GRAPHMEM: DURABLE, MULTI-AGENT GRAPH MEMORY FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A growing family of systems gives LLM agents long-term memory by letting an LLM curate a structured note graph over time, tagging, linking, and revising entries as experience arrives, rather than using a static vector index (Xu et al., 2025; Park et al., 2023; Gutiérrez et al., 2024; Rasmussen et al., 2025). These designs report large gains in small, validation-scale evaluations. Two questions remain open: whether this curation can run durably as first-class, auditable structure, and whether its benefits hold at real scale under statistical scrutiny. This study examines both questions with **graphmem**, a durable, concurrency-safe graph memory architecture built on RustFS (durable storage) and FalkorDB (a graph database with a native vector index), instantiating A-Mem's (Xu et al., 2025) Zettelkasten-style evolution loop while making every link a real, traversable edge that removes an id-hallucination failure mode present in A-Mem's own reference implementation. Using graphmem's reproducible harness, this paper reports a full-scale, statistically tested evaluation of this evolution paradigm across three long-conversation QA benchmarks (LoCoMo (Maharana et al., 2024), LongMemEval (Wu et al., 2025), BEAM (Tavakoli et al., 2026)). Memory beats no memory on nearly every well-powered comparison, but evolution's edge over a simpler retrieval-only ablation reaches significance on only one of three benchmarks, and a category-level claim about its cross-session value does not survive scaling up or a controlled reasoning-mode toggle. Running A-Mem's own code directly, not only its published numbers, yields the same pattern: its simplest configuration's apparent edge over graphmem is also not statistically significant. A separate, LLM-cost-free storage comparison isolates the architecture's own contribution: FalkorDB's throughput increases under concurrent load, where A-Mem's in-process store's throughput decreases under the identical load pattern.

1 INTRODUCTION

LLM agents that operate over many sessions need some way to retain information beyond a single context window. Concatenating the full interaction history into every prompt degrades as conversations grow long, both because models struggle to reason over very long contexts and because doing so is expensive at inference time. A large body of recent work instead equips agents with an external *memory system*: a store an agent writes salient information into and later retrieves from selectively (Xu et al., 2025; Chhikara et al., 2025).

Within that broader space, a specific and increasingly common design pattern lets an LLM curate the memory store's own structure over time, rather than treating it as a flat index. Generative Agents' memory stream and reflection (Park et al., 2023) established this pattern; A-Mem (Xu et al., 2025) sharpened it into a Zettelkasten-style workflow, in which an LLM decides, per new note, whether to link it to a retrieved neighbor or revise a neighbor's own tags and context, and knowledge-graph systems such as HippoRAG (Gutiérrez et al., 2024) and Zep (Rasmussen et al., 2025) independently arrived at representing memory as an explicit graph (see Zhang et al. (2024) for a broader survey). These designs report large gains from small, validation-scale evaluations, and their reference implementations typically keep the curated structure as transient, single-process state rather than a durable, auditable graph. This leaves two open questions this paper studies using A-Mem's own

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

evolution workflow as a concrete, reproducible test case: whether this curation can run durably, as structure owned by a system of record rather than an artifact confined to one process's memory, and whether its reported category-level benefits hold up at real scale, under statistical scrutiny, rather than on a handful of validation examples.

A-Mem's own reference implementation illustrates the first risk: memory lives in a single Python process, held as an in-memory dictionary with an embedded vector index (ChromaDB) alongside it, so the store cannot outlive the process or be shared between agents without external coordination, and links are unstructured strings inside a JSON blob rather than queryable relationships. This also has a concrete correctness consequence: its neighbor-linking step indexes into a candidate list using integer positions taken from a differently ordered list, so a link can be created to a memory the model was never shown. Section 3.3 describes graphmem's structural fix for this failure mode.

This paper studies both questions with **graphmem**, which instantiates A-Mem's evolution workflow, keeping every design decision about how an agent decides to link or revise memories unchanged, on a storage layer built for durability and concurrent, multi-agent access from the outset. This paper makes five contributions: (1) a dual-engine storage architecture separating durability from querying (Section 3); (2) a graph-native link representation carrying real graph-database ids end-to-end and filtering every suggested connection against the actual neighbor set shown, structurally ruling out the id-hallucination failure mode above (Section 3.3); (3) a stateless orchestration layer letting multiple agents share one store concurrently (Section 3.2); (4) pluggable, comparable embedding and LLM backends (Sections 3.4–3.4); and (5) the first full-scale evaluation of this evolution paradigm known to this study: three ablated configurations across three QA benchmarks with paired significance testing, a latency/throughput suite, and a controlled thinking-mode toggle (Section 4). This paper also reports a run of A-Mem's own code directly against the same benchmarks and graphmem's storage layer (Section 4.5, Section 5), reporting plainly where its edge over graphmem does and does not survive statistical testing, even as the storage design shows the concurrency advantage it was built for.

2 Related Work

Vector-store memory. The most common pattern, descending from retrieval-augmented generation (RAG) (Lewis et al., 2020), embeds each piece of information, stores it in a vector database, and retrieves the k nearest neighbors of a query. Simple and effective for basic recall, but every memory is an independent point in embedding space: the storage layer represents no relationship between memories, and the set returned for a query does not change as the store grows richer.

Non-graph agent memory systems. A separate line of work manages long context without vector retrieval as the primary mechanism: MemGPT (Packer et al., 2023) borrows OS-style virtual-memory paging; MemoryBank (Zhong et al., 2024) adds an Ebbinghaus-forgetting-curve-inspired decay rule; ReadAgent (Lee et al., 2024) compresses documents into "gist" episodes, looking up the original passage only when needed. None represents inter-memory relationships as explicit, queryable structure the way graph-based systems do, and their compression/paging/decay mechanisms confound a direct QA comparison with graphmem's evolution step on an unrelated design axis, so this paper discusses rather than benchmarks against them; graphmem's evolution mechanism (Section 3.3) targets *structure* (which memories relate to which) rather than compression or decay.

Graph-based memory. More recent systems incorporate a graph database directly. GraphRAG (Edge et al., 2024) builds a knowledge graph offline and summarizes graph communities for global, corpus-level questions a standard RAG baseline cannot answer. HippoRAG (Gutiérrez et al., 2024) builds its graph from one-shot entity/relation extraction and retrieves via Personalized PageRank, modeled on the hippocampal-indexing theory of human memory; unlike A-Mem-style evolution, this graph is fixed once extracted, not iteratively revised. Zep (Rasmussen et al., 2025) (built on the Graphiti engine) and Mem0 (Chhikara et al., 2025)'s graph-augmented variant both dynamically extract and link facts from ingested content; Zep is graphmem's closest architectural neighbor among deployed systems (both durable, graph-database-backed services), though its graph is populated by extraction, not an explicit link/revise loop over existing notes. Both report results on one of this paper's three benchmarks in their own papers (Zep on LongMemEval, Mem0 on

LoCoMo; Section 4.5). Systems in this family generally fix their graph's schema at extraction time, which the A-Mem authors note limits adaptability to novel information categories, the gap A-Mem's iterative evolution loop targets and the one graphmem inherits and makes durable.

A-Mem and Zettelkasten-style evolution. A-Mem (Xu et al., 2025) lets memory structure emerge from content itself through iterative, per-note LLM decisions, following the Zettelkasten method (Ahrens, 2017) and extending Generative Agents' (Park et al., 2023) earlier memory-stream-plus-reflection pattern into an explicit link/revise loop: the evolution policy graphmem instantiates directly (Section 3.3). graphmem does not propose a new evolution policy; it asks what this style of LLM-mediated graph curation looks like once durable, queryable, and shared safely across processes (Sections 3–3.2), then evaluates whether its reported benefits hold up at real scale (Section 4).

3 System Architecture

3.1 Design goals and the dual-engine backend

This work is organized around four goals. *Durability*: a memory, once written, must survive the process that wrote it. *Structural correctness*: a link between two memories must be a relationship the storage layer actually knows about, not a string that happens to look like an identifier. *Shareability*: two agents, or two concurrent requests to the same agent, must use one memory store without external locking or a rebuild step. *Comparability*: swapping the embedding model or evolution LLM should be a configuration change, not a code change, and must not corrupt or collide with data written under a different backend.

Figure 1 shows the resulting structure. Application code talks only to graphmem's central coordinator, which composes an embedding model, a durable object store, a graph and vector index, and an LLM interface. RustFS is an S3-compatible object store; every memory is a single JSON record, and RustFS is the only place a read is ever authoritative, independent of whether the graph index happens to be caught up. FalkorDB is a graph database with a native vector index, holding one node per memory plus the links the evolution step creates; it is treated purely as a rebuildable query and search index, never the system of record. This is a deliberate split: durability and queryability are different concerns with different failure modes, and coupling them, as an in-process dictionary plus an embedded vector store does, means a crash loses both at once. In graphmem, a memory is always written durably before the graph index reflects it at all, so a crash partway through adding a memory can only leave the graph index briefly behind the durable store, never ahead of it with an entry the durable store never recorded.

