Does AI Memory Actually Work for Coding Agents?

A Controlled Benchmark of Persistent Memory
in Production Codebase Tasks

Markus Sandelin

Independent Research
markus@stompy.ai

February 2026

Abstract

Every AI memory company claims 80–95% token savings. We tested ours on a production codebase and measured 15–28%. This paper explains why that's the honest number—and why nobody else has published one.

We present the first controlled benchmark of persistent AI memory's effect on coding agent performance. Three conditions—persistent memory (Stompy), static file context, and no memory—were evaluated across three tasks of increasing complexity on a 4,895-line Python/FastAPI production codebase. The results challenge prevailing narratives in two directions.

First, persistent memory does not improve code quality. All three conditions achieved 84–96% scores across all tasks. The quality ceiling is a property of the model, not the memory system.

Second, persistent memory reduces exploration overhead—the turns, tokens, and time an agent spends rediscovering architecture it has already mapped. This effect scales with task complexity: negligible on simple tasks (where memory is pure overhead), but producing 28–40% fewer turns and 22–32% lower cost on complex cross-cutting tasks.

These findings matter because the AI memory industry benchmarks exclusively on conversational recall. No competitor has published controlled measurements of memory's effect on coding agent efficiency. The standard benchmark (LOCOMO) tests whether a system remembers dietary preferences across conversations—not whether it helps an agent navigate a production codebase. We argue this gap has allowed inflated claims to persist unchallenged, and propose a methodological framework for memory benchmarking in agentic coding contexts.

1 Introduction

The AI memory market has a measurement problem.

Mem0 claims 90% token savings [1]. A-Mem reports 85–93% reduction [2]. MemMachine advertises 80% [3]. Zep publishes 94.8% accuracy [4]. These numbers are technically correct and practically meaningless for anyone building coding agents.

Mem0's 90% measures memory footprint compression—26,000 tokens of conversation history condensed to 1,800 tokens of extracted facts. It does not measure whether a downstream task completes faster, cheaper, or better. A-Mem's 85–93% measures tokens consumed per memory *operation*—the cost of storing and retrieving, not the cost of the task the memory supports. MemMachine's 80% was measured on LOCOMO [7], a benchmark that tests whether a system can recall facts from long conversations ("What restaurant did I mention last Tuesday?"). Zep's

94.8% measures retrieval accuracy—whether the right memory was found, not whether finding it improved anything.

Every published benchmark tests conversational recall. None tests coding.

This matters because coding agents are the primary commercial use case for AI memory. Developers don't need AI to remember their dietary preferences across sessions. They need AI to remember that the codebase uses a provider pattern, that the payment service is idempotent, that the test fixtures live in `conftest.py`—and to use that knowledge to avoid re-exploring architecture it already mapped yesterday.

We asked a simple question: does persistent memory help an AI coding agent work more efficiently on a real production codebase? Not "can it remember facts"—can it *use* remembered facts to reduce wasted effort?

The answer is nuanced, which is exactly why nobody else has published it.

Persistent memory does not improve code quality. Given sufficient turns, all conditions—memory, static file, and no memory—converge on the same quality ceiling (84–96%). The model's capability, not its memory, determines correctness.

What memory does is reduce exploration overhead. On complex tasks requiring cross-cutting architectural knowledge, the memory-equipped agent needed 28–40% fewer turns and completed 22–32% faster than the memoryless agent. Memory compresses the discovery phase without improving the solution phase. Think of a senior developer who knows the codebase versus a competent contractor who reads everything each morning. Same code. Different cost.

This effect has a complexity threshold. On simple, well-scoped tasks, memory is pure overhead—the memoryless agent was fastest. Memory only pays for itself when there is sufficient architecture to remember. A prior phase on a toy codebase confirmed this: without complexity, memory lost.

The contributions of this paper are:

1. The first controlled benchmark of persistent memory's effect on coding agent task performance, measured on a production codebase.

2. Evidence that memory affects exploration efficiency, not solution quality—a distinction absent from current industry discourse.

