SMU SINGAPORE MANAGEMENT UNIVERSITY
School of Computing and Information Systems

Does Cross-Instance Memory Help an LLM Software-Engineering Agent?

SIM Kim Sia
Supervisors: Prof. Jiang Lingxiao, Prof. Dai Bingtian

Motivation

Large Language Model (LLM) coding agents are stateless. Every issue starts from zero, and the experience from earlier issues in the same repository is discarded.

We call the ability to retain and retrieve that experience *cross-instance memory*. This capstone investigates whether it improves the agent's performance.

RQ1. Does memory improve task success?

RQ2. How does memory change the agent's behaviour?

RQ3. What mechanism explains both?

Method & Key Result

Method

- Matched-pair experiments: two identical configurations, except one using memory, graded by the benchmark's own test harness
- A mini-SWE-agent-based harness with the Kimi K2.5 model on the SWE-bench Pro benchmark, validated against published baselines
- Six repositories in three languages; complete coverage of element-web (all 56 instances); dense-23 windows replicated ×3
- Every shell command in every trajectory mined and classified as explore, edit, or verify
- Predictions written down before running; run-to-run noise floor measured

RQ1: the effect on task success is zero

Arm	Resolved
memory-OFF	23 / 56 (41%)
memory-ON	21 / 56 (38%)

McNemar p = 0.774: statistically indistinguishable from zero. Memory rescued 5 instances and broke 7 (net −2).

Density moderator

Where fixes cluster in time and code (median commit gap 4.5 days), memory gives a small, replicated +1 to +2 passes. Across sparse tails it has nothing to retrieve. The benefit is smaller than the ±3-instance run-to-run noise floor.

Contributions

1. A harness-validated null at 100% repository coverage
2. The verification-erosion finding, with its boundary located at checks that are skippable and expensive (two predictions written down beforehand; both held)
3. Verification behaviour is orthogonal to success (three interventions, zero recoveries)
4. The reasoning-bound mechanism and a portable selection rule
5. Reproducible infrastructure: a public harness fork, mining and analysis scripts, and ten checksummed release archives

Findings

RQ2: memory erodes verification

Memory halves the agent's verification activity on element-web (2.74 to 1.27 verify actions per task). The rate of submitting patches with zero verification rises:

element-web: 6% → 50% (p < 0.001) **webclients: 30% → 74% (p = 0.007)**

	Check cheap	Check expensive
Check forced	none	flat (Go ×2)
Check optional	flat (Python ×2)	**ERODES (TypeScript ×2)**

Repository	optional?	expensive?	zero-verify OFF → ON	result
element-web (TS)	yes	yes (~13 s)	6% → 50% (p < 0.001)	ERODES
webclients (TS)	yes	yes	30% → 74% (p = 0.007)	ERODES
qutebrowser (Py)	yes	no (< 1 s)	3% → 4%	flat
openlibrary (Py)	yes	no	0% → 0%	flat
flipt (Go)	no	yes	4% → 4%	flat
teleport (Go)	no	yes	0% → 4%	flat

Erosion appears only where the pre-submission check is both skippable and expensive. We wrote down two of these cells as predictions before running (webclients erodes, openlibrary stays flat). Both held.

Restoring verification recovers zero passes

Intervention	Behaviour	Passes
Forced tsc gate	fires 23/23	0 recoveries
Verification-aware memory	tsc per task ×2 (0.6 → 1.26)	8/23, flat
Self-authored test	6/23 complied	7/23, flat; 5 of 6 still failed

All three interventions moved the intended behaviour, and none recovered a pass. Type-correctness is orthogonal to task-correctness.

RQ3: the failures are reasoning-bound

Across all eight arms, 10 of 23 dense-window instances never pass. Nine of the ten are logic failures rather than localization failures: the agent edits the exact files whose tests fail, writes 55 to 424 added lines, and still fails. It found the right code and wrote the wrong fix. Trajectory memory carries navigation, which is already solved, so memory was solving a problem the agent did not have.

Cold-start trap: the subsystem most in need of transferable knowledge (Voice Broadcast, 6 of 10 persistent failures) has no solved exemplar to transfer from.

Takeaway

Before building memory for a coding agent, classify its existing failures using public trajectories and gold patches. No new runs are needed:

- When localization failures dominate, trajectory memory can lift passes
- When logic failures dominate (element-web was 9 of 10), memory is inert by construction
- When the check is skippable and expensive, expect erosion and pair memory with an enforced gate

Reference

Deng, X., et al. (2025). SWE-bench Pro: Can AI agents solve long-horizon software engineering tasks? arXiv:2509.16941.

Lindenbauer, T., et al. (2025). From knowledge to noise: CTIM-Rover and the pitfalls of episodic memory in software engineering agents. arXiv:2505.23422.

Moonshot AI. (2026). Kimi K2.5 [Large language model].

Peng, C., et al. (Organizers). (2026). Intelligent coding assistant enhanced with memory [Competition]. FSE AIWare 2026 Workshop.

Yang, J., et al. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. NeurIPS 37.

Data and code: https://github.com/simkimsia/capstone-mem-artifacts

Acknowledgements: Prof. Jiang Lingxiao, Prof. Dai Bingtian, SMU School of Computing and Information Systems.