3.2 Graph schema, write path, and multi-agent statelessness

Every memory is stored as one graph node carrying its content, a short contextual summary, a category, tags, a timestamp, and an embedding vector; a vector index over these embeddings is built once per graph, using cosine similarity,

$$\mathrm{sim}(u, v) \;=\; \frac{u \cdot v}{\|u\|\,\|v\|}, \tag{1}$$

as the distance measure. Writing a memory is idempotent: re-adding a memory under the same identifier updates its node in place rather than creating a duplicate. A link created by evolution is a real, directed edge between two memory nodes, traversable like any relationship in the graph, not a string that merely looks like one. Because two embedding backends produce vectors that are not comparable to each other, each backend is isolated in its own graph; RustFS still holds exactly one canonical copy of each memory's content, so a corpus can be re-embedded under a second backend and compared side by side without either backend's vectors colliding with the other's.

Adding a memory embeds the content, then, unless evolution is disabled, runs the evolution procedure (Section 3.3) before durably recording the memory and updating the graph index; both writes always happen regardless of what evolution decided. Disabling evolution reproduces graphmem's storage and retrieval mechanics with the evolution step removed, A-Mem's own "w/o ME" ablation, reused below as the No Evolution configuration (Section 4.3). Unlike A-Mem's design, which keeps every memory in an in-process dictionary, graphmem's coordinator keeps no memory state inside

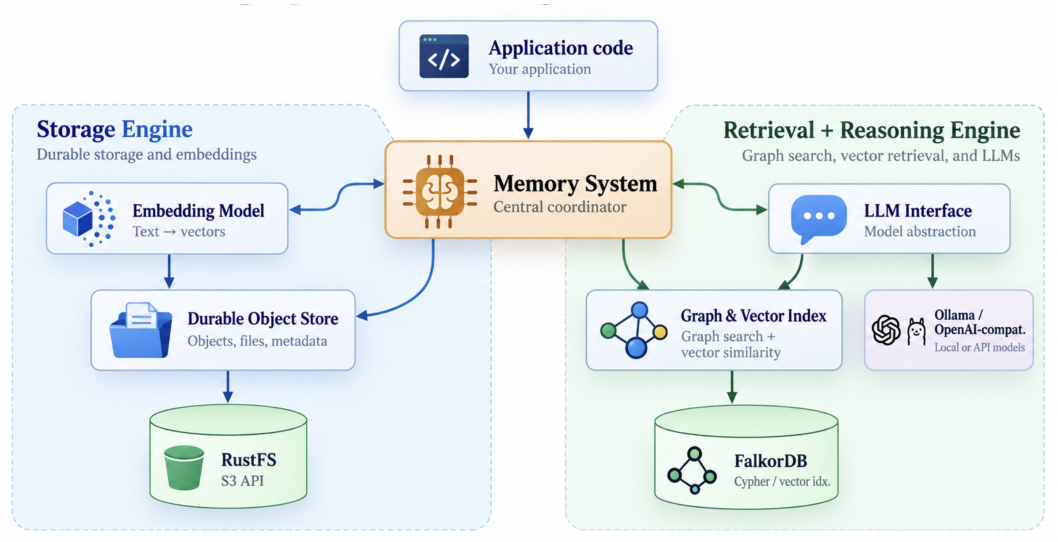


Figure 1: graphmem's dual-engine architecture. A central coordinator composes an embedding model, a durable object store (RustFS), a graph and vector index (FalkorDB), and an LLM interface. RustFS is the sole source of truth; FalkorDB is a rebuildable query index.

the process itself, so two agents can add memories to the same graph concurrently, each observing a consistent result, because RustFS and FalkorDB, not the process, own that state (Section 4.5 measures the consequence of this directly).

3.3 Memory Evolution and Eliminating Hallucinated Edges

graphmem's evolution write path. Every candidate the LLM can propose comes from the retrieval step's own result set $N_k(x)$; a candidate outside that set is dropped before it can become a link, which rules out the id-hallucination failure mode described in Section 1.

Adding a memory retrieves $N_k(x)$, the $k{=}5$ existing memories most similar to the new memory x under Equation 1 (empty if the graph holds no memories yet), then asks an LLM, via a schema-constrained completion, whether x should evolve and, if so, whether to link it to one or more members of $N_k(x)$, or instead revise a member's own summary and tags. Before any link is created, every proposed connection c is checked against the exact set the model was shown (Figure 3.3):

$$\text{a link to } c \text{ is created only if } c \in N_k(x). \tag{2}$$

A proposed connection outside that set is dropped rather than realized as a link. A-Mem's reference implementation has no equivalent check: it enumerates its retrieved neighbors as positions $0, 1, \ldots, k{-}1$, then reuses those position numbers to index into a separately ordered list, so a link can be created to a memory the model was never shown. Because graphmem tracks each neighbor by its own identity rather than by list position, no step can confuse a neighbor's position with its identity, and the membership check in Equation 2 rules out a link to anything outside $N_k(x)$. This is a structural guarantee about where a mistaken decision can land, not a claim about decision quality (Section 5): a poor decision can still link two memories that were both genuinely shown to the model, but it cannot manufacture a connection to one it never saw. Measured directly over 1,240 proposed connections across 788 real LoCoMo conversation turns (Appendix M), 2 (0.2%) fell outside $N_k(x)$ and would have become exactly this kind of hallucinated link under A-Mem's original position-based indexing; Equation 2 rules out both.

3.4 Pluggable Embedding and LLM Backends

The graphmem project supports two embedding backends, each isolated in its own graph (Section 3.2): Snowflake Arctic Embed (`snowflake-arctic-embed-l-v2.0`) (Yu et al., 2024), text-only, and Qwen3-VL-Embedding (Qwen Team, 2026), multimodal, mapping text, image,

document-image, and video inputs into one shared space. Content analysis and evolution decisions are driven by a common LLM interface with two backends: a local Ollama server by default (graphmem is only ever a client, never provisioning one itself), and an OpenAI-compatible backend that works against OpenAI's cloud API or any compatible local server (e.g. LM Studio). Swapping the underlying model is a configuration change, not a code change.

4 Evaluation Methodology

The graphmem evaluation separates how fast it is from whether its memory helps, with a distinct harness for each (Appendix A details both). Section 4.1 states graphmem's asymptotic costs and Section 4.2 its measured latency and throughput, both properties of the architecture itself. Section 4.3, the answer-quality evaluation, depends on that architecture directly: scoring many independent (variant, sample) graphs with paired significance testing needs exactly the isolation and crash-safety Section 3 establishes.

4.1 Algorithmic complexity

Let N be the number of memories in a graph and $k{=}5$ the fixed evolution neighbor budget. RustFS's per-key operations (storing, reading, and deleting a memory's content) are $O(1)$ in N. FalkorDB's vector index implements Hierarchical Navigable Small World graphs (Malkov & Yashunin, 2020), giving expected $O(\log N)$ nearest-neighbor query and insertion rather than the $O(N)$ a brute-force scan costs; writing a memory (an $O(1)$ identifier-keyed update plus $O(\log N)$ index insertion) and the evolution procedure's own neighbor retrieval both inherit this bound, and since k is fixed, evolution's prompt size and the membership check in Equation 2 do not grow with N. One operation has no dependence on N at all, and dominates wall-clock cost once evolution is enabled: the LLM call the evolution procedure issues, whose cost is set by the serving backend and token counts, not by graph size. Summing these terms, adding a memory costs

$$T_{\text{add}}(N) \;=\; \underbrace{O(1)}_{\text{durable write}} \;+\; \underbrace{O(\log N)}_{\text{index update}} \;+\; \underbrace{O(1)}_{\text{LLM call}} \;=\; O(\log N), \tag{3}$$

even though its measured latency with evolution enabled exceeds surrounding graph operations by three to four orders of magnitude at every N tested (Section 5): a favorable asymptotic curve with a very large constant factor, so graph size is not what determines graphmem's real end-to-end latency. Full derivation in Appendix B.