3. Identification of a complexity threshold below which memory systems add overhead without benefit.

4. A critical analysis of existing memory benchmarks, demonstrating that published claims measure fundamentally different quantities than coding agent performance.

We publish our 15–28% because that's what we measured. We believe modest, honest numbers advance the field further than impressive, irrelevant ones.

2 Background and Related Work

2.1 The AI Memory Landscape

Persistent memory for LLM-based systems has become an active area of both research and commercial development. Systems differ in architecture—graph-based extraction (Mem0 [1]), Zettelkasten-inspired associative notes (A-Mem [2]), stateful agent frameworks (Letta [5]), temporal knowledge graphs (Zep [4]), and hierarchical compression (MemMachine [3])—but share a common evaluation pattern: all benchmark primarily on conversational recall tasks.

Table 1 summarizes published claims and what they actually measure.

Letta deserves separate mention for intellectual honesty. Their Terminal-Bench and Context-Bench evaluate agent capabilities rather than inflated token metrics. However, Letta has not

Table 1: Published AI Memory Claims vs. Measurement Methodology

System	Claimed Saving	Actually Measures	Benchmark
Mem0	90% tokens	Memory compression	LOCOMO
A-Mem	85–93% tokens	Per-operation cost	Dialogue QA
MemMachine	80% tokens	Recall accuracy	LOCOMO
Zep	94.8% accuracy	Retrieval precision	Custom
Letta	N/A (honest)	Agent capability	Terminal-Bench
This work	**15–28%**	**Task efficiency**	**Coding tasks**

published an A/B comparison of with-memory versus without-memory on coding tasks. Their most revealing contribution may be demonstrating that a plain filesystem achieves 74% on LOCOMO with GPT-4o-mini [6]—a result that questions whether LOCOMO discriminates between memory architectures at all.

2.2 The LOCOMO Problem

LOCOMO (Long Conversation Memory Benchmark) [7] is the de facto evaluation standard for AI memory systems. It tests whether systems can answer questions about facts mentioned across long conversational histories: names, preferences, events, relationships.

The benchmark has two structural limitations for coding applications. First, it tests *recall*—can the system retrieve a fact?—not *utility*—does retrieving that fact improve task performance? A system scoring perfectly on LOCOMO may add zero value to a coding agent if the recalled facts are never relevant to the task at hand.

Second, LOCOMO's discriminative power is questionable. When Letta demonstrated that simply writing conversation summaries to a text file achieves 74% with a mid-tier model [6], the benchmark's ability to differentiate sophisticated memory architectures from naive approaches came into doubt. If the baseline is 74%, the ceiling for "impressive" improvement is narrow.

Mem0's own ECAI 2025 paper [1] reveals further limitations: full-context approaches outperform Mem0 for conversations under 30 turns, and Mem0 shows 30–45% accuracy degradation on implicit information tasks between 30 and 150 turns. The paper's evaluation focuses on dietary preferences, travel histories, and relationship details—valid use cases, but categorically different from coding agent workflows.

2.3 Coding Agent Benchmarks

SWE-bench [8] evaluates coding agents on real GitHub issues but tests single-shot problem solving—each task is independent, with no prior context to remember. Memory systems have no theoretical advantage here, making SWE-bench a useful negative control but not a memory benchmark.

ToM-SWE [9] comes closest to a memory-aware coding benchmark, introducing "Stateful SWE-bench" with persistent user preferences across tasks (59.7% vs 18.1% without preference memory). However, it measures whether agents respect stated user preferences (coding style, library choices), not whether architectural memory reduces exploration overhead. The questions are different: "Does the agent remember I prefer tabs over spaces?" versus "Does the agent avoid re-mapping the service architecture?"

Terminal-Bench [10] tests tool-use capabilities in terminal environments. Relevant to agent competence, but includes no memory A/B comparison.

No published benchmark measures: *does persistent memory reduce the cost of multi-session coding work on a real codebase?*

2.4 Memory as Exploration Reduction

We propose a theoretical frame: persistent memory in coding agents functions as exploration reduction, not quality enhancement.

