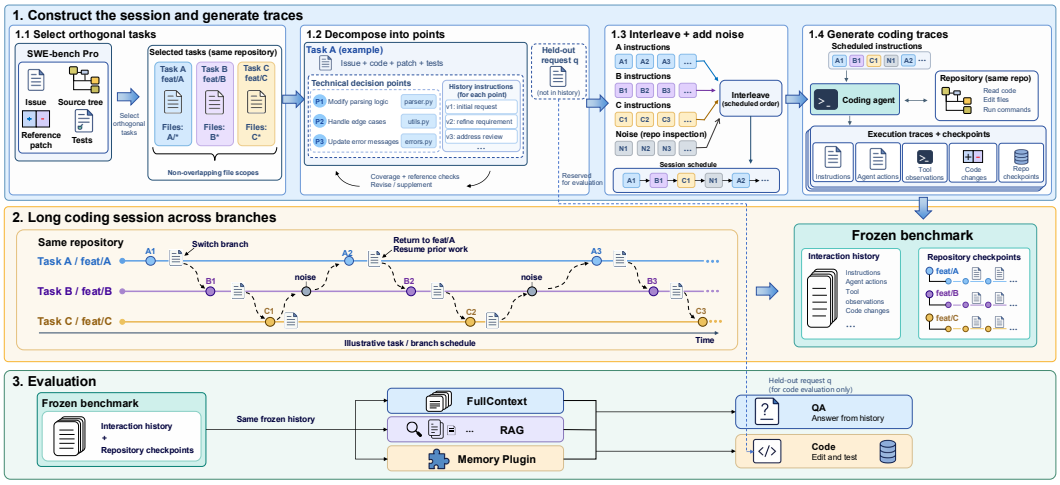


Figure 1: CMB construction: select repository tasks, derive decision points and history-dependent queries, execute an interleaved session, then freeze its text and repository checkpoints. Evaluation replays this history for each memory mechanism and measures recall and subsequent code edits.

1. Select tasks and prepare repository states. We use SWE-bench Pro (Deng et al., 2025) issue/patch/test artifacts. After filtering documentation, lock files, CI paths, and test-only edits, candidates are visited in increasing editable-file count. Task i is admitted only when $F_i \cap \bigcup_{j \in \mathcal{S}} F_j = \varnothing$ for the already selected set $\mathcal{S}$. This prevents overlapping patch scopes, but does not prove semantic independence. Each selected task receives a dedicated branch rooted at its own b_i, verified before trajectory generation.

2. Turn reference changes into decisions and queries. The constructor reads the issue, base-code excerpts, reference patch restricted to F_i, and test information. It first extracts points (k, f, g): a stable identifier, file anchor, and the required behavior g. Coverage checks compare these points

against the external patch/test specification and prompt targeted supplements for omissions. Each point is then expanded into three to five implementation instructions followed by a separate final request:

$$p_{ij} = (k_{ij}, f_{ij}, g_{ij}, u_{ij}^{1:n_{ij}}, q_{ij}), \qquad n_{ij} \in \{3, 4, 5\}. \tag{1}$$

Here j indexes points and $u_{ij}^{1:n_{ij}}$ is the ordered instruction list. The instructions express alternatives; q_{ij} refers back to the intended decision without restating its full implementation. Positional, ordinal, descriptive, and exclusion-based references prevent a single wording from defining the task. The constructor checks referential consistency, filename anchors, and answer restatement, retrying failed points. These are rule/LLM checks rather than exhaustive execution of every alternative.

3. Execute and freeze the long session. For each task, we concatenate its non-final point instructions; a round-robin merge interleaves these task streams, inserting noise turns that query repository state between rounds of tasks. The first encounter names the feature branch; subsequent requests refer to the task and require the agent to recall its branch. The generator uses that task's prior turns when producing a new response. Four generation shards are merged by their original global indices, producing one interleaved replay stream per repository. Thus the long session is the memory input; generation need not place every unrelated task in each LLM prompt. The frozen bundle retains instructions, traces, intended/actual branches, and repository checkpoints identified by point and segment.

4. Replay memory, then evaluate. Every mechanism ingests the same frozen text stream in order. The agent starts from a clone of the frozen repository. For each point, we retrieve memory for q_{ij} and reset the agent's message history; edits accumulate in the workspace across points. We then extract the task's patch from the designated branch relative to b_i, restricted to F_i, and grade it with $t_i^\star$:

$$y_i = v_i \, \mathbf{1}\left[(T_i^+ \cup T_i^=) \subseteq \mathrm{Passed}(b_i, \widehat{p}_i, t_i^\star)\right]. \tag{2}$$

Here $v_i = 1$ requires a nonempty patch and a valid test report; otherwise $v_i = 0$ and Passed is empty. The reported rate is $\sum_i y_i / N$ over N tasks. Final requests are absent from the initial frozen stream.

The QA track evaluates whether an agent can recover decisions from the same long session without editing the repository. We group related technical decision points into a set Q of 750 open questions, each with a reference answer. For every question, the memory mechanism retrieves from the frozen session and the agent produces an answer independently. For each answer, a judge derives m_q requirements from the reference and counts $0 \le h_q \le m_q$ matches. Invalid judgments have $m_q = 0$. The QA score is

$$S_{\mathrm{QA}} = \frac{1}{|Q|} \sum_{q \in Q} c_q, \qquad c_q = \begin{cases} h_q / m_q, & m_q > 0, \\ 0, & m_q = 0. \end{cases} \tag{3}$$

The denominator is judge-derived rather than fixed across runs, so QA is a diagnostic complement to execution grading.

3.2 Scale and memory demands

Table 4 summarizes the recorded histories and executable tasks. Each repository contributes 302–928 interaction turns (median 583), with 5,595 turns and 8.23M frozen-trajectory tokens in total. Of these turns, 5,377 are coding turns and 218 are noise turns. The histories cover 212 branches and 216 graded tasks across four languages. A turn is one developer instruction and its corresponding agent/tool response, not an individual tool call. Turn counts include the frozen pre-evaluation history; task counts refer to the execution-graded Code track. Frozen tokens measure the stored trajectory text under the cl100k_base tokenizer; they exclude system prompts, instruction templates, and repeated context sent across later agent steps.

