

MemCommit: A Transactional Memory Layer for Agents

Lam Nguyen

Case Western Reserve University

Cleveland, Ohio, USA

ltn18@case.edu

Abstract

Agents that remember need two things at once: the full record of what happened, so detail is never lost, and short notes of what matters, so they can read what they know in a few lines. Most agent-memory systems keep one or the other in a store that has no notion of a transaction, no record of where a memory came from, and no way to take a bad one back. We present MemCommit, a memory layer that treats agent memory as a database problem. Memories are immutable and content-addressed; status and tier are a fold over an append-only log; every memory records what it was derived from, so when a premise goes bad everything built on it goes stale in one transaction. On this base we build episodic sessions, derived notes at paths with a generated index and five agent tools, an offline consolidation pass that extracts what agents did not think to save and can be previewed and undone, and a declared memory structure that a commit guard enforces. On a synthetic multi-session benchmark, search over the record answers 96% of questions from 20 words of context where the full history needs up to 1,358; the notes reach 100% with a perfect extractor and about 49% with our rule-based one, so extraction quality, not the pipeline, is the limit. We report where the benchmark favours us and where it does not.

Keywords

agent memory, LLM agents, provenance, transactions, log-structured storage, memory consolidation

1 Introduction

An agent that talks to a user for months, or a team of agents that work from each other’s output, accumulates far more than fits in a prompt. The usual answers are a vector store of past text, a set of notes the agent writes for itself, or both. Each leaves the same gaps. When a note was built on something that later turns out to be wrong (a tool that was compromised, a user who corrected themselves, a document that changed) nothing says which other notes and answers leaned on it. When two agents write at once, nothing orders them. When a consolidation step rewrites the notes badly, nothing takes it back. These are the problems a database solves, and agent memory has mostly not borrowed its answers.

MemCommit treats agent memory as a database problem. A memory is an immutable, content-addressed JSON value; everything that changes about it (that it was written, went stale, was retracted, or moved to another tier) is a record in an append-only log, and the state of the store is a fold over that log. Every memory names what it was *derived from*, so invalidation is a graph walk: when a premise stops holding, everything built on it goes stale, transitively, in one transaction. Rollback is compensation, as in write-ahead logging [7], so nothing is erased.

On that base MemCommit keeps two kinds of memory. *Episodic* memory is the full record of sessions, turns, and tool calls, appended as they happen. *Derived* memory is what was learned from it: short notes at paths, with a generated index an agent reads first, five tools it can call to save, update, delete, search, and fetch, and an offline *dreaming* pass that extracts what the agent did not think to save, supersedes conflicting notes, merges duplicates, and splits oversized ones. Because a note records the episodes it came from, a bad episode takes its notes with it, and a pass can be previewed and undone. A project declares its own memory structure once, as a schema of kinds, and a commit guard enforces it for every writer.

Contributions.

- A memory model that unifies a log, tiers, and provenance: status and tier are a fold over an append-only log, and invalidation and rollback follow from it (Section 3).
- Two tiers of memory built on that model, with an index, agent tools, an offline consolidation pass, and a declared structure, all expressed as transactions with provenance (Section 4).
- A deterministic benchmark that separates retrieval from reading and the extractor from the rest of the pipeline, with its limitations stated (Section 5).

The library is MIT-licensed, has no runtime dependencies, and comes with a manual and a log of the design decisions behind it.

2 Related Work

Agent memory. MemGPT pages information between a main context and external stores the agent manages [9]. Generative Agents keep a stream of observations and retrieve by recency, importance, and relevance, with reflections that summarize it [10]; Reflexion stores verbal self-reflections after each trial [12]. MemoryOS organizes short-, mid-, and long-term stores with heat-based promotion [4]. A-MEM links notes to each other and updates the links as notes arrive [14]. Mem0 extracts facts and applies add, update, delete, and no-op operations [1], and Zep keeps a temporal knowledge graph in which facts have validity intervals [11]. MemCommit takes their ideas as policies and operations on a different base: the recall score and the summaries-as-reflections of Generative Agents, the tiers and promotion of MemoryOS, the status operations of Mem0 as records in a log rather than updates in place, and the temporal view of Zep as `as_of(1sn)`. What these systems do not offer together is transactions, a log with compensating rollback, and provenance that invalidates downstream work.

Evaluating long-term memory. LoCoMo [5] and Long-MemEval [13] evaluate chat assistants on very long conversations. We do not use them here: our benchmark is small and synthetic, built to isolate retrieval and the extractor, and is not a substitute.