Consider the analogy of developer onboarding. A senior developer who has worked on a codebase for six months and a competent new hire will ultimately produce comparable code—given sufficient time. The senior's advantage is not superior coding ability but *compressed discovery*. They know the architecture, the patterns, the conventions, the gotchas. The new hire must discover all of this by reading code, running tests, and asking questions.

An AI coding agent without memory is permanently a new hire. Every session, it re-reads the codebase. Every task, it re-discovers the architecture. The code it eventually produces may be identical—the quality ceiling is set by model capability, not by familiarity—but the path to that code is longer, consuming more turns, more tokens, and more time.

Memory should compress the discovery phase without improving the solution phase. If this frame is correct, we should observe: (a) no quality difference between conditions, (b) efficiency differences that scale with how much discovery a task requires, and (c) no effect on tasks where discovery is minimal.

3 Methodology

3.1 System Under Test

Stompy is an MCP-based (Model Context Protocol) persistent memory system with a PostgreSQL backend. It stores versioned, tagged context snapshots with embedding-based retrieval (VoyageAI), priority levels, and conflict detection. When an AI agent connects via MCP, it can store contexts (`lock_context`), retrieve them (`recall_context`), and search across them (`context_search`).

For this benchmark, we are both the system developers and the experimenters. This is a limitation we address in Section 6. It is also an ecological strength: we tested memory on the Stompy backend codebase itself (dementia-production), where 1,200+ commits of development history informed the pre-loaded contexts. The contexts reflect genuine architectural knowledge accumulated through real development, not synthetic test fixtures.

3.2 Codebase

All tasks were performed on the Stompy backend (dementia-production), a production Python/FastAPI/Postgre application: 4,895 lines in the main server module, 158 Python source files in the core source tree, and 563 Python files total including tests, scripts, and tooling. The system implements the full MCP server with context management, ticketing, database queries, conflict detection, and project isolation.

This codebase was chosen because it is complex enough to have discoverable architecture (service patterns, database conventions, error handling approaches) but small enough for a single agent to navigate in one session. It represents the kind of real-world codebase where developers most plausibly deploy AI coding agents.

3.3 Experimental Design

We evaluated three conditions across three tasks. Each condition represents a different approach to providing the agent with prior knowledge about the codebase:

The **file** condition is a critical control. It contains the same information as the Stompy contexts, delivered as a static markdown file. This isolates the effect of the *knowledge content* from the *delivery mechanism*. If Stompy outperforms file, the MCP retrieval architecture adds value.

Table 2: Experimental Conditions

Condition	Description
stompy	12 pre-loaded architectural contexts available via MCP (patterns, conventions, architecture decisions)
file	Identical knowledge consolidated into a single `CONTEXT.md` file in the project root
nomemory	No prior knowledge; agent reads only source code

If file matches Stompy, the knowledge content is what matters, and any delivery mechanism suffices.

The file condition does not scale—even a small team working on the same codebase will struggle to maintain a single coherent context file as architecture evolves, conventions shift, and contributors disagree. The file cannot be selectively retrieved (the agent reads it entirely regardless of task relevance), cannot detect conflicts between documented and actual state, and provides no versioning or attribution. For this benchmark's scope, it is a fair and informative comparison.

3.4 Tasks

Three tasks of increasing complexity simulate the progression from routine modifications to cross-cutting architectural changes:

Table 3: Benchmark Tasks

Task	Complexity	Description	Simulated Session
1	Moderate	Add data count preview to project deletion tool	Session 20
2	High	Extract services from monolithic server file	Session 40
3	Very High	Implement rate limiting with Redis and tiered policies	Session 100

Task complexity was designed to increase the amount of architectural knowledge required. Task 1 modifies a single function with local scope. Task 2 requires understanding the service extraction patterns, import conventions, and test organization across the codebase. Task 3 demands knowledge of the authentication system, database patterns, Redis integration conventions, and the existing middleware architecture.

"Simulated Session" indicates the development maturity the task assumes—Task 3 represents work that would occur after extensive prior development, when accumulated knowledge matters most.

3.5 Scoring

Each task was scored on a 25-point rubric: five criteria, each rated 1–5.