These sessions require retaining earlier constraints, reconnecting task context after interruptions, and applying remembered decisions to code. We make these demands executable through a revision-intensive task family. Of 1,482 technical decision points, 1,284 (87%) refer to a non-latest revision; the point-depth counts are 760 at three revisions, 513 at four, and 209 at five. Version reasoning is thus an important diagnostic within CMB's long-session setting. Tables 7 and 4 summarize revision patterns and dataset scale; the aggregate results do not isolate each memory demand.

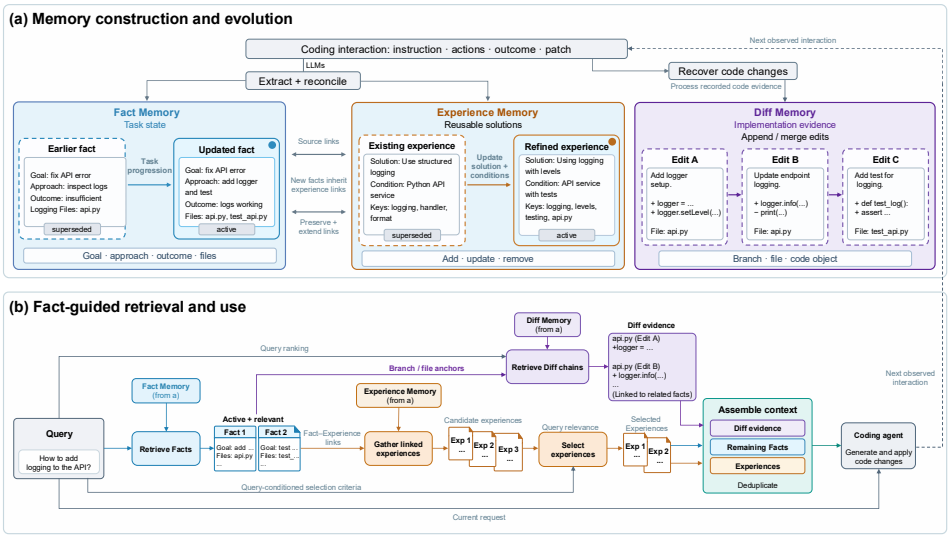


Figure 2: TriLink links evolving task facts, reusable experience, and code changes. The main retrieval path follows facts to relevant experience and diff evidence; direct experience matches can supplement it. New observations update memory. Diff maintenance uses recorded code evidence without LLM calls.

4 TriLink: Fact, Experience, and Diff Memory

TriLink connects task state, reusable solutions, and implementation changes so that the agent can recover their relationship when work resumes. Facts identify the task context for retrieving experience and code evidence. Each observation $o_t = (u_t, a_t, \Delta_t, c_t)$ contains developer input, agent/tool output, a code change, and repository/branch context. Memory evolves as

$$\mathcal{M}_t = (\mathcal{F}_t, \mathcal{E}_t, \mathcal{D}_t; \mathcal{L}_t, \mathcal{S}_t), \quad \mathcal{M}_{t+1} = \mathcal{U}(\mathcal{M}_t, o_{t+1}), \quad C_q = \mathcal{R}(\mathcal{M}_t, q). \tag{4}$$

$\mathcal{F}$, $\mathcal{E}$, and $\mathcal{D}$ store facts, experience, and diffs, keyed by stable identifiers. Using identifiers to denote their records, $\mathcal{L} \subseteq \mathcal{F} \times \mathcal{E}$ links facts to existing experience; $\mathcal{S} \subseteq \mathcal{F} \times \mathcal{F}$ records fact succession. Fact metadata anchors diff retrieval (Figure 2).

4.1 Three complementary representations

Facts summarize a task's goal, approach, outcome, and affected files, with an active/superseded status. Experiences describe reusable solutions: the task they address, solution steps, retrieval keys, and applicability conditions. Their source-fact references connect these abstractions to the observations that produced them. The distinction separates the state of a particular task from knowledge that may help with subsequent work.

Diffs preserve implementation evidence that summaries can omit. A chain d_κ holds ordered records (j, u_j, δ_j, t_j): revision index, instruction, patch fragment, and time. Its scope $\kappa = (b, f, o, s, I)$ combines branch, file, optional object identifier, symbol, and line region. Recorded diffs, checkpoints, or supported edit commands supply the patches. Symbol and nearby line-region matches extend chains within the same branch, file, and object; fragments with identical instruction text are merged. CMB's point/segment metadata supports construction-time grouping and checkpoint lookup, but is not used to select the answer chain at query time.

4.2 Coupled updates with preserved provenance

LLM-based extraction produces fact and experience drafts; reconciliation compares them with retrieved candidates. When work advances the same task, new facts supersede earlier states. Experience is added, refined, skipped as duplicate, or removed when explicitly invalidated. Refinement

updates the solution, retrieval keys, and conditions under the same identifier while preserving source references.

For a successful write, $\mathcal{F}^+$ denotes new facts, $\mathcal{E}^+$ surviving added/updated experience, and $P(f)$ the predecessors superseded by f. $\overline{\mathcal{L}}_t$ retains previous links to existing experience:

$$\begin{aligned}\mathcal{S}' &= \mathcal{S}_t \cup \{(p, f) : f \in \mathcal{F}^+,\ p \in P(f)\},\\ \mathcal{L}' &= \overline{\mathcal{L}}_t \cup (\mathcal{F}^+ \times \mathcal{E}^+) \cup \{(f, e) : f \in \mathcal{F}^+,\ p \in P(f),\ (p, e) \in \overline{\mathcal{L}}_t\}.\end{aligned} \tag{5}$$

Equation 5 preserves earlier experience links when task state advances and connects facts to experience written in the same batch. Reverse references are updated consistently. Superseded facts remain traversable but leave the active retrieval index; stale references are ignored. These links record shared provenance, not a one-to-one or causal correspondence.