Database ideas. Write-ahead logging with compensating roll-back [7], log-structured merge trees and the compaction they motivate [8], and buffer replacement policies (2Q [3], ARC [6]) are the direct sources of MemCommit’s log, its two levels of compaction, and its short-term cache and eviction policies. Provenance as the basis for propagating changes follows the database view of lineage [2].

3 The Memory Model

Memories are values. A memory has a kind, a key, a body (any JSON value), the agent that wrote it, a scope that says who may read it, an importance, and `derived_from`. Its `id` is a hash of all of that. Writing the same memory twice names the same memory, and work that is redone and comes out the same restores what was there. A body cannot change without its `id` changing, so tampering shows on load.

State is a fold over a log. A record puts, restores, retracts, or marks stale one memory, or moves it between tiers, inside a transaction. Status (live, stale, retracted) says whether a memory holds; tier (short, mid, long, archive) says where it sits. Neither is stored on the memory. Replaying the log in order gives both, and `as_of(lsn)` is a replay that stops early. A transaction either commits all its records or none, and passes through guards before the commit and observers after it. Rollback to an LSN writes the records that return every memory to its status and tier there; the log still shows what happened and what undid it.

Provenance drives invalidation. `derived_from` holds memory `ids` and outside references such as a document version. Invalidating a reference marks stale every live memory reachable through the reference index, in one transaction. A reference can also name a *read*: a memory built on “no notes about this topic” derives from the query that returned nothing, and when a matching memory later goes live the work built on the empty read goes stale. A retraction stales everything built on the retracted memory, directly or through a read that included it.

A declared structure. The core names no domain. A project declares its kinds as a schema: a JSON-schema subset for each body, the scopes a kind may be written in, the kinds it must be derived from, and a default tier. A commit guard rejects a write that is out of line with its kind, for every writer in every process, and the schema is itself a memory in the log, so `as_of(lsn)` shows which structure held then. A kind that *requires* a premise (an answer derived from a question) declares provenance together with structure.

The cache and the database underneath. Short-term memory is a bounded cache in front of the log. A use counts as a hit, a warm hit, or a miss; a missed live memory is paged back in; eviction policies (FIFO, LRU, LFU, 2Q, ARC, a forgetting curve, a scored and a learned policy) are pluggable. Compaction works at two levels: memory compaction folds finished traces into long-term summaries and moves steps to the archive, and log compaction folds each memory’s history to the records that replay to the same state, as an LSM tree rewrites sorted runs. Snapshots, hash-chain seals, a retention policy, and a server that owns the log for agents in many processes complete it.

4 Two Tiers of Memory

Episodic memory. A session is a trace: a root and one memory per step, each committed on its own so that a crash keeps what was said. User and agent turns, tool calls, and tool results are steps; a wrapper records any function’s calls and results. A reply is derived from the memories it read, so retracting one of them stales the reply. Nothing is distilled away, so detail that a summary dropped can still be fetched.

Derived memory. Notes live at paths (user/alice/city) as memories keyed by path. An update writes a new version and marks the old one stale, and by default everything derived from it; a delete retracts. The *index* is a Markdown catalog of every note’s path, title, and first line, grouped by directory and sized to a budget, keeping the most important and recent notes when it must cut. It is generated on demand and meant to be injected at the start of a session. Five tools expose the operations to a model: `add_memory`, `update_memory`, `delete_memory`, `search_memory`, and `fetch_memory`. Search spans both tiers, with relevance from a BM25 index, recency, and importance; fetch returns a note in full, or a session as its transcript.

Dreaming. Tools only save what an agent thought to save. A dreaming pass reads the sessions that ended since the last pass, asks an extractor for the learnings in each, and writes them as notes derived from the session and the turns they came from. It then maintains the store: a learning whose path holds a different note replaces it (the old note goes stale, with what was built on it), near-duplicates in a directory are moved to the archive, and oversized notes are split into parts derived from the original. The extractor is a function from a session to learnings. We provide a rule-based one that needs no model, which finds stated facts and preferences and records a successful tool sequence as a playbook; a language model is a function with the same signature. A pass is a set of transactions with provenance, can be previewed with a dry run, records what it covered as a memory in the log so the next pass knows where to start, and can be undone by rollback unless another agent has written since.