4.2 Latency and throughput

A standalone benchmark suite (Appendix A) measures the storage and retrieval layers directly, seeding without the LLM-bound evolution call so it does not dominate storage-layer measurements. Nearest-neighbor retrieval's mean latency stays within 2.01–2.09ms as corpus size grows 20× ($N{=}100{\to}2000$), consistent with $O(\log N)$ growth (at this range $\log_2 N$ itself grows only 1.65×, small enough to sit inside measurement noise); writing a memory shows the more visible instance of the same bound (1.96ms→3.08ms). RustFS's store/read/delete operations run 3.8–4.4ms at $c{=}1$, rising to 7.5–13.6ms at $c{=}8$ concurrent threads with 568–1006 ops/s; comparing two embedding backends, Snowflake is roughly 2× Qwen3-VL-Embedding's throughput but uses more memory at load time (3.35GB vs. 2.40GB), so latency and memory footprint do not trade off in the same direction. Full tables, the payload-size and concurrency sweeps, and the embedding-backend comparison are in Appendix C.

4.3 Answer-quality evaluation

To measure whether graphmem's evolution machinery improves an agent's ability to answer questions about a conversation it has seen before, this study evaluates on three long-conversation QA benchmarks of increasing scale and difficulty: **LoCoMo** (Maharana et al., 2024) (A-Mem's own benchmark), **LongMemEval** (Wu et al., 2025) (an independent multi-session haystack per question), and **BEAM** (Tavakoli et al., 2026) (100K–10M-token conversations, ten task categories). Three agents are seeded with identical turns and asked the same questions, varying only retrieval:

Variant	LoCoMo (n=1986)	BEAM	LongMemEval (n=58)
With Evolution	0.3380	0.2815 (n=272)	0.3573
No Evolution	0.3766	0.2876 (n=1600)	0.4003
No Memory	0.1947	0.2433 (n=1600)	0.1905
A-Mem (Full)	0.3558	**0.2925** (n=272)	0.2535
A-Mem (Storage)	**0.3918**	0.2704 (n=272)	**0.4123**

Table 1: Overall F1 by variant, all three benchmarks (Section 4.3, Section 4.5). With Evolution's and both A-Mem variants' BEAM n is scoped to 17 conversations by With Evolution's own evolution cost; No Evolution and No Memory cover all 100. Bold marks each column's highest point estimate only; none of the three is a statistically significant edge over the column's next-best entry (Appendix E). Per-category detail: Appendix F.

With Evolution (evolution on, retrieval expanded to include linked neighbors); No Evolution (evolution disabled, retrieval limited to the nearest neighbors themselves, otherwise identical mechanics); and No Memory (the full transcript dumped into every prompt, matching A-Mem's own baseline). Each (variant, sample) pair seeds an isolated FalkorDB graph, dropped once scored. Metrics are ported from A-Mem's own evaluation utilities (exact match, F1, ROUGE, BLEU, METEOR, SBERT (Reimers & Gurevych, 2019) cosine similarity); this paper reports F1 as the headline metric, full set and per-category breakdowns in Appendix F.

None of the three benchmarks is covered at literal 100%, though all are at full or near-full scale (exact coverage and cost projections in Appendix D): LoCoMo is complete (all 10 samples, 5 categories, 1,986 questions, at both thinking settings, Section 4.4); LongMemEval is a stratified full-scale sample (10 questions per type across all 6 types, 58 questions, since each question's independent haystack means seeding cost is not amortized across questions); BEAM's No Evolution/No Memory cover all 100 released conversations at all 4 scales (1,600 questions each), while With Evolution is scoped to 17 conversations (272 questions).

Main results. Table 1 reports overall F1 by variant and benchmark; Appendix E reports a paired bootstrap 95% CI and permutation-test p-value for every comparison below (10,000 resamples each, from per-question F1). Both graphmem variants beat No Memory significantly on LoCoMo and LongMemEval, and No Evolution does on BEAM (n=1600, largest p=0.0009); the exception is With Evolution − No Memory on With Evolution's own smaller BEAM sample (n=272), where a positive estimate (+0.0180) is not significant (p=0.253), consistent with the smaller sample size rather than an absence of effect.

No Evolution leads With Evolution in every point estimate (LoCoMo 0.3766 vs. 0.3380; LongMemEval 0.4003 vs. 0.3573; BEAM 0.2876 vs. 0.2815), but this edge is significant on only one of three benchmarks: LoCoMo, at the largest scale tested (p<0.0001). The same-shaped LongMemEval gap is not significant at n=58 (p=0.305). On BEAM's only valid paired subset (With Evolution's own 272 questions; Section 4.3's asymmetric coverage means With Evolution/No Evolution share no larger question set), the difference is small and not significant (p=0.611). LoCoMo is the only comparison in this paper's own ablation that reaches significance.

Category-level results are reported with the same caveat: per-category n is too small at several benchmark/category cells for the bootstrap to be informative (Appendix E). An earlier, 2-chat validation-scale BEAM run found the cross-session-reasoning category (cross-session-synthesis questions, where evolution's linking was expected to help most) to be With Evolution's clearest category win. At the scale reported here that reverses: No Evolution leads instead (0.3393 vs. 0.2720; counts are unmatched, 34 observations from 17 conversations vs. 200 from 100, so this is not evidence evolution fails here, only that the validation-scale estimate did not hold at full scale). LongMemEval's independent multi-session result, at full question-type scale, shows the same pattern on an unrelated benchmark and question design (Appendix F), though neither category result carries its own significance test.

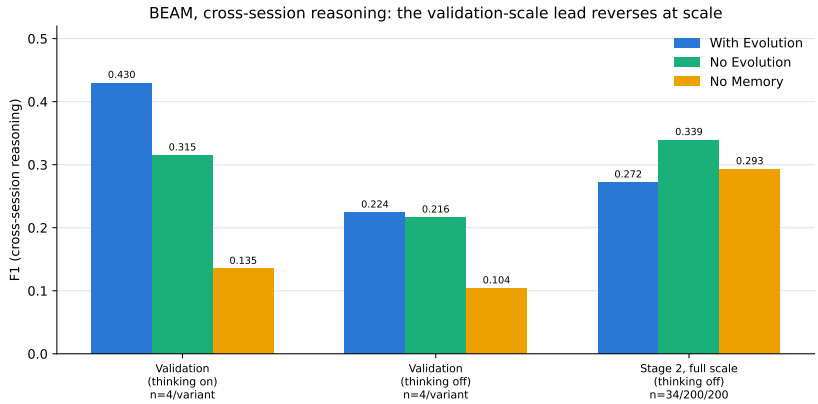


Figure 2: The cross-session-reasoning category's F1, across the three conditions this paper measured it under. With Evolution's clear lead at validation scale shrinks once thinking is turned off, then flips into a loss once the sample grows from 2 conversations to full scale.

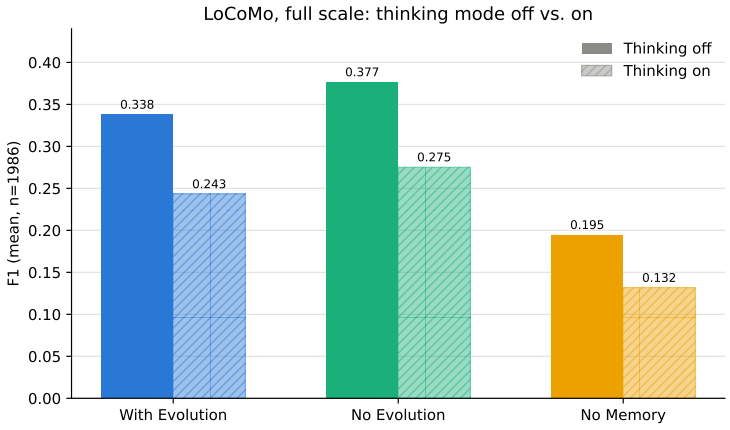


Figure 3: LoCoMo's full-scale matched pair: thinking mode lowers F1 for every variant, not just one, with the ranking between variants unchanged either way.

4.4 Effect of disabling the model's thinking mode

All results above run against `qwen3:latest` (emits an extended `<think>` block by default) via Ollama; a toggle for its native thinking parameter was added and LongMemEval/BEAM were re-run at validation scale and LoCoMo at full scale with thinking off. Seeding time (With Evolution's dominant cost) dropped 72–79% on LongMemEval/BEAM; LoCoMo's full-scale matched pair, the only one run at true full scale on both sides, showed an 8.5× total wall-time increase with thinking on and a uniform F1 drop across every variant and category. Quality moved both ways: thinking off eliminated 5 of 96 scored BEAM predictions containing raw, unparsed `<think>` output, but also shrank or erased With Evolution's validation-scale lead on the cross-session-reasoning category (0.4583→0.2011 at BEAM-10M-truncated, landing exactly at No Evolution's score), a second, independent axis (after Section 4.3's scale-up) suggesting part of that category's apparent advantage came from the model's own deliberation, not graph evolution's linking, though this paper does not isolate which specific call, evolution's decision or the final answer generation, drives the effect. All thinking-off runs use a fixed generation-length cap and request timeout on the evolution call, since disabling thinking removes the model's own natural stopping point (full timing table and caveats in Appendix G).