The layers can change at different times. Fact and experience updates preserve their associations, while recorded edits independently extend Diff Memory. At retrieval, fact anchors reconnect the task state to this code history. This is the implemented co-evolution: shared observations and persistent access relations, with existence checks for partially persisted records.

4.3 Provenance-guided retrieval and context assembly

A query first retrieves active facts, then follows their links to candidate experience. One query embedding scores fact summaries and experience task/key matches. Let rank_{s_q} sort finite, scoped candidates by score, take_k keep the first k, and $N_{\mathcal{L}}(F)$ list linked experience in fact order. For positive relevance thresholds,

$$\begin{aligned}F_q &= \mathrm{take}_{k_F}\, \mathrm{rank}_{s_q}\{f \in \mathrm{Cand}_F(q) : s_q(f) \geq \theta_F\},\\ \widetilde{E}_q &= \{e \in \mathrm{Cand}_E(q) : s_q(e) \geq \theta_E\},\\ E_q &= \mathrm{take}_{k_E}\, \mathrm{live}\, \mathrm{dedup}\left([e \in N_{\mathcal{L}}(F_q) : e \in \widetilde{E}_q] \parallel \mathrm{rank}_{s_q}(\widetilde{E}_q)\right).\end{aligned} \tag{6}$$

Brackets preserve order and $\parallel$ concatenates lists. Linked experience must pass the query-relevance gate; direct matches supplement it after linked candidates. Deduplication keeps the first occurrence, and live removes missing records before truncation. Provenance therefore changes priority, while the query still determines relevance.

In parallel, selected facts locate Diff chains through branch information, with file anchors as a fallback. Query overlap with recorded instructions and added code ranks the chains; symbol mentions also guide ordering. The rendered block B_q includes the associated fact text, whose indices are $I_q \subseteq \{1, \ldots, |F_q|\}$. Context is assembled as

$$C_q = B_q \parallel \mathrm{text}([F_q[i] : i \notin I_q]) \parallel \mathrm{text}(E_q). \tag{7}$$

Facts without usable Diff evidence remain in the context. The Code configuration uses $k_F = k_E = 5$ with separate layer quotas. Diff rendering limits chains, code lines, and instruction length while retaining revision labels, using a character-budget heuristic. Diff maintenance requires no additional LLM or embedding calls.

5 Experiments

5.1 Setup and metrics

We evaluate with claude-sonnet-5 and deepseek-v4-flash in the same mini-swe-agent scaffold (Yang et al., 2024), using frozen initial histories and common prompt construction. The five CMB baselines are Mem0, MemOS, RAG, OpenViking, and RepoMemory. RAG performs plain vector retrieval over the frozen interaction history. FullContext provides a raw-history reference. SWE-ContextBench Lite_Related supplies an external 99-task evaluation, with NoMemory and FullContext references; the RAG baseline is not part of that suite's reported comparison.

The CMB main table reports resolved task counts by repository, Resolved (%) for Code, and QA score (%). The SWE-ContextBench table reports Patch N/A, test-level F2P/P2P micro-averages,

task-level F2P/P2P rates, and Resolved; the corresponding CMB metrics appear in Table 9 in the appendix. Missing-patch tasks remain failures in task-level metrics. CMB Resolved requires F2P and P2P together; the SWE-ContextBench harness used here determines resolution from F2P and reports P2P separately.

5.2 MAIN RESULTS

Table 2: CMB results across nine repositories. Repository columns and Total report resolved task counts; Code and QA report Resolved and QA scores (%), respectively. **Bold**: best; underline: second best, excluding references (grey method names). Improvement compares Code/QA with the best other memory method for each model, in percentage points.

Model	Method	Resolved tasks by repository									Overall		
		vu	fl	na	tu	No	qu	el	ol	an	Total	**Code↑**	**QA↑**
Claude Sonnet-5	FullContext	6	10	12	8	11	21	19	15	25	127	58.8	56.1
	RAG	3	**7**	2	5	8	11	9	12	**18**	75	34.7	48.0
	Mem0	**5**	**7**	4	5	8	10	**13**	11	16	79	36.6	37.8
	MemOS	**5**	6	**6**	6	8	**13**	12	7	12	75	34.7	51.9
	OpenViking	4	6	3	6	9	6	**13**	6	10	63	29.2	46.8
	RepoMemory	2	4	2	3	7	6	5	4	5	38	17.6	18.7
	TriLink	**5**	**7**	**6**	**7**	**11**	11	**13**	**15**	**18**	**93**	**43.1**	**58.2**
Improvement (pp)												6.5↑	6.3↑
DeepSeek v4-flash	FullContext	5	7	4	3	6	12	7	4	13	61	28.2	53.4
	RAG	**5**	5	4	2	7	5	8	8	14	58	26.9	62.2
	Mem0	1	5	4	3	6	6	6	8	8	47	21.8	40.0
	MemOS	3	4	**7**	2	6	**7**	13	**10**	9	61	28.2	51.4
	OpenViking	1	4	4	1	4	2	2	3	7	28	13.0	38.6
	RepoMemory	2	3	2	2	4	2	5	2	3	25	11.6	9.6
	TriLink	**5**	**6**	6	**5**	**10**	**7**	**14**	9	**15**	**77**	**35.6**	**64.2**
Improvement (pp)												7.4↑	2.0↑

On CMB, TriLink improves Resolved over the strongest existing memory by 6.5 and 7.4 percentage points with Sonnet-5 and flash, respectively (Table 2). QA improves by 6.3 and 2.0 points. Resolved aggregates task outcomes rather than averaging repository percentages.