1. **Functional correctness**: Does the implementation work as specified?
2. **Code integration**: Does it follow existing patterns, naming, and structure?

3. **Error handling**: Are edge cases and failure modes addressed?
4. **Test coverage**: Are tests present, passing, and meaningful?
5. **Architecture alignment**: Does the solution fit the codebase's design philosophy?

Scoring combined automated test scripts (functional correctness, test coverage) with structured manual review (integration, architecture alignment). The same rubric and reviewer applied to all nine runs.

3.6 Metrics

Beyond quality scores, we captured:

- **Turns**: Number of agent interaction cycles (proxy for exploration effort)
- **Cost**: Total API cost in USD (input + output tokens)
- **Wall time**: Seconds from task start to completion
- **Cost per point**: USD per quality point (efficiency metric)
- **Points per minute**: Quality output rate

3.7 Model and Controls

All runs used Claude Opus 4.6 (Anthropic). The same model version, codebase state, and scoring rubric applied to every run. Nine total runs were executed: three conditions $\times$ three tasks. Each run started from an identical codebase snapshot.

3.8 Phase 1 Context

A prior benchmark phase used a toy Express/TypeScript codebase (approximately 200 lines) with three sequential sessions of simple tasks. In that phase, the no-memory condition won outright: 70.3% versus 59.5% for the memory condition. We include this result because it establishes the complexity threshold—memory is not universally beneficial. On a trivial codebase, the overhead of memory retrieval exceeds its value. Phase 1's negative result informed the Phase 2 design: use a real production codebase where there is architecture worth remembering.

4 Results

4.1 Quality Results

Table 4 presents the complete results matrix.

The quality finding is unambiguous: all conditions achieve scores between 84% and 96%. No condition demonstrates a systematic quality advantage. The model's capability ceiling—not the memory system—determines code correctness.

The file condition achieved the highest aggregate score (68/75), edging out both stompy and nomemory (67/75 each). We report this without qualification. If the goal were maximum quality regardless of cost, a well-crafted context file wins. This result is consistent with our theoretical frame: memory affects efficiency, not quality.

Table 4: Complete Results: 3 Conditions × 3 Tasks

Task	Condition	Score	Time (s)	Cost ($)	Turns	Pts/Min
1 (Moderate)	stompy	23/25	368	1.78	31	3.75
	file	23/25	372	2.86	40	3.71
	nomemory	22/25	216	1.48	23	6.11
2 (High)	stompy	21/25	391	3.18	45	3.22
	file	21/25	412	3.39	50	3.06
	nomemory	21/25	438	4.08	63	2.88
3 (Very High)	stompy	23/25	207	1.79	37	6.67
	file	24/25	238	1.93	34	6.05
	nomemory	24/25	304	2.30	45	4.74

Table 5: Aggregate Performance by Condition

Condition	Score	Avg %	Cost ($)	Turns	$/Point
stompy	67/75	89.3	6.75	113	0.101
file	68/75	90.7	8.18	124	0.120
nomemory	67/75	89.3	7.86	131	0.117

4.2 Efficiency Results

Table 5 aggregates performance across all three tasks.

Stompy is the cheapest overall ($6.75 vs $7.86 and $8.18), uses the fewest turns (113 vs 131), and achieves the best cost-per-point ($0.101 vs $0.117 and $0.120). The efficiency advantage is 14% over nomemory and 16% over file in cost-per-point terms.

The file condition presents an instructive pattern: highest quality, highest cost. The static context file provides excellent knowledge but without selective retrieval—the agent processes the entire file regardless of relevance to the current task. Stompy's MCP-based retrieval loads only relevant contexts, reducing token overhead on tasks where only a subset of architectural knowledge applies.

4.3 The Complexity Gradient

The most significant finding is how memory's effect varies with task complexity.