Why transactions matter here. The same machinery serves all of it. A spoofed session, once retracted, stales the notes learned from it and the replies built on those notes. A fact that changes supersedes the old note and stales the plans built on it. A bad consolidation is one rollback away. A declared kind refuses a malformed or orphaned write before it lands.

5 Evaluation

We ask two questions. Does memory help an agent answer questions about earlier conversations, and what does each tier and each writer contribute (Section 5.1)? And how do the cache’s replacement policies compare on an agent-shaped workload (Section 5.2)? Both benchmarks are synthetic and deterministic, and need no language model. They show behaviour, not real-world accuracy.

Table 1: Accuracy by sessions per user, and words of context per question at 64 sessions. Starred rows use an oracle extractor (an upper bound).

Configuration	8	32	64	Words
none	0.000	0.000	0.000	0
full history	1.000	1.000	1.000	1,358
episodic search	0.958	0.958	0.958	20
notes from tools only	0.154	0.133	0.113	3
notes (rule extractor)	0.492	0.512	0.487	15
episodic + notes	0.942	0.971	0.958	35
notes (oracle)*	1.000	1.000	1.000	12
episodic + notes (oracle)*	1.000	1.000	1.000	33

5.1 Does memory help?

Workload. Twenty users chat with an assistant over 8, 32, or 64 sessions each. Facts come up in passing; one is a request to remember (“Remember that my locker code is 4412”); 40% of users move cities later; and the rest is chatter with distractors that look like facts (“I like long walks”, “My friend lives in Rome”). Half the statements are in the forms the rule-based extractor is built for (“my dog is named Rex”) and half are paraphrases it cannot parse (“Our dog goes by the name Rex”). Afterwards each user is asked for their current city, their dog’s name, their drink, and their locker code. We report means over three seeds.

Reader. A configuration builds a context for the question from its memory, and a recency-aware reader answers from it: the answer is the latest-mentioned of the values the user ever gave for the attribute, and it is right if that is the current one. This measures what memory retrieves and how clean it is, not how well a model reads.

Configurations. *Full history* gives the reader everything the user said. *Episodic* searches the record and returns the best three turns. *Notes* are what dreaming extracted with the rule-based extractor, searched and returned three at a time; *notes from tools only* are those the agent saved by tool call when asked to remember. *Episodic + notes* gives three items from each tier. The *oracle* variants replace the rule-based extractor with one that parses every phrasing the workload uses: an upper bound on extraction, which a good model-backed extractor would approach.

Results. Table 1 and Figure 1 show four things. (1) *The full history is not needed.* Search over the record reads about 20 words where the full history needs 196, 694, and 1,358 as it grows, and is right 96% of the time. (2) *Search alone fails on facts that change.* It is perfect on facts said once (dog, drink, locker) and right on the city 83% of the time, because a move often shares few words with the question and the old statement is retrieved instead. (3) *The notes fix that if extraction is good.* With a perfect extractor the notes are right every time from 12 words, since the later statement supersedes the old note and the reader sees one value. With the rule-based extractor, which parses about half the phrasings, they are right about half the time, and the two tiers together only match search alone. The extractor is the limit, not the pipeline. (4) *Tools alone remember what they are told to.* Saving by tool call catches the explicit

Table 2: Hit rate of working memory by capacity, and log records written at capacity 32.

Policy	16	32	64	Records
FIFO	0.506	0.551	0.602	1,112
LRU	0.642	0.744	0.828	930
LFU	0.354	0.460	0.606	1,198
2Q	0.381	0.481	0.606	1,178
ARC	0.551	0.663	0.763	1,006
Forgetting curve	0.648	0.771	0.867	904
Learned (logistic)	0.627	0.742	0.803	932

request, and only when phrased as the agent is built to recognize it, so accuracy is 0.11 to 0.15.

Threats to validity. The workload is synthetic and small. Our first version stated every fact in the phrasing the rule-based extractor is built for, and the notes scored 1.000; that was circular, since both were written together, and the paraphrases are the correction. The paraphrases were written to share a stem with the questions (“lived” for “live”), which favours lexical search; a paraphrase with no shared words would defeat it and need an embedding index, which this benchmark does not test. Finally, a recency-aware reader is not a language model.