The tracks rank methods differently. With Sonnet-5, Mem0 beats RAG on Resolved (36.6% versus 34.7%) but trails on QA (37.8% versus 48.0%). With flash, RAG approaches TriLink on QA (62.2% versus 64.2%) while lagging on Resolved (26.9% versus 35.6%). These rankings motivate evaluating both recall and executable edits. The tracks use different units and scoring rules, so their score difference is not an implementation failure rate.

FullContext is sensitive to the backbone: it reaches 58.8% with Sonnet-5 and 28.2% with flash. Raw history can help, but its use still depends on the agent's ability to identify and apply relevant details.

On SWE-ContextBench, TriLink reaches 47.5% and 28.3% Resolved, exceeding the strongest competing memories at 45.5% and 25.3% (Table 3). Several plugins fall below NoMemory: retrieved context must help the next action, and its presence alone does not ensure a benefit. These fixed-run comparisons support usefulness beyond CMB without establishing statistical significance.

5.3 ABLATION STUDY

On CMB, removing Diff Memory, Experience Memory, or Fact Memory reduces Resolved from 35.6% to 20.8%, 19.9%, or 19.9%, respectively (Figure 3). Removing Diff Memory also lowers SWE-ContextBench Resolved from 28.3% to 21.2%. The losses are 14.8 and 7.1 percentage points:

Table 3: SWE-ContextBench Lite_Related results (%). **Bold**: best; underline: second best, excluding references (grey method names). Improvement is relative to the best other memory method for each metric, in percentage points; - denotes a decrease after accounting for lower-is-better Patch N/A.

Model	Method	Patch	Tests		Tasks		Resolved
		N/A↓	F2P↑	P2P↑	F2P↑	P2P↑	↑
Claude Sonnet-5	FullContext	5.1	71.9	99.8	37.4	85.9	37.4
	NoMemory	4.0	72.2	98.9	43.4	86.9	43.4
	Mem0	6.1	70.8	97.9	38.4	81.8	38.4
	MemOS	**4.0**	70.3	**99.0**	39.4	88.9	39.4
	OpenViking	**4.0**	70.5	98.8	36.4	83.8	36.4
	RepoMemory	6.1	73.5	98.8	45.5	85.9	45.5
	TriLink	6.1	**80.4**	98.9	**47.5**	**89.9**	**47.5**
Improvement (pp)		-	6.9↑	-	2.0↑	1.0↑	2.0↑
DeepSeek v4-flash	FullContext	8.1	52.5	99.2	23.2	76.8	23.2
	NoMemory	6.1	57.7	97.9	20.2	80.8	20.2
	Mem0	10.1	24.6	98.8	25.3	78.8	25.3
	MemOS	**2.0**	23.4	99.2	25.3	**84.8**	25.3
	OpenViking	7.1	**55.5**	97.9	19.2	76.8	19.2
	RepoMemory	7.1	22.6	**99.5**	19.2	79.8	19.2
	TriLink	3.0	23.4	99.4	**28.3**	**84.8**	**28.3**
Improvement (pp)		-	-	-	3.0↑	0.0	3.0↑

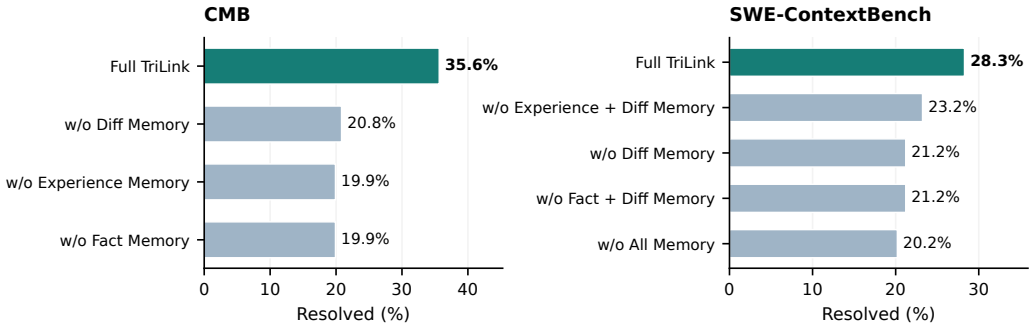


Figure 3: Layer ablations with deepseek-v4-flash. Labels identify removed components (Section 4), including joint removals on SWE-ContextBench. Bar labels show Resolved (%).

concrete code evidence helps in both settings, with a larger observed effect in CMB's revision-intensive sessions.

The components interact through retrieval: facts anchor Diff chains, while experiences provide reusable procedures. On SWE-ContextBench, facts alone reach 23.2%, above the 21.2% of facts and experience without diffs. Additional representations therefore need not improve Resolved. Layer removal changes both retrieval and context; these experiments do not isolate the links themselves. Full TriLink is best or tied across the four CMB variants in eight of nine repositories, with qutebrowser the exception. Table 10 provides the repository breakdown.

5.4 Analysis experiment: performance under repeated code revisions

We group Code tasks by the largest number of recorded revisions among their decision points. Figure 4 shows the 3-, 4-, and 5-revision groups for both backbones; Table 8 gives the exact values. The three-revision group contains only 12 tasks and is shown for completeness.

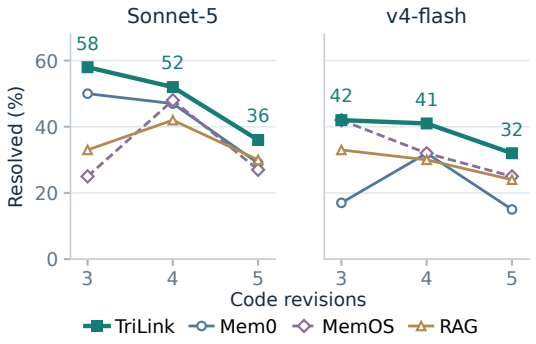


Figure 4: Resolved on CMB across 3–5 revisions.

TriLink reaches 58%, 52%, and 36% with Sonnet-5, and 42%, 41%, and 32% with flash. In the five-revision group, the strongest shown memory baseline reaches 30% with Sonnet-5 and 25% with flash. The advantage therefore persists when several plausible implementations coexist in the history, a setting where retrieving the right topic alone leaves the agent to distinguish among competing decisions.