Table 6: Stompy vs. No-Memory Efficiency by Task Complexity

	Task 1 (Mod.)	Task 2 (High)	Task 3 (V. High)
Turn reduction	−35%†	+28%	+18%
Cost reduction	−20%†	+22%	+22%
Time reduction	−70%†	+11%	+32%

† Negative = nomemory was more efficient

On Task 1 (moderate complexity), the no-memory condition was fastest, cheapest, and used the fewest turns. Memory retrieval added overhead that exceeded its value—there simply wasn't enough architectural complexity to justify the discovery cost.

On Task 2 (high complexity), the pattern reverses. The no-memory agent needed 40% more turns (63 vs 45) to complete the same task at the same quality. The additional turns represent exploration—reading files, mapping imports, understanding patterns—that the memory-equipped agent already knew.

On Task 3 (very high complexity), stompy completed 32% faster at 22% lower cost. The file condition scored one point higher (24 vs 23) but took 15% longer and cost 8% more.

4.4 Phase 1 versus Phase 2

Phase 1 (toy codebase, simple tasks): no-memory won, 70.3% vs 59.5%.

Phase 2 (production codebase, complex tasks): scores converged, but stompy was most cost-efficient.

The shift confirms the complexity threshold. Memory is not intrinsically valuable—it is conditionally valuable, and the condition is sufficient architectural complexity to amortize retrieval overhead. A codebase with 200 lines of Express.js has nothing worth remembering. A codebase with 4,895 lines of Python across 158 files, with service patterns, database conventions, and authentication middleware, has plenty.

5 Discussion

5.1 Why Quality Doesn't Improve

LLMs have a quality ceiling for a given model, task, and codebase. This ceiling is determined by the model's training, its reasoning capability, and the inherent difficulty of the task. Memory does not raise this ceiling.

Given sufficient turns, a capable model will explore the codebase, discover the relevant patterns, and produce correct code. Memory reduces the number of turns required to reach this point—it compresses the search—but the destination is the same. This is why all three conditions converge on 84–96%: the model is equally capable in all conditions, and the tasks are within its ability range.

This finding should be expected but apparently isn't. Memory system marketing implies that persistence improves output quality. Our data shows it improves output *economics*. These are different claims with different value propositions.

The practical implication is significant: memory's value should be evaluated in cost and efficiency terms, not quality terms. Asking "does memory make the agent write better code?" is the wrong question. The right question is "does memory make the agent write the same code cheaper?"

5.2 The Industry Measurement Problem

Why hasn't anyone published this kind of benchmark?

Wrong benchmarks. LOCOMO is tractable and produces impressive numbers. It is also categorically different from coding agent evaluation. Testing memory on conversational recall is like testing a car's engine by measuring its paint quality—technically related (both are car properties), practically irrelevant.

Wrong metrics. "Token savings" is reported as memory footprint compression (input representation) or per-operation cost (internal efficiency), not task-level efficiency (end-to-end cost). These metrics are meaningful for system design but misleading when marketed as user-facing benefits. A system that compresses 26,000 tokens to 1,800 tokens of stored memory has not demonstrated that any task completes 90% cheaper.

Incentive misalignment. A controlled benchmark on coding tasks might reveal modest gains—as ours did. Marketing departments prefer 90% to 15–28%. The absence of rigorous measurement is not accidental; it is economically motivated.

Methodological cost. Controlled coding benchmarks are expensive. Each run consumes real API tokens, requires careful codebase setup, and demands reproducible scoring. Nine runs

on our benchmark cost approximately $23 in API fees alone. A full factorial design with multiple models would cost hundreds. The economics favor cheap conversational benchmarks over expensive task-based ones.

We do not claim bad faith by competitors. We claim insufficient rigor in an immature field. The correction is straightforward: publish what you measure, distinguish between metrics, and test on tasks your users actually perform.

5.3 When Memory Hurts

Phase 1 and Task 1 of Phase 2 demonstrate that memory can degrade performance. The mechanism is straightforward: memory retrieval has cost. MCP calls consume tokens (the query, the returned context, the model's processing of that context). On simple tasks, this cost exceeds the savings from reduced exploration because there is minimal exploration to reduce.