4.5 Comparison against A-Mem's own reference implementation

Every result above compares graphmem's own ablations against each other; none runs A-Mem's actual code. Two further variants are added (vendored from A-Mem's paper-reproduction repository, commit `0c8039f`), reported in Table 1: A-Mem (Full) (A-Mem's own storage and retrieval exactly as its evaluation script invokes it) and A-Mem (Storage) (the same storage, one direct embedding-similarity search, evolution skipped). This comparison does not show graphmem's evolution mechanism winning outright, though the point estimates overstate how equivocal it is under significance testing: A-Mem (Storage), A-Mem's cheapest, simplest configuration, has a higher point-estimate F1 than every graphmem variant on LoCoMo (+0.0152) and LongMemEval (+0.0120), and is within noise of With Evolution on BEAM, but neither apparent win is statistically significant (p=0.089/0.575; Appendix E), so this comparison shows only that neither configuration reaches a confident edge over the other at these sample sizes. A-Mem (Full) is sharper: its collapse on LongMemEval (0.2535 vs. A-Mem (Storage)'s 0.4123) against graphmem's With Evolution reaches significance at a nominal threshold (p=0.035), though not once a multiple-comparisons correction is applied across this paper's full set of headline tests (Appendix E); descriptively, the drop is concentrated in two single-recent-fact categories, consistent with query-keyword generation drifting from the question's literal terms where direct embedding search would not, though this paper offers no direct evidence isolating that mechanism. BEAM is the one dataset where evolution-style machinery holds its own in point estimate (A-Mem (Full) 0.2925 vs. With Evolution's 0.2815), consistent with its more synthesis-heavy design, though not significant at n=272 (p=0.440). Both A-Mem variants use its own default embedding model rather than graphmem's, a confound this comparison does not control for (Section 5). Full detail in Appendix H.

External reference points. Mem0 and Zep each report their own published results on one of this paper's three benchmarks, on a different backend and metric than this harness, so these numbers are context rather than a controlled comparison. Both systems were also run directly at pilot scale on this harness's own backend (Appendix I): a Mem0 result on one LoCoMo sample (F1 0.4188, single-hop) and a per-turn ingestion-cost measurement for Graphiti, discussed in Section 5.

5 Discussion

Cost of durability. Every graphmem write crosses a network boundary at least once (RustFS) and often twice (with FalkorDB), plus one LLM round trip per insert if evolution is enabled, strictly more expensive than an in-process dictionary. A raw memory write costs 2–3ms (Section 4.2), while one evolution round trip costs 24–39s thinking on, 6–7s off: three to four orders of magnitude more, set almost entirely by the LLM backend. Whether that cost is acceptable depends on deployment: a single-agent prototype has little to gain from durability it does not need, while a multi-agent deployment cannot function correctly with an in-process store at all. Both the ordering guarantee and the multi-agent sharing goal above were verified directly against the live services, not only argued from the code or a throughput number: 60 fault-injection trials interrupted execution at the two points the ordering guarantee depends on (immediately after the first RustFS write, and immediately after evolution's post-decision re-write), all 60 holding the invariant (Appendix K); separately, two agents on independent Ollama instances concurrently added 20 evolution-enabled memories to one shared graph and bucket, all durably present afterward with no phantom nodes or corrupted links (Appendix L).

Cross-system cost comparison. Mem0 and Graphiti (Section 2) were also run at pilot scale on this same local backend (Appendix I): Mem0's per-turn cost (11.6s/turn) matches graphmem's own evolution step; Graphiti's (33s/turn) is 2.8× more expensive, consistent with its five-LLM-call pipeline against every other system's one call per turn. Reading both systems' own source directly (Appendix B) confirms this is a constant-factor gap, not a difference in how any of the three systems scale with corpus size: both retrieve candidates via the same bounded ANN search before an LLM decision that graphmem's own evolution step uses, so neither breaks the $O(\log N)$ bound Section 4.1 establishes for graphmem itself.

Storage-layer concurrency comparison. Table 2 compares three stores at matched corpus size (N=2000) and concurrency, isolated from LLM cost: A-Mem's actual storage (in-process dict plus

brute-force cosine scan), graphmem's own FalkorDB, and Qdrant, a durable, client-server vector store with no graph layer, added to isolate whether FalkorDB's behavior comes from being client-server at all or from anything graph-specific. Single-threaded, A-Mem's scan is faster than either client-server store (a gap that would not reverse before $N \approx$5,000–6,000), but only FalkorDB and A-Mem diverge sharply under concurrent load, consistent with the multi-agent sharing goal in Section 3.1: a single in-process object accessed by multiple threads does not parallelize the way a client-server database does. Qdrant's own concurrent gain is real but markedly smaller than FalkorDB's, so being client-server alone does not explain FalkorDB's full advantage (Appendix J).

Store	**Single-thread**	**ops/s** (c=1)	**ops/s** (c=8)
A-Mem (in-process)	0.12–0.73ms	419	200–314
FalkorDB	∼2.0ms	500	1,000–1,060
Qdrant (non-graph)	∼2.0ms	515	591

Table 2: Search-operation latency/throughput, matched corpus size (N=2000) and concurrency. A-Mem is fastest single-threaded but the only store that does not scale with concurrency; Qdrant scales, but far less than FalkorDB.

Limitations. Every comparison carries a paired bootstrap CI and permutation-test p-value (Appendix E), quantifying sampling uncertainty over question selection at a fixed decoding setting, not generation variance: every LLM call uses the same fixed, non-zero temperature (0.7), so variants are not confounded by a decoding difference, but no result is repeated across seeds to measure how much a fresh run would move a given F1 or p-value. Per-category claims (single-digit n at several cells) remain untested point estimates. Table 7's 15 tests, corrected for multiple comparisons (Appendix E), leave only the No Memory comparisons and LoCoMo's No Evolution − With Evolution result significant, so the two weaker A-Mem (Full) findings in Section 4.5 are point estimates, not established effects. The A-Mem comparisons use its own default embedding model, an uncontrolled confound. The generation-length cap/request timeout that make thinking-off feasible (Section 4.4) are calibrated for Ollama, untested against OpenAI-compatible backends. Evolution's link decisions depend on each note's derived semantic metadata (Section 3.3); a dedicated rerun with that metadata found no significant F1 change on any benchmark (Appendix N). RustFS and FalkorDB are single instances, not a clustered deployment. Mem0/Zep's published numbers remain external context on a different backend/metric; both ran at pilot scale on a smaller local model (`qwen2.5:7b`) than every other result here (`qwen3:latest`), an uncontrolled confound (Appendix I); only Mem0's pilot completed, and Graphiti's measured per-turn cost makes a full-scale run a costed deferral.

Future work. Every evaluation above uses only the text modality of each benchmark's conversation data, though graphmem's embedding layer already supports a multimodal backend (Qwen3-VL-Embedding, Section 3.4, benchmarked for latency/throughput in Appendix C but not exercised end-to-end here). Whether evolution-driven linking behaves differently over multimodal memory, images, document scans, or video frames sharing an embedding space with text, remains untested. A second open direction is evolution design itself: this paper evaluates A-Mem's specific Zettelkasten-style workflow (Section 2) as one reproducible instance of LLM-mediated memory curation, not the only one possible. Comparing alternative evolution policies, note-taking styles, and prompts within graphmem's own durable storage is a direct next step toward improving retrieval quality over long-standing context. A third is decoding configuration: repeating each evaluation across seeds at the fixed temperature this paper uses, or sweeping temperature directly, would separate genuine generation variance from the differences attributed to evolution.

6 Conclusion

A growing family of systems now lets an LLM curate its own agent memory's structure rather than treating memory as a flat, static index. This paper asked whether that curation can be made durable and correct as first-class, auditable structure, and whether its reported benefits hold up at real scale under statistical scrutiny. **graphmem** answers the first question with a dual-engine backend separating a durable source of truth from a rebuildable query index, links that cannot be corrupted with references to memories a model never saw, and direct empirical verification, not just architectural

argument, that durability and multi-agent sharing both hold under real fault injection and concurrent load. The second question's answer is more nuanced than a clean win or failure for LLM-mediated evolution: memory beats no memory wherever the comparison is well-powered, but evolution's edge over a simpler ablation reaches significance on only one of three benchmarks, its cross-session category-level advantage does not survive scaling up or a thinking-mode toggle, and A-Mem's own code shows the same pattern, a higher point estimate that is not a significant edge. graphmem's durable, isolable storage is what made a full-scale, statistically tested evaluation of that question possible at all, independent of which evolution heuristic wins on a given benchmark.