From four to five revisions, TriLink's flash score drops nine points, versus seven for MemOS. These strata contain different tasks and may differ in complexity, so the comparison does not isolate the causal effect of additional revisions.

5.5 Analysis experiment: token efficiency

Figure 5 compares injected memory tokens and Resolved on SWE-ContextBench/flash; the star marks TriLink and the dashed line NoMemory.

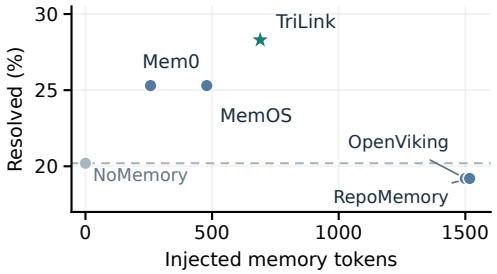


Figure 5: Injected memory tokens and Resolved on SWE-ContextBench/flash.

Retrieval volume alone does not explain Resolved scores. TriLink reaches 28.3% Resolved with 690 injected tokens, compared with 25.3% at 479 tokens for MemOS and 25.3% at 257 tokens for Mem0. RepoMemory and OpenViking inject roughly 1,500 tokens yet resolve 19.2% of tasks, below the 20.2% NoMemory baseline. TriLink uses less than half their retrieved context while achieving a higher Resolved score.

Mem0 and MemOS achieve equal scores with different injection sizes; TriLink adds 211 tokens over MemOS for a 3.0-point gain. This tradeoff favors evidence that supports the next action over retrieval volume alone. These are observed operating points, without matched token budgets. Tokens cover retrieved memory only, excluding other agent input and memory construction costs. Table 11 lists the plotted values.

6 Discussion and Conclusion

CMB evaluates whether memory supports continued coding through frozen repository sessions, diagnostic recall, and executable edits. The results motivate linking task progress, reusable solutions, and code evidence: TriLink improves over existing memory mechanisms on CMB and remains useful on SWE-ContextBench.

The evaluation has limits. Sessions are synthetic and deliberately emphasize interleaving and revision. CMB replay supplies code-object metadata that assists diff grouping; robustness without that metadata needs separate evaluation. QA rubrics and construction checks depend on LLM judgments. Further work should study naturally occurring developer sessions and explicit ablations of the links between the three layers.

AI USE STATEMENT

Generative models produced the coding trajectories and the evaluated agent outputs. AI tools also assisted with drafting, code inspection, analysis scripts, and figure preparation. The authors are responsible for verifying the final text, artifacts, and measurements before submission.

ETHICS STATEMENT

The benchmark uses public open-source repositories and synthetic interaction sessions rather than personal developer conversations. Repository and dependency licenses apply to distributed artifacts. Release preparation must exclude credentials and unrelated private data from traces and supplementary files.

REPRODUCIBILITY STATEMENT

CMB construction and evaluation are described in Section 3, with construction checks, trajectory generation, and replay details in Appendix A.3.1. TriLink's memory representations, update rules, and retrieval procedure are specified in Section 4; implementation settings and context assembly are detailed in Appendix A.3.2. Dataset statistics and supplementary results are provided in Appendices A.1 and A.2.

REFERENCES

Qingyao Ai, Yichen Tang, Changyue Wang, Jianming Long, Weihang Su, and Yiqun Liu. Memorybench: A benchmark for memory and continual learning in llm systems. *arXiv preprint arXiv:2510.17281*, 2025.

Ting Bai, Jiayang Fan, Xiaoshuai Wen, Jiazheng Kang, Hengzhi Lan, Ruochen Zhao, Pingzheng Wu, Zepeng Zhang, Yutian Zhong, Gezi Li, et al. Survey on AI memory: Theories, taxonomies, evaluations, and emerging trends.

Yaoqi Chen, Haibin Lai, Yuru Feng, Chuyu Han, Qianxi Zhang, Baotong Lu, Menghao Li, Xinjiang Wang, Zhirui Wang, Shusen Xu, et al. Beyond semantic organization: Memory as execution state management for long-horizon agents. *arXiv preprint arXiv:2606.06090*, 2026.

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.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, 2026.

Tianxiang Fei, Cheng Chen, Yue Pan, Mao Zheng, and Mingyang Song. CodeDelegator: Mitigating context pollution via role separation in code-as-action agents. *arXiv preprint arXiv:2601.14914*, 2026a.

Tianxiang Fei, Mingyang Song, Mao Zheng, and Xiang Yu. Memory beyond recall: A dual-process cognitive memory system for self-evolving LLM agents. *arXiv preprint arXiv:2606.09483*, 2026b.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Carlos E Jimenez, 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*, volume 2024, pp. 54107–54157, 2024.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. Swe-bench-cl: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory OS of AI Agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25972–25981. Association for Computational Linguistics, 2025.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents. *arXiv preprint arXiv:2602.05892*, 2026.

Zhiyu Li, Chenyang Xi, Chunyu Li, Ding Chen, Boyu Chen, Shichao Song, Simin Niu, Hanyu Wang, Jiawei Yang, Chen Tang, et al. Memos: A memory os for ai system. *arXiv preprint arXiv:2507.03724*, 2025.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*, 2024.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Boshi Wang, Weijian Xu, Yunsheng Li, Xuemei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In *International Conference on Learning Representations*, volume 2026, pp. 111266–111285, 2026a.

Peiding Wang, Li Zhang, Fang Liu, Lin Shi, Minxiao Li, Bo Shen, and An Fu. Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation. *arXiv preprint arXiv:2503.22688*, 2025.