Part of this overhead is specific to the integration path. MCP is the heaviest retrieval mechanism—the LLM decides what to retrieve, formulates queries, and processes returned contexts within its token budget. Lighter integrations (direct API calls, CLI pre-injection) can deliver the same knowledge with lower per-call overhead by moving retrieval logic outside the model's reasoning loop. The benchmark used MCP exclusively because it is the primary interface for chat-based coding agents, but the overhead profile would differ for API or CLI integration.

This suggests memory systems should be complexity-aware. Aggressive retrieval on every task is wasteful. An adaptive approach—retrieving more context for tasks that involve unfamiliar or cross-cutting concerns, less for well-scoped modifications—would better match the observed pattern. The complexity threshold is an empirical property of the codebase and task, not a fixed parameter of the memory system.

5.4 File Context and Scalability

The file condition deserves careful interpretation. It achieved the highest aggregate quality (68/75) while being more expensive than Stompy but cheaper than no-memory on complex tasks. A well-written context document is genuinely valuable.

However, the file condition does not scale. Even a small team will struggle to maintain a single coherent context file—architecture evolves, conventions shift, and two developers will document the same pattern differently. The file cannot be selectively retrieved (the agent processes it entirely regardless of task relevance), cannot detect conflicts between documented and actual architecture, and cannot version or attribute its contents.

The file condition establishes a useful baseline: this is how well *having the right knowledge* performs, independent of delivery mechanism. That Stompy nearly matches its quality at lower cost suggests the MCP retrieval mechanism adds efficiency value even when the knowledge content is equivalent.

6 Limitations

We report these limitations not as pro forma caveats but because the absence of such transparency is precisely what makes competitor claims unreliable.

Single model (N=1). All runs used Claude Opus 4.6. Different models may exhibit different sensitivity to memory-provided context. Models with larger context windows might extract more value from the file condition; smaller models might benefit more from selective retrieval.

Single codebase. The benchmark used one Python/FastAPI codebase. Results may differ for other languages (statically typed languages may require less architectural discovery), other frameworks, or other codebase sizes. We chose a production codebase for ecological validity at the cost of generalizability.

Our own system on our own codebase. We tested Stompy's memory on the Stompy backend itself. The pre-loaded contexts reflect genuine architectural knowledge from real development—they were not fabricated for the benchmark. But we cannot exclude the possibility that our familiarity with the codebase influenced context quality in ways that would not transfer to external users.

Nine runs, not ninety. With one run per cell (3 conditions $\times$ 3 tasks), we report observed differences, not statistically significant effects. LLM outputs are stochastic; repeating the same run might produce different scores. This is a pilot study establishing methodology and directional findings, not a definitive statistical claim.

No multi-model validation. Extending to Claude Sonnet, GPT-5-Codex, and Gemini 2.5 Pro would strengthen generalizability. Budget constraints limited this phase to a single model.

Pre-loaded contexts. The stompy condition used 12 hand-curated architectural contexts representing accumulated development knowledge. Automatically generated contexts (from conversation history, code analysis, or LLM summarization) might perform differently. The benchmark tests memory *utility*, not memory *generation*.

Scoring subjectivity. Despite automated test components, criteria like "architecture alignment" involve judgment. The same reviewer scored all runs to ensure consistency, but inter-rater reliability was not assessed.

These limitations bound our claims: persistent memory provides 15–28% efficiency gains on complex tasks in this specific configuration. Generalization requires replication across models, codebases, and independent evaluation.

7 Conclusion

Persistent AI memory does not make coding agents write better code. It makes them write the same code cheaper.

On a production Python/FastAPI codebase, memory-equipped agents achieved identical quality (84–96%) to memoryless agents while consuming 14% less total cost, 14% fewer turns, and completing complex tasks 22–32% faster. This efficiency advantage scales with task complexity and vanishes on simple tasks where exploration overhead is minimal.

The finding is modest. We think modesty, grounded in controlled measurement, is what this field needs. Every AI memory company publishes impressive numbers. None of them measure what coding developers actually care about: does this make my AI agent cheaper to run on real work? We measured it. The answer is 15–28%, conditional on complexity, and honest about its limitations.