AI USE STATEMENT

Generative AI tools (Claude, Anthropic) assisted with portions of this work, under author direction and review throughout. Assistance included drafting and editing sections of this paper's prose, condensing an earlier draft into the present ICLR-formatted version, and implementing, refactoring, testing, and documenting specific components of the graphmem codebase against author-specified designs and requirements. The system architecture, experimental design, and evaluation methodology are the authors' own. Generative AI tools were not used to generate synthetic experimental data, to fabricate results, or to write text presented as a human evaluator's judgment. All AI-assisted contributions, including code, prose, and reported claims, were reviewed for correctness, and responsibility for the final content of this work is retained by the authors.

ETHICS STATEMENT

This work uses three publicly released long-conversation QA benchmarks (LoCoMo, LongMemEval, BEAM), each already vetted and released by its own authors for this purpose; no new human-subjects data collection is introduced. graphmem's default LLM backend runs locally (Ollama), so no conversation content from these benchmarks was sent to a third-party cloud API during the evaluations reported here unless the optional OpenAI-compatible backend was explicitly configured. No dual-use or safety concern specific to this work is known: graphmem is a storage and retrieval architecture, not a new capability for generating harmful content, and its evaluation is restricted to answering questions about benchmark conversations the agent itself was seeded with.

REPRODUCIBILITY STATEMENT

graphmem is implemented as an open-source Python package with a bundled `docker-compose.yml` provisioning RustFS and FalkorDB; the LLM backend is configured, not bundled, and every result in this paper states the specific model and thinking-mode setting used (Section 4.4, Appendix G). Section 4 and its appendices state exact benchmark coverage (sample/question counts, categories, scales) for every reported result, and Section 4.1/Appendix B give the full derivation behind this paper's asymptotic claims. Metric implementations are ported unmodified from A-Mem's own published evaluation utilities. The evolution and content-analysis prompts, the exact FalkorDB Cypher used for indexing/upsert/link operations, and the benchmark harness's variant definitions (Table entries in Section 4.3) are described in enough structural detail in Section 3 and Section 4 to reimplement independently; code will be released upon acceptance.

REFERENCES

Sönke Ahrens. *How to Take Smart Notes: One Simple Technique to Boost Writing, Learning and Thinking*. CreateSpace Independent Publishing Platform, 2017.

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.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.14831.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts. *arXiv preprint arXiv:2402.09727*, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 9459–9474, 2020. arXiv:2005.11401.

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 (ACL)*, 2024. arXiv:2402.17753.

Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020. arXiv:1603.09320.

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

Joon Sung Park, Joseph C. O'Brien, Carrie J. 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 (UIST)*, 2023. arXiv:2304.03442.

Qwen Team. Qwen3-VL-embedding and qwen3-VL-reranker: A unified framework for state-of-the-art multimodal retrieval and ranking. *arXiv preprint arXiv:2601.04720*, 2026.

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

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP-IJCNLP)*, 2019. arXiv:1908.10084.

Mohammad Tavakoli, Alireza Salemi, Carrie Ye, Mohamed Abdalla, Hamed Zamani, and J. Ross Mitchell. Beyond a million tokens: Benchmarking and enhancing long-term memory in LLMs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026. arXiv:2510.27246.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.10813.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for LLM agents. *arXiv preprint arXiv:2502.12110*, 2025.

Puxuan Yu, Luke Merrick, Gaurav Nuti, and Daniel Campos. Arctic-embed 2.0: Multilingual retrieval without compromise. *arXiv preprint arXiv:2412.04506*, 2024.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*, 2024.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024. arXiv:2305.10250.

A Extended benchmark suite description

Four standalone scripts measure the storage and retrieval layers directly, seeding their corpus by calling the object-store/graph-store write methods directly rather than through the normal add-a-memory path, so the (expensive, LLM-bound) evolution step does not dominate storage-layer measurements: an object-store suite (RustFS put/get/delete latency and throughput at 1/4/8 concurrent threads, plus a 1KB–1MB payload sweep); a graph-store suite (FalkorDB upsert/link/vector-search latency and throughput as corpus size grows, 100→2000 by default, plus memory use vs. corpus size); an embeddings suite (per-backend latency/throughput and peak resident-memory delta from loading each model); and an end-to-end suite (adding memories with evolution included, plus both retrieval modes, at modest corpus size); this last suite has not yet been run at the time of writing.

B Full asymptotic complexity derivation

FalkorDB's vector index implements Hierarchical Navigable Small World (HNSW) graphs (Malkov & Yashunin, 2020) (default M=16, `efConstruction`= 200, `efRuntime`= 10), a multi-layer proximity graph in which a query greedily descends from a coarse top layer to progressively finer ones rather than scanning every stored vector. HNSW's own analysis gives both approximate nearest-neighbor query and insertion expected $O(\log N)$ time, so nearest-neighbor retrieval and the index-insertion component of writing a memory both inherit this bound rather than the $O(N)$ a brute-force scan would cost. Space is $O(Nd)$ for the raw vectors (d the embedding dimensionality) plus $O(NM)$ for the proximity graph's own edges, consistent with FalkorDB's documented estimate of vectors $\times$ dimensions $\times$ 4 bytes, plus roughly 20% index overhead. Writing a memory's node properties is a single update keyed on its unique identifier, $O(1)$ expected under the hash- or index-based id lookup graph databases use for exact-match node patterns, combined with the $O(\log N)$ HNSW insertion above, so one call is $O(\log N)$ overall. Creating a link between two already-resolved node ids requires no search, so it is $O(1)$, consistent with it being the fastest and flattest of the three FalkorDB operations measured. The evolution procedure issues one $O(\log N)$ vector-search call for its k nearest neighbors, then does $O(k)$ work to build the evolution prompt and evaluate the membership check in Equation 2 against the k shown neighbor ids; because k is a fixed constant, neither this check nor the evolution prompt's size grows as the graph grows. Neighbor-expanded retrieval performs one $O(\log N)$ vector search followed by up to k additional $O(1)$ RustFS lookups for linked neighbors, so it is $O(\log N + k) = O(\log N)$.

Mem0 and Graphiti complexity. This analysis is based on their installed source (`mem0ai` 2.0.19, `graphiti-core` 0.30.1) rather than their papers. Mem0's memory-write path runs, per addition: a fixed-size context fetch (the last 10 messages, $O(1)$), one nearest-neighbor search bounded to the 10 closest candidates (ANN-indexed, $O(\log N)$ under the same HNSW-family argument, since Mem0's default Qdrant backend uses an HNSW index too), and exactly one LLM extraction call over that bounded candidate set, structurally identical in shape to graphmem's own evolution procedure: one bounded search, one LLM decision over a fixed-size candidate list, $O(\log N)$ overall, not $O(N)$. Graphiti's entity-resolution step embeds each newly extracted entity and runs one nearest-neighbor search per entity against the existing graph, an ANN candidate search rather than a scan over every existing node, before an LLM call decides, per candidate set, whether the extraction is a duplicate; entity extraction itself is prompted with a fixed window (the current episode plus the last 4 prior episodes), not the full episode history. Each of these per-entity resolution rounds is consequently $O(\log N)$ by the same argument, and Graphiti runs up to five such search-then-LLM-decision phases per episode (entity extraction, entity resolution, fact extraction, fact resolution, temporal extraction/invalidation) against graphmem's and Mem0's one. The $2.8\times$ measured per-turn cost gap Section 5 reports is this call-count multiplier made concrete, not evidence any of the three systems scales worse with corpus size than the others: all three share the same bounded-candidate-retrieval-then-LLM-decision architecture, and consequently the same $O(\log N)$-per-operation shape, as far as this analysis and this direct source reading can establish.

Store	Operation	mean (ms)	p95 (ms)	ops/s
RustFS (c=1)	put	3.80	5.45	262.8
RustFS (c=1)	get	4.42	6.33	226.1
RustFS (c=1)	delete	3.76	7.21	265.6
RustFS (c=8)	put	10.41	16.46	751.8
RustFS (c=8)	get	7.54	15.91	1005.9
RustFS (c=8)	delete	13.62	28.23	567.3
RustFS, put (1KB payload)	–	2.77	3.27	361.6
RustFS, put (1MB payload)	–	12.46	15.44	80.3
FalkorDB (N=100)	upsert_memory	1.96	2.47	511.2
FalkorDB (N=100)	vector_search	2.01	2.88	498.0
FalkorDB (N=100)	link_memories	0.47	0.77	2127.2
FalkorDB (N=2000)	upsert_memory	3.08	3.99	324.2
FalkorDB (N=2000)	vector_search	2.09	3.22	479.1
FalkorDB (N=2000)	link_memories	0.92	1.53	1090.9

Table 3: Storage-layer latency/throughput (c = concurrent threads, N = corpus size). Single-instance Docker Compose deployment, local host.