Peiding Wang, Li Zhang, Fang Liu, Chongyang Tao, and Yinghao Zhu. Codemem: Ast-guided adaptive memory for repository-level iterative code generation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16903–16917, 2026b.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Jiayuan Zhu, Junde Wu, Minhao Hu, Shengda Zhu, Jiazhen Pan, Weixiang Shen, Yijun Yang, Fenglin Liu, Jianye Hao, Yueming Jin, et al. Swe context bench: A benchmark for context learning in coding. *arXiv preprint arXiv:2602.08316*, 2026.

A Additional Details and Results

A.1 Dataset Statistics and Analysis

Repository histories and evaluation units

CMB separates the history available to memory from the requests used to evaluate it. Turns measure developer–agent interactions in the frozen history; points identify technical decisions within a task; Code tasks are graded by repository execution. QA questions examine decisions from those same histories. Keeping these units separate is necessary to interpret both the scale of the benchmark and the aggregate scores.

Table 4 reports the history for each repository, in the order used by the main CMB table. Coding and noise turns sum to the full session length. Frozen tokens count the stored trajectory text, including agent and tool responses. They describe how much history must be represented, rather than the input consumed by a single evaluation request.

Table 4: Frozen histories and Code tasks by repository. Token counts use cl100k_base and exclude system prompts, templates, and repeated context.

Repository	Language	Turns	Frozen tokens	Coding	Noise	Tasks
vuls	Go	324	591.2K	305	19	16
flipt	Go	468	952.7K	444	24	18
navidrome	Go	546	1.17M	520	26	20
tutanota	TS	302	509.4K	282	20	14
NodeBB	JS	774	868.1K	742	32	23
qutebrowser	Py	583	727.5K	563	20	27
element-web	TS	836	1.12M	808	28	28
openlibrary	Py	928	1.19M	899	29	30
ansible	Py	834	1.11M	814	20	40
Total	**4**	**5,595**	**8.23M**	**5,377**	**218**	**216**

Language groups: Go 54 tasks; TypeScript/JavaScript 65 tasks; Python 97 tasks.

Session length and evaluation size vary independently. For example, openlibrary has 928 turns for 30 Code tasks, whereas ansible has 834 turns for 40 tasks. Thus the repository with more graded requests need not have the longest history. This variation matters because memory must retain useful context throughout the recorded work, while each Code task contributes one outcome to Resolved.

Table 5 connects those tasks to the underlying decisions and diagnostic questions. There are 6.9 points per Code task on average, with repository averages ranging from 5.0 in vuls to 8.6 in openlibrary. These averages include the four tasks without constructed decision points. QA groups related decisions into 750 questions; a question may cover several points, so question counts and point counts are not interchangeable.

Table 5: Code tasks, technical decision points, and QA questions by repository. Averages use all Code tasks.

Repository	Code tasks	Points	QA questions	Points/task	QA/task
vuls	16	80	48	5.0	3.0
flipt	18	124	71	6.9	3.9
navidrome	20	145	75	7.3	3.8
tutanota	14	83	41	5.9	2.9
NodeBB	23	196	101	8.5	4.4
qutebrowser	27	158	80	5.9	3.0
element-web	28	223	107	8.0	3.8
openlibrary	30	258	114	8.6	3.8
ansible	40	215	113	5.4	2.8
Total	216	1,482	750	6.9	3.5

The two tracks consequently place different demands on memory. Code requires the relevant decisions to be applied together in a working edit. QA can reveal which requirements the agent recalls even when its account is incomplete. Their shared histories make the tracks complementary, but their scores should be interpreted using their own grading units.

Historical references and code revisions

Table 6 describes how a final Code request refers to earlier work. Positional cues identify a decision through its place in the history, while ordinal cues specify a particular attempt. Descriptive cues name a distinguishing behavior or structure; exclusion cues eliminate rejected alternatives. These forms require the retrieved context to preserve enough information to tell related decisions apart.

Table 6: Reference forms in the 1,482 point-level Code requests.

Reference	Cue in the final request	Points	Share (%)
Positional	Locate an earlier decision by its position in the history.	319	21.5
Ordinal	Identify a decision by its numbered attempt.	363	24.5
Descriptive	Identify a decision through a distinguishing feature.	400	27.0
Exclusion	Recover a decision by ruling out the alternatives.	400	27.0
Total		1,482	100.0

Each reference form accounts for between 21.5% and 27.0% of the points, so none dominates the requests. The categories describe how the Code requests are worded. They do not partition the QA track or provide separate execution scores for each form, since a task can contain several kinds of point-level request.

Table 7 separates the requested revision from the number of alternatives in the history. Earlier revisions are requested at 1,284 points, while 198 target the latest one. A policy that always uses the most recent edit therefore does not recover the intended decision at most points. The second block records how many revisions were present, regardless of which one the final request selects.

Table 7: Requested and recorded revisions. Each block partitions the same 1,482 decision points.

Characteristic	Category	Points	Share (%)
Requested revision	Earlier	1,284	86.6
	Latest	198	13.4
Recorded revisions	3	760	51.3
	4	513	34.6
	5	209	14.1

Three-revision points are the largest group, but many tasks contain at least one point with more revisions. This explains why point counts and task counts have different distributions: the task grouping uses the maximum revision count among its points, rather than the most common count across the repository.

Performance under repeated code revisions

Table 8 provides the task-level results behind the main-text analysis. The three strata contain 12, 88, and 112 tasks, respectively. The four tasks without constructed points remain outside this stratification. One task changes the smallest group by over eight percentage points, so its results are reported without ranking the methods.

Table 8: CMB Resolved by task revision count (%). Bold marks the best reported memory result in the two larger strata; grey names indicate references.

Model	Method	Revisions		
		3	4	5
Claude Sonnet-5	TriLink	58	**52**	**36**
	Mem0	50	47	29
	RAG	33	42	30
	MemOS	25	48	27
	RepoMemory	8	25	16
DeepSeek v4-flash	TriLink	42	**41**	**32**
	MemOS	42	32	25
	FullContext	33	30	26
	RAG	33	30	24
	Mem0	17	32	15

TriLink has the highest Resolved score among memory methods in both larger strata under each backbone. With Sonnet-5, the ordering of other methods changes: MemOS reaches 48% in the four-revision group, just above Mem0 at 47%, but falls below it in the five-revision group, at 27% versus 29%. With flash, TriLink reaches 32% in the latter group, compared with 25% for MemOS and 24% for RAG. The advantage persists across these groups, although the absolute scores show that repeated revision remains difficult.