The benchmark gap is the deeper contribution of this work. An industry benchmarking on conversational recall while marketing to coding use cases has created space for inflated claims. LOCOMO tests whether a system remembers dietary preferences. Our benchmark tests whether a system reduces the cost of implementing rate limiting on a production API. These are different questions, and conflating them does a disservice to practitioners evaluating memory systems for real-world deployment.

This benchmark is the first in a series of planned studies, each addressing a distinct question about AI memory in practice:

Multi-agent swarms. Where a single agent re-explores architecture across sessions, multiple agents re-explore across *each other*—compounding the redundancy that memory should eliminate. Early industry data suggests multi-agent systems consume 1.5–7$\times$ more tokens than necessary due to coordination overhead [11, 12]. We are benchmarking a six-agent swarm on a production codebase with and without shared persistent memory. If single-agent memory saves 15–28%, shared memory across coordinating agents may save substantially more by eliminating the N-agent duplication of architectural discovery. These results will be presented separately.

Multi-model validation. This benchmark used a single model (Claude Opus 4.6). We plan to extend across model families using open coding agent frameworks (OpenCode and similar), testing Claude Sonnet 4.5, GPT-5-Codex, and Gemini 2.5 Pro. Different models may exhibit different sensitivity to memory-provided context—particularly models with smaller context windows, where selective retrieval may provide proportionally greater benefit.

Serialization format efficiency (TOON vs JSON). Memory systems must serialize context for storage and retrieval. Stompy already supports both JSON and TOON (a token-optimized notation) for context representation. We are benchmarking whether format-level token efficiency compounds with memory-level efficiency—whether the same architectural knowledge, delivered in fewer tokens, shifts the complexity threshold at which memory becomes cost-effective.

Additionally, we plan LOCOMO evaluation to contextualize our system against competitor benchmarks on their own terms, and a longitudinal multi-session study tracking efficiency across 27 sessions on a sustained development task.

We built a memory system. We tested it honestly. The results were modest. Then we looked at what everyone else claims—and realized nobody tested theirs at all.

References

[1] T. Chhablani, P. Jain, D. Khashabi, et al., "Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory," *Proc. ECAI*, 2025. arXiv:2504.19413.

[2] Y. Li, Y. Ding, and Z. Li, "A-Mem: Agentic Memory for LLM Agents," *Proc. NeurIPS*, 2025. arXiv:2502.12110.

[3] MemMachine, "LLM Memory Benchmark Results," Technical Report, 2025. Evaluated using Mem0's LOCOMO implementation.

[4] Zep AI, "Deep Memory Retrieval: Evaluation of Temporal Knowledge Graphs for LLM Memory," Technical Report, 2024.

[5] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, "MemGPT: Towards LLMs as Operating Systems," *Proc. ICLR*, 2024. arXiv:2310.08560.

[6] Letta, "LOCOMO Benchmark Analysis: Filesystem Baseline Results," Blog Post, 2025. Demonstrated 74% LOCOMO accuracy with plain filesystem using GPT-4o-mini.

[7] A. Maharana, D. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fung, "Evaluating Very Long-Term Conversational Memory of LLM Agents," *Proc. ACL*, 2024. arXiv:2402.17753.

[8] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *Proc. ICLR*, 2024. arXiv:2310.06770.

[9] J. Liu, Z. Feng, W. U. Ahmad, H. Fang, A. Cazes, C. Chen, and N. Peng, "Theory-of-Mind-Aware SWE Agents," 2025.

[10] Letta, "Terminal-Bench: Evaluating Agent Tool Use in Terminal Environments," 2025.

[11] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, et al., "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," *Proc. ICLR*, 2024. arXiv:2308.00352. Reports 72% token duplication in multi-agent coordination.

[12] Galileo AI, "Hidden Costs of Multi-Agent LLM Systems," Technical Report, 2025. Documents 1.5–7× token overhead from redundant context sharing.

[13] T. Huynh, A. Budhkar, and M. Ruder, "Agents in the Real World: Challenges and Opportunities," O'Reilly, Feb. 2026. Identifies shared memory as infrastructure requirement for coherent multi-agent architectures.