C FULL LATENCY/THROUGHPUT TABLES

Delete scales worse than put/get at every concurrency step: from c=1 to c=4, put/get throughput roughly triple while delete only reaches $2.2\times$, and from c=4 to c=8 delete throughput drops slightly (578→567 ops/s) while its p95 latency nearly triples.

Backend	c	mean (ms)	p50 (ms)	ops/s	load RSS (GB)
Snowflake	1	30.9	32.5	32.3	3.35
Snowflake	4	99.3	94.4	39.9	3.35
Qwen3-VL-Embedding	1	69.5	49.6	14.4	2.40
Qwen3-VL-Embedding	4	197.5	167.7	20.1	2.40

Table 4: Embedding backend latency/throughput (c = concurrent threads), n=100 text notes per backend/concurrency pair.

Qwen3-VL-Embedding's c=1 maximum latency (2068ms, not shown in the table means) is a single-call outlier roughly $40\times$ its own median, consistent with a one-time kernel-compilation cost on the first call to a newly loaded model rather than a per-call cost; it accounts for nearly all of the gap between its mean (69.5ms) and median (49.6ms) latency and does not recur once the backend is warm.

D EXACT BENCHMARK COVERAGE AND COST PROJECTIONS

Table 5 states exact coverage for every benchmark and variant. LongMemEval's shortfall from literal 100% is a structural property of the benchmark, not an oversight: every question carries its own independent multi-session haystack (verified directly, 500 unique `haystack_session_ids` sets across 500 questions, no sharing), so seeding cost scales per question rather than being amortized across a shared conversation the way LoCoMo's does, and full coverage was projected at ∼457h for With Evolution's seeding alone. A stratified sample of 58 questions is reported instead, roughly 10 per category across all 6 types. BEAM separately excludes two of its ten task categories (`instruction_following`, `preference_following`), which have a compliance rubric rather than a ground-truth answer field in the released dataset, rather than scoring them against a non-answer. With Evolution's narrower BEAM scope reflects the same per-turn evolution cost driving LongMemEval's stratified sampling: full coverage at all four scales was projected at ∼567h and not attempted, so With Evolution instead covers 17 of the 100 conversations (5 each at 100K/500K/1M, 2 at 10M truncated to their first 1,500 of ∼19,328 average messages).

Benchmark	Variant scope	Coverage	Questions
LoCoMo	every variant	10/10 samples, 5/5 categories	1,986
LongMemEval	every variant	stratified, 10 questions/type, 6 types	58
BEAM	No Evolution, No Memory	100/100 conversations, all 4 scales	1,600
BEAM	With Evolution	17/100 conversations, all 4 scales	272

Table 5: Exact benchmark/variant coverage behind every result in Section 4.3. LoCoMo seeds 5,882 conversation turns per variant, at both thinking settings.

A staged plan for all three benchmarks' full-scale runs, costed in advance against measured per-turn latencies, undershot its own projection in every case, though not by a uniform margin (Table 6): BEAM and LongMemEval landed within about 6%, while LoCoMo's original estimate undershot by 36%. Unlike BEAM's and LongMemEval's own projections, each partly calibrated against an earlier, smaller-scale run of the same benchmark, LoCoMo's per-turn/per-question rates were borrowed entirely from those two benchmarks' validation runs, whose conversational turns average 152–322 words against LoCoMo's own 22.7, and whose no-memory baseline answers against transcripts up to $40\times$ longer than LoCoMo's 13,377-word average per sample. Refitting each of these two rates to their own measured content length (a fixed-plus-variable linear model over the same three calibration runs, applied to LoCoMo's actual turn and transcript lengths, rather than assuming cost scales in direct proportion to word count with no fixed per-call cost) lowers the projection to 14.4h. This narrows the gap without closing it: LoCoMo's actual run still finished about 21% faster than this length-corrected estimate, down from 36% against the original, so turn and transcript length account for part of the original overestimate, not all of it. This is a different projection method than the separately-run LoCoMo thinking-on rerun elsewhere in this paper, whose 96h actual time overshot a 22–45h estimate extrapolated from other benchmarks' per-call overhead rather than this stage's own measured rates.

Benchmark	Projected	Length-corrected	Actual	Deviation
LoCoMo	17.6h	14.4h	11.3h	-21%
BEAM	44.7h	–	42.3h	-5%
LongMemEval	54.9h	–	51.5h	-6%

Table 6: Projected vs. actual wall time for each benchmark's staged full-scale run. LoCoMo's length-corrected column refits its seeding and no-memory-answering rates to LoCoMo's own measured turn/transcript length rather than the borrowed cross-benchmark average; deviation is measured against this corrected value where one exists, otherwise the original projection.

E STATISTICAL SIGNIFICANCE: METHODOLOGY AND FULL RESULTS

Methodology. For every headline comparison in Section 4.3 and Section 4.5, this study computes the mean per-question F1 difference between two variants, a paired bootstrap 95% confidence interval (10,000 resamples of the paired differences, seeded for reproducibility), and a paired permutation-test p-value (10,000 random sign-flips of the paired differences, two-sided). Both are standard, distribution-free choices for comparing NLP systems on a bounded, often skewed metric like F1, and neither assumes normality the way a paired t-test would. **Pairing.** Within each same-length variant group (LoCoMo's and LongMemEval's five variants, each run on the identical question set; BEAM's With Evolution/A-Mem (Full)/ A-Mem (Storage) triplet, 272 questions; BEAM's No Evolution/ No Memory pair, 1,600 questions), list-position pairing was verified to be exact by checking every question's text matches at the same position across variants (zero mismatches, all groups, full check). BEAM's With Evolution (272 questions, 17 conversations) and No Evolution (1,600 questions, 100 conversations) are not the same list; pairing instead uses the true overlapping subset, matched by (scale, question text), which recovers exactly With Evolution's 272 questions as a subset of No Evolution's 1,600, verified directly, not assumed. All computation is over the real per-question scores this paper's own results are aggregated from, not summary statistics re-derived after the fact.

Benchmark	Comparison	n	mean diff.	95% CI	p
LoCoMo	No Evolution − With Evolution	1986	$+0.0386$	$[+0.0235, +0.0538]$	< 0.0001
LoCoMo	With Evolution − No Memory	1986	$+0.1434$	$[+0.1235, +0.1633]$	< 0.0001
LoCoMo	No Evolution − No Memory	1986	$+0.1820$	$[+0.1624, +0.2016]$	< 0.0001
LoCoMo	A-Mem (Storage) − No Evolution	1986	$+0.0152$	$[-0.0021, +0.0324]$	0.089
LoCoMo	A-Mem (Full) − With Evolution	1986	$+0.0178$	$[+0.0001, +0.0353]$	0.047
LongMemEval	No Evolution − With Evolution	58	$+0.0430$	$[-0.0366, +0.1260]$	0.305
LongMemEval	With Evolution − No Memory	58	$+0.1668$	$[+0.0772, +0.2615]$	0.0009
LongMemEval	No Evolution − No Memory	58	$+0.2098$	$[+0.1067, +0.3143]$	0.0002
LongMemEval	A-Mem (Storage) − No Evolution	58	$+0.0120$	$[-0.0264, +0.0530]$	0.575
LongMemEval	A-Mem (Full) − With Evolution	58	-0.1039	$[-0.1989, -0.0139]$	0.035
BEAM	A-Mem (Storage) − With Evolution (matched 272)	272	-0.0111	$[-0.0381, +0.0159]$	0.415
BEAM	A-Mem (Full) − With Evolution (matched 272)	272	$+0.0110$	$[-0.0170, +0.0391]$	0.440
BEAM	No Evolution − No Memory (matched 1600)	1600	$+0.0444$	$[+0.0308, +0.0581]$	< 0.0001
BEAM	No Evolution − With Evolution (true overlap)	272	-0.0062	$[-0.0298, +0.0188]$	0.611
BEAM	With Evolution − No Memory (true overlap)	272	$+0.0180$	$[-0.0128, +0.0484]$	0.253

Table 7: Paired bootstrap 95% CIs and permutation-test p-values for every headline F1 comparison in Sections 4.3/4.5, computed from real per-question scores (10,000 resamples/permutations each).