The groups contain different tasks, so their score differences also reflect differences in task and repository composition. They describe where a method succeeds within CMB; they do not measure the effect of adding one revision to an otherwise identical request. This distinction is especially relevant when comparing a method's decline with its absolute Resolved score.

A.2 Supplementary Results and Ablations

Reading the execution metrics alongside QA

Table 9 expands the Code and QA results in the main paper. Patch N/A captures failures to produce an applicable edit. F2P measures the requested behavior, while P2P checks whether existing behavior is preserved. Test-level rates pool individual tests; task-level rates require every test in the corresponding group to pass. Resolved requires both groups to pass and remains the main measure of completed Code tasks.

These columns can rank methods differently. With Sonnet-5, MemOS reaches an F2P test score of 80.6%, above TriLink's 65.8%, but its F2P task score is 38.4%, below TriLink's 47.2%. Tasks contribute different numbers of tests, and passing some checks does not complete a request. The improvement row uses the strongest other memory method in each column, accounting for the lower-is-better direction of Patch N/A.

Table 9: Detailed CMB execution metrics and QA scores (%). **Bold**: best; underline: second best, excluding references (grey names). Improvement is in percentage points; - denotes a decrease.

Model	Method	Patch	Tests		Tasks		Resolved	QA
		N/A↓	F2P↑	P2P↑	F2P↑	P2P↑	↑	↑
Claude Sonnet-5	FullContext	10.2	89.4	98.5	62.0	81.5	58.8	56.1
	RAG	17.1	47.8	94.1	39.4	69.4	34.7	48.0
	Mem0	13.9	48.1	96.1	40.3	68.1	36.6	37.8
	MemOS	14.4	**80.6**	96.3	38.4	70.4	34.7	51.9
	OpenViking	15.8	54.3	93.9	33.5	67.0	29.2	46.8
	RepoMemory	19.4	33.5	95.8	20.8	59.3	17.6	18.7
	TriLink	**13.0**	65.8	**97.6**	**47.2**	**74.5**	**43.1**	**58.2**
Improvement (pp)		0.9↑	-	1.3↑	6.9↑	4.1↑	6.5↑	6.3↑
DeepSeek v4-flash	FullContext	16.7	38.9	94.9	29.6	70.4	28.2	53.4
	RAG	16.7	36.5	**96.9**	29.6	68.5	26.9	62.2
	Mem0	13.9	40.3	96.8	25.5	66.2	21.8	40.0
	MemOS	14.4	**72.9**	96.3	31.9	70.4	28.2	51.4
	OpenViking	26.9	21.9	89.1	14.4	53.2	13.0	38.6
	RepoMemory	24.5	28.6	92.1	13.4	52.8	11.6	9.6
	TriLink	**11.6**	52.0	**96.9**	**38.9**	**72.7**	**35.6**	**64.2**
Improvement (pp)		2.3↑	-	0.0	7.0↑	2.3↑	7.4↑	2.0↑

P2P shows the same difference in granularity. TriLink passes 97.6% of individual P2P tests with Sonnet-5, while 74.5% of tasks pass their full P2P group. A small share of failed tests can therefore affect a substantial share of tasks. Both views are useful: one describes the amount of preserved behavior, and the other whether a task meets all of its preservation requirements.

QA adds a different view of the remembered context. Under flash, RAG reaches 62.2% on QA, close to TriLink's 64.2%, while their Resolved scores are 26.9% and 35.6%. A recalled requirement may be incomplete or may not lead to a successful implementation. These aggregate scores motivate evaluating both tracks; they do not identify which individual QA errors caused a Code failure.

How layer contributions vary across repositories

Table 10 compares the layer ablations by repository under flash. Full TriLink is best or tied in eight repositories, with varying contributions from each layer. The Overall row is weighted by task count.

Table 10: CMB ablations by repository with deepseek-v4-flash: Resolved (%). Bold marks the best variant in each row; Overall is weighted by tasks.

Repository	TriLink	Removed memory		
		Diff	Experience	Fact
vuls	**31.3**	18.8	12.5	6.3
flipt	**33.3**	22.2	16.7	22.2
navidrome	**30.0**	25.0	10.0	15.0
tutanota	**35.7**	7.1	14.3	**35.7**
NodeBB	**43.5**	21.7	34.8	21.7
qutebrowser	25.9	29.6	**33.3**	29.6
element-web	**50.0**	25.0	25.0	25.0
openlibrary	**30.0**	16.7	13.3	10.0
ansible	**37.5**	17.5	15.0	17.5
Overall	**35.6**	20.8	19.9	19.9

On navidrome, removing Experience Memory lowers Resolved from 30.0% to 10.0%, while removing Diff Memory gives 25.0%. On tutanota, removing Fact Memory leaves Resolved at 35.7%, but removing Diff Memory lowers it to 7.1%. Aggregate reductions therefore conceal different dependencies across repositories.

On qutebrowser, all three ablated variants outperform full TriLink; removing Experience Memory gives the highest Resolved score. Some retrieved material may be unhelpful, but establishing the cause requires inspecting the retrieved records and subsequent agent actions.

Injected memory tokens and Resolved

Table 11 shows the tradeoff between injected memory tokens and Resolved. Mem0 uses 257 tokens, while MemOS uses 479 for the same 25.3% Resolved. TriLink achieves the highest score with 690 tokens.

Table 11: Injected memory tokens and Resolved on SWE-ContextBench/flash.