5.2 Eviction policies

We replay a synthetic agent-shaped workload (60 tasks writing 6 trace steps each, then 472 skewed reuses of earlier memories) through the real store under each policy and measure how often the memory an agent needs is still in working memory. Table 2 shows hit rate at three capacities, and the log records the policy cost at capacity 32. Every miss is a page-in and an eviction, both log records, so cost follows quality: the forgetting curve has the best hit rate under the cap and also writes the fewest records, 19% fewer than FIFO. LFU and 2Q do poorly because this workload reuses fresh trace steps often and both evict memories used only once first. Which policy wins depends on the workload, which is why policies are pluggable and the benchmark can replay one recorded from real agents.

6 Discussion and Limitations

What the results do and do not show. Memory is cheap where the full history is not, and the two tiers answer different questions: the record holds detail and facts said once, the notes hold what is current. The case for the notes, however, rests on extraction, and our rule-based extractor finds only about half the facts in the benchmark. A language model behind the same signature is the largest open gain, and the benchmark’s oracle rows show what it could deliver. We have not built or measured one.

Retrieval. The index matches words, not meaning. Stemming and stopword removal fixed the most visible failures, but a paraphrase that shares no words with the query is not found. MemCommit accepts any embedding function, and testing it on a workload designed to defeat lexical search is future work.

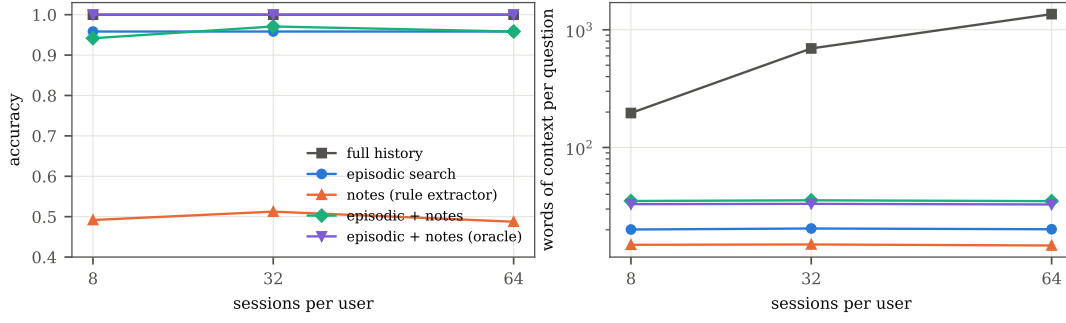


Figure 1: Accuracy and context size as each user’s history grows. The full history costs more with every session; search and notes read about the same amount at every length.

Scale and realism. Both benchmarks are synthetic, at small scale, and run in seconds. We have not evaluated on recorded conversations or with a model as the reader, and we make no claim about accuracy on public long-conversation benchmarks. The harness takes a workload and a retrieval function, so both are straightforward to add.

Design costs. A single log with a single writer per store keeps commits ordered and guards meaningful, at the price of a server when agents run in many processes. Cascading invalidation is the right default, but a frequently updated, heavily cited note stales a lot; a flag turns it off for minor edits. Conflicts resolve by supersession, not by judgement: the newest statement wins until someone retracts the session it came from.

7 Conclusion

Agent memory is a database problem, and a log with provenance gives it transactions, invalidation, and rollback without special cases. On that base, two tiers of memory, an index, tools, a declared structure, and an offline consolidation pass fit together because each is a transaction with a record of where it came from. Measured honestly on a synthetic benchmark, the record alone is a strong baseline, and the notes are as good as the extractor behind them.

References

- [1] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. *arXiv preprint arXiv:2504.19413* (2025).
- [2] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *ACM Symposium on Principles of Database Systems (PODS)*.
- [3] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *International Conference on Very Large Data Bases (VLDB)*.
- [4] Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. 2025. Memory OS of AI Agent. *arXiv preprint arXiv:2506.06326* (2025).
- [5] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating Very Long-Term Conversational Memory of LLM Agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.
- [6] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX Conference on File and Storage Technologies (FAST)*.
- [7] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems* 17, 1 (1992).
- [8] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996).
- [9] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560* (2023).
- [10] Joon Sung Park, Joseph C. O’Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *ACM Symposium on User Interface Software and Technology (UIST)*.
- [11] Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. 2025. Zep: A Temporal Knowledge Graph Architecture for Agent Memory. *arXiv preprint arXiv:2501.13956* (2025).
- [12] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [13] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. 2025. LongMemEval: Benchmarking Chat Assistants on Long-Term Interactive Memory. In *International Conference on Learning Representations (ICLR)*.
- [14] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-MEM: Agentic Memory for LLM Agents. *arXiv preprint arXiv:2502.12110* (2025).