Reading the table. Every comparison against No Memory run at genuine scale ($n \geq 1600$, or LoCoMo/ LongMemEval's full n) is significant, several with very small p-values. Comparisons involving With Evolution specifically are mixed: significant only on LoCoMo (No Evolution − With Evolution, $p<0.0001$), plus two considerably weaker cases, LoCoMo's A-Mem (Full) − With Evolution ($p=0.047$) and LongMemEval's A-Mem (Full) − With Evolution ($p=0.035$); every BEAM comparison touching With Evolution is non-significant, consistent with With Evolution's smaller BEAM sample ($n=272$ vs. the $n=1600$ available to No Evolution and No Memory) rather than evidence of a true null effect. The A-Mem (Storage) − No Evolution comparison is not significant on either benchmark where its point estimate favors A-Mem (LoCoMo $p=0.089$, LongMemEval $p=0.575$); the point estimates reported in Table 1 should not be read as a clean win for A-Mem's simpler configuration.

Multiple comparisons. Table 7 reports 15 headline tests at once, so a nominal $\alpha=0.05$ threshold read off each p-value in isolation understates the chance some would appear significant by test volume alone. Applying Benjamini-Hochberg false-discovery-rate control at $q=0.05$ across all 15 (equivalently, a Bonferroni threshold of $0.05/15 \approx 0.0033$ gives the identical result here) leaves exactly six comparisons significant: LoCoMo's No Evolution − With Evolution ($p<0.0001$) and all five comparisons against No Memory at genuine scale (LoCoMo's two, $p<0.0001$; BEAM's, $p<0.0001$; LongMemEval's two, $p=0.0002/0.0009$). The two weaker A-Mem (Full) − With Evolution findings above do not survive this correction ($p=0.047$ and $p=0.035$ both rank below the sixth-largest Benjamini-Hochberg critical value), so they are reported here as point estimates worth noting, not as an established effect: this paper's only ablation comparison that remains significant after correcting for the full set of tests conducted is LoCoMo's own No Evolution − With Evolution result.

F Full per-category QA results

LoCoMo, category detail. Table 9 breaks down per-category F1. No Memory edges out both graphmem variants only on the adversarial category, the one category scored against a forced-choice `adversarial_answer` field, where seeing the entire transcript may make ruling "not mentioned" in or out easier than retrieval does when the deciding memory was not among the top-k. No Evolution leads With Evolution on every other category, most visibly on multi-hop and temporal questions, exactly the categories where linking related notes together should help most, an advantage that does not clearly appear in With Evolution's favor.

LoCoMo, thinking-on comparison. A second full run, same 1,986-question slice and model, with thinking left on, took 346,043.1s (96h7m) against the thinking-off run's 40,540.2s (11h16m), an $8.5\times$ increase; every variant's F1 drops with thinking on (Figure 3), and thinking off wins on F1 in

Variant	n	F1	BLEU-1	ROUGE-L	METEOR	SBERT-sim.	EM
LoCoMo (all 10 samples, 5 categories), thinking off							
With Evolution	1986	0.3380	0.2673	0.3384	0.2155	0.4859	0.0916
No Evolution	1986	0.3766	0.2979	0.3785	0.2375	0.5302	0.1017
No Memory	1986	0.1947	0.1557	0.1956	0.1408	0.3428	0.0509
LongMemEval (stratified, 10 questions/type, 6 types), thinking off							
With Evolution	58	0.3573	0.3235	0.3779	0.2420	0.5735	0.1552
No Evolution	58	0.4003	0.3843	0.4276	0.3003	0.5976	0.2241
No Memory	58	0.1905	0.1755	0.1816	0.1265	0.3781	0.0862
BEAM (pooled, all 4 scales, asymmetric coverage; see D), thinking off							
With Evolution	272	0.2815	0.1838	0.2424	0.1791	0.5574	–
No Evolution	1600	0.2876	0.1920	0.2573	0.1776	0.5704	–
No Memory	1600	0.2433	0.1681	0.2101	0.1629	0.5214	–

Table 8: Full metric set by benchmark and variant (Section 4.3). BEAM's harness does not report exact match.

Category	With Evolution	No Evolution	No Memory
`multi_hop`	0.2839	0.3289	0.1592
`temporal`	0.3016	0.3656	0.1528
`open_domain`	0.1594	0.1783	0.1417
`single_hop`	0.4308	0.4764	0.1850
`adversarial`	0.2619	0.2695	0.2769

Table 9: LoCoMo per-category F1, thinking off (1,986 questions total across the 5 categories).

every category for every variant, unlike BEAM (Appendix G), where thinking helped one category enough to matter. No Memory spends more wall time answering (46,767.4s) than With Evolution (33,530.3s) despite doing no seeding at all, since thinking mode's per-question cost scales with prompt size and No Memory hands the model an entire raw transcript per question.

LongMemEval, category detail. Table 10 breaks down the stratified sample by question type. `temporal-reasoning` is the one type No Memory wins outright, though all three scores sit close together, more consistent with a difficult question type than memory hurting specifically. `single-session-preference` is a near-three-way tie where With Evolution edges out the other two, its only category win and narrowest margin. No Evolution leads With Evolution on 4 of 6 types, most sharply on `multi-session`, where With Evolution nearly fails the type outright, structurally similar to what BEAM's cross-session-reasoning category tests, where the same pattern appears independently at full scale (Section 4.3).

Type	n	With Evolution	No Evolution	No Memory
`multi-session`	10	0.0296	0.1834	0.0324
`temporal-reasoning`	10	0.2281	0.2198	0.2346
`knowledge-update`	9	0.3889	0.4464	0.1437
`single-session-user`	10	0.6746	0.7413	0.2950
`single-session-assistant`	10	0.6496	0.6553	0.3168
`single-session-preference`	9	0.1562	0.1336	0.1077

Table 10: LongMemEval per-type F1, stratified sample (58 of 500 questions).

BEAM, category and scale detail. Table 11 breaks down BEAM's eight covered categories. No Evolution edges out With Evolution overall but not consistently: With Evolution leads 4 of 8 categories and No Evolution the other 4, a mixed picture unlike LoCoMo's consistent No Evolution advantage. Per scale, With Evolution leads at 100K and 10M-truncated, while No Evolution leads at 500K and 1M, so neither variant's advantage is tied to conversation length in a simple way.

Category	With Evolution (n=34)	No Evolution (n=200)	No Memory (n=200)
`abstention`	0.3494	0.3280	0.2934
`contradiction_resolution`	0.1794	0.1519	0.1618
`event_ordering`	0.2154	0.1782	0.1452
`information_extraction`	0.3650	0.3830	0.2769
`knowledge_update`	0.3356	0.3999	0.2136
`multi_session_reasoning` (cross-session reasoning)	0.2720	0.3393	0.2931
`summarization`	0.2127	0.2428	0.2024
`temporal_reasoning`	0.3224	0.2780	0.3599

Table 11: BEAM per-category F1, pooled across all four scales. With Evolution's asymmetric coverage (Table 5) means its column is a different, smaller set of conversations than the other two.

G THINKING-MODE TOGGLE: FULL DETAIL

The Ollama backend enforces a generation-length cap (1000 tokens with thinking off, matching the OpenAI-compatible backend; 4096 with thinking on, to leave headroom for a legitimate `<think>` block) and an independent 180-second request timeout; the evolution procedure already treats a timed-out or malformed response as a graceful no-op, so this cap only bounds latency, it does not change evolution's correctness behavior.

Run	seed, on	seed, off	reduction	total speedup
LongMemEval	30,858.6s (8h34m)	6,423.6s (1h47m)	79.2%	4.7×
BEAM-10M-trunc.	36,648.1s (10h11m)	9,726.3s (2h42m)	73.5%	3.7×
BEAM-1M	42,873.9s (11h55m)	11,815.6s (3h17m)	72.4%	3.6×

Table 12: With Evolution's per-run seeding time (the phase paying one LLM evolution round trip per turn) and total run wall time, thinking on vs. off.

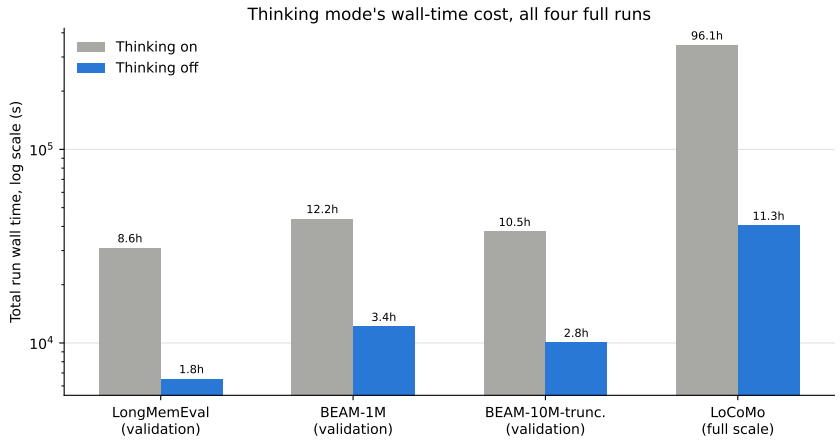


Figure 4: The same seeding-time reduction as Table 12, plotted against LoCoMo's full-scale total wall time for scale, on a log axis.