Method	Injected memory tokens	Resolved (%)
NoMemory	0	20.2
Mem0	257	25.3
MemOS	479	25.3
TriLink	690	28.3
OpenViking	1,500	19.2
RepoMemory	1,517	19.2

OpenViking and RepoMemory each inject about 1,500 tokens yet score below NoMemory on Resolved, so more retrieved text does not consistently improve this metric. These measurements exclude other agent input, memory construction, and execution costs. They compare the reported configurations; rankings under matched token budgets remain untested.

A.3 Implementation Details

This section specifies construction checks, stored information, and retrieval settings. The evaluated input is the frozen history; regenerating synthetic conversations would produce a different input under the same procedure.

A.3.1 Benchmark construction and replay

Each source task retains its base revision, editable files, issue, reference patch, test patch, and F2P/P2P test identifiers. After file-disjoint selection (Section 3), each task branch is initialized at its own base revision and checked against it. The branches coexist in one repository while preserving the original task states.

Construction separates decision specification from dialogue generation. The first prompt receives the issue, base-code excerpts, the patch restricted to editable files, and test requirements. It produces decision identifiers, file anchors, and target behaviors. Overloaded decisions are split further. Coverage checks compare the combined decisions with the patch and tests; missing behaviors and newly introduced symbols trigger targeted supplements.

A second prompt generates three to five implementation instructions per decision, followed by a final Code request. The instructions describe distinct alternatives at the same location, including the intended behavior, and are prompted to name the file and symbol. The final request uses its assigned historical reference form and is reserved for evaluation without restating the implementation.

Point-level checks assess filename anchoring, answer recoverability, and answer restatement. Failed instructions are regenerated with the rejection reason, preserving the decision identifier and target behavior; the default permits three additional attempts. Dialogue coverage is then checked against the patch and tests. Missing decisions trigger new points, while specified decisions omitted from the dialogue trigger instruction repair. Specification and dialogue coverage each allow up to three rounds. Deduplication follows, retaining unresolved diagnostics. These checks combine rules and LLM judgments; they do not certify that every intermediate implementation passes the task tests.

The round-robin schedule takes the next non-final instruction from each task and is computed before generation is divided across four shards. After each block of as many coding turns as selected tasks, a noise turn asks about commits, working-tree state, files, or branches. A task's branch is named only at its first encounter. The construction agent receives that task's prior turns. The saved histories record claude-sonnet-5 as this agent. Each coding or noise turn runs for at most 100 agent steps.

Each turn stores its global position, instruction, trace, and task identity. Coding turns additionally store point and segment identifiers and intended and observed branches; checkpoints preserve the resulting code. Shard outputs are merged by global position and their branches collected into the frozen repository. Auditing checks instruction coverage, unique point–segment pairs, and replay order. Observed branches retain evidence of generation errors.

Initial ingestion reads the frozen stream without rerunning the construction agent. Code evaluation retrieves context per final request and resets the agent's conversation while retaining workspace edits. Each task patch is extracted from its designated branch relative to its base and graded in the corresponding repository environment. QA records group related decisions into open questions with reference answers and covered-point identifiers. QA retrieval is read-only and answers are generated independently, using the grading rules in Section 3.

A.3.2 TriLink implementation

SQLite stores records and relations; separate vector indices cover active fact statements and experience task descriptions/search keys. Experience matches are grouped by identifier using the highest similarity score. The default embedder is text-embedding-3-small with 1,536 dimensions. Each CMB repository has its own memory namespace. Retrieval receives no target-branch filter; it must recover that information from the evidence.

One extraction call can produce a task-state fact and reusable experiences from an observation. Fact reconciliation compares up to five active candidates to identify progress on the same task. A successor receives a new identifier, inherits experience links, and supersedes its predecessors while

retaining their records. Its evolution summary inspects up to eight predecessors and is returned as context, but is excluded from the fact embedding.

Experience reconciliation compares up to three candidates and adds, updates, skips, or deletes records. Updates retain identifiers and source-fact references. After persistence, new facts and added/updated experiences from the same write are linked in both directions (Equation 5); missing records are filtered on retrieval. Fact Memory and Experience Memory can ingest subsequent Code-agent observations after the initial frozen history.

Diff ingestion uses recorded patches, checkpoint differences, or supported edit commands, without LLM reconciliation. Noise and final-evaluation turns are excluded. Chains group changes by branch, file, and optional code object, using symbol matches and nearby line regions with a 20-line tolerance. Fragments with identical instructions share one revision index. CMB point/segment identifiers support ingestion and checkpoint lookup, but not query-time chain selection. Each chain also retains an excerpt around its latest available checkpoint.

One query embedding scores active facts and experience candidates. Code uses separate quotas of five facts and five experiences; QA defaults to seven each, with relevance thresholds of 0.20 for both layers. The base experience candidate request is $\max(4k_E, 8)$ before scope filtering and deduplication. Linked experiences are visited in fact order and must pass the query threshold; absence from the scored pool excludes a record. Other eligible direct matches follow. Missing records and duplicates are removed before applying the experience quota.

In parallel, facts locate Diff chains by branch, falling back to file anchors. Symbol mentions guide ordering; query overlap with instructions and added code ranks competing chains. The renderer augments the first fact with usable evidence with up to three chains, retaining revision labels. Table 12 lists the adapter defaults.

Table 12: Default TriLink adapter settings.

Setting	**Default**
Fact / Experience quota, Code	5 / 5
Fact / Experience quota, QA	7 / 7
Fact / Experience relevance threshold	0.20 / 0.20
Fact / Experience reconciliation candidates	5 / 3
Predecessors used in a fact evolution summary	Up to 8
Facts augmented with Diff evidence	First 1 with usable chains
Diff chains per augmented fact	Up to 3
Revisions per chain with detailed code	Up to 8
Code excerpt per revision	6 lines, 480 characters
Developer instruction per revision	800 characters
Latest checkpoint excerpt per chain	900 characters

Context contains the Diff block, remaining facts, and experiences, in that order. Facts included in the block are removed from the separate list; facts without usable diffs remain available. The two layer quotas apply separately.