At BEAM-10M-truncated, With Evolution's F1 on the cross-session-reasoning category dropped from 0.4583 (thinking on) to 0.2011 (thinking off), landing exactly at No Evolution's new score, erasing the lead completely; at BEAM-1M, With Evolution dropped from 0.4014 to 0.2474 while No Evolution and No Memory both moved slightly up. Each of these comparisons rests on only 2 questions per scale (validation-scale runs, pooled as n=4 in Figure 2); two scales moving in the same direction on the same category at this sample size is worth taking seriously rather than dismissing outright, but it is not, on its own, a statistically established effect.

H A-MEM COMPARISON: FULL DETAIL

Both A-Mem variants are vendored from A-Mem's paper-reproduction repository (`WujiangXu/A-mem`, commit `0c8039f`) rather than its JSON-schema "official" package. A-Mem has no evolution-disabled equivalent: its per-turn content-analysis call always runs when an LLM controller is configured, so A-Mem (Storage) pays per-turn LLM cost, unlike graphmem's No Evolution; BEAM and LongMemEval therefore apply With Evolution's own scoping to both A-Mem variants rather than letting A-Mem (Storage) run at No Evolution's true full scale.

Wall-clock cost and answer quality are not correlated here: A-Mem (Full) costs more wall-clock seeding time than graphmem's own With Evolution on both BEAM ($2.7\times$) and LongMemEval ($1.2\times$), despite doing conceptually less per turn (one content-analysis call, no explicit cross-note linking decision); A-Mem's LLM-driven ingestion is not cheaper than graphmem's, even where it scores worse. A-Mem (Storage) is the cheapest memory-using configuration measured in this comparison.

I EXTERNAL REFERENCE POINTS AND PILOT RUNS FOR MEM0 AND GRAPHITI

Mem0 and Zep (Section 2) each report their own published results on one of this paper's three benchmarks, though on a different backend, metric, and (for Mem0) category scope than this paper's harness, so they are reported here as external context, not a controlled comparison; none of these numbers were re-derived or re-run as part of this study.

Mem0 (Chhikara et al., 2025) evaluates on LoCoMo using GPT-4o-mini and only 4 of LoCoMo's 5 categories (it omits adversarial), reporting per-category F1/BLEU-1/LLM-judge scores rather than an aggregate F1. Averaging its own reported per-category F1s unweighted, an approximation computed for this comparison, not a number Mem0's paper states directly, and not directly comparable to this paper's per-question-weighted F1, since LoCoMo's categories have very different question counts, gives $\sim$0.41 for Mem0's best configuration, alongside its own re-run of A-Mem (temperature 0) at $\sim$0.25 and of Zep, run via Zep's hosted platform, at $\sim$0.37. This is a third, independently-implemented setting in which A-Mem-style evolution trails simpler retrieval, though the backend and metric differences mean it is suggestive corroboration of Section 4.5's finding, not confirmatory evidence for it.

Zep (Rasmussen et al., 2025) evaluates on LongMemEval using its own native correctness metric (GPT-4o-as-judge accuracy, not F1) rather than the metrics Section 4.3 reports, so its numbers are not on the same scale as Table 1 at all: 63.8%/71.2% (gpt-4o-mini/gpt-4o) against a 55.4%/60.2% full-context baseline it reports itself (exact question count not stated in its paper). This points the same qualitative direction as this paper's own no-memory comparisons, memory-augmented retrieval beats a no-memory baseline, without being evidence either way on this paper's own evolution-versus-simpler-ablation question.

Pilot runs on Mem0 and Graphiti. Mem0 (`mem0ai`, its OSS vector-memory configuration; graph-memory support was removed from Mem0's open-source SDK as of `mem0ai` ≥ 2.0, so this is not the graph-augmented variant its own paper also reports) and Graphiti (`graphiti-core`, using its FalkorDB driver rather than its default Neo4j) were added as two further on-harness variants, both pointed at the same local Ollama and FalkorDB services this paper's own results use, though configured with `qwen2.5:7b` (thinking off) rather than the `qwen3:latest` every graphmem and A-Mem result in this paper uses, a model-choice difference this pilot does not control for. A validation run (a few synthetic turns, one question) confirmed both integrate correctly against this paper's existing FalkorDB and Ollama services before either was run at pilot scale.

Table 13 summarizes both pilots. Mem0's run completed cleanly (one LoCoMo sample, 419 turns, single-hop questions only, n=70), with no turns or questions skipped by this harness's own timeout wrapper; extrapolating each phase by its own measured rate to LoCoMo's full 1,986 questions/5,882 turns projects $\approx$ 20.8h total, $1.8\times$ graphmem's own With Evolution variant's measured full-scale time (11.3h), the same order of magnitude as every other variant already run at full scale in this paper, not a projection that rules a full run out. Graphiti's run did not complete: seeding stopped at 360

of 550 turns, before the question was reached, so no F1 is available, but the seeding rate was stable up to that point, a $\approx 2.8\times$ multiplier over Mem0's measured rate on the same backend, consistent with Graphiti's documented five-LLM-call-per-episode pipeline (entity extraction, entity resolution, fact extraction, fact resolution, temporal extraction/invalidation) against every other system's single call per turn. LongMemEval gives each of its 500 questions an independent haystack (seeding cost is not amortized the way LoCoMo's shared-conversation design allows, the same structural fact Section 4.3 already states for graphmem's own LongMemEval coverage), so even graphmem's own 58-question stratified sample would cost ≈ 290h (≈ 12 days) for Graphiti's seeding phase alone on this hardware, and a literal full 500-question run $\approx 2{,}500$h (≈ 104 days). Neither is attempted here; this cost estimate is the pilot's primary product, and sizes a full run for later scheduling even though the pilot run itself did not finish.

System	Pilot scope	Seeding rate	F1
Mem0	LoCoMo, 1 sample, 419 turns, single-hop only (n=70)	11.6s/turn	0.4188
Graphiti	LongMemEval, 1 question, 550-turn haystack, 360 seeded before stopping	≈33s/turn	did not finish

Table 13: Mem0 and Graphiti pilot runs on this paper's own backend (Ollama, `qwen2.5:7b`). Mem0's pilot ran 4,849.4s total. Projected full-scale costs: Mem0 ≈20.8h for full LoCoMo; Graphiti ≈290h for graphmem's own 58-question LongMemEval sample, ≈2,500h for a literal full 500-question run.

J Storage-layer comparison against A-Mem and a non-graph baseline: full detail

Comparing each system's own public retrieval API instead of raw storage primitives (embedding cost included, the fairer "as actually used" comparison) puts the two at rough parity, 1.7–2.4ms vs. 1.9–2.1ms, since embedding cost dominates at this scale for both. Under concurrency, A-Mem's throughput decreases from 419 to 200–314 ops/s (full search) and from 1,370 to 761–1,038 ops/s (even for the raw scan alone) as concurrency rises from c=1 to c=4–8, a single in-process Python object hit by multiple threads does not parallelize the way a client-server database does.

Non-graph baseline. FalkorDB is durable via local disk (a Redis module, inheriting Redis's own RDB/AOF persistence) and client-server, but it is also a graph database; A-Mem's storage is neither durable nor client-server. To separate these two properties, the same upsert/vector-search operations (Table 14) were run against Qdrant, a durable (also local-disk, via a write-ahead log), client-server vector database with no graph layer, at matched corpus size and concurrency, using the same harness as every other storage benchmark in this paper. Qdrant's raw latency is comparable to FalkorDB's at c=1 (2.3–2.5ms vs. FalkorDB's ∼2.0–3.1ms), but its throughput gain under concurrency is smaller: 515 to 591 ops/s from c=1 to c=8 at N=2000 (roughly $1.15\times$), against FalkorDB's roughly $2\times$ gain over the same range. Being a durable, client-server store is not, on its own, sufficient to reproduce FalkorDB's full concurrency advantage; something beyond client-server architecture, plausibly FalkorDB's own connection/threading model or something graph-specific, contributes the remainder. `link_memories` has no non-graph analog and so is not part of this comparison.

Store	Operation	N	c	ops/s
Qdrant	upsert	100	1	393.7
Qdrant	vector_search	100	1	448.2
Qdrant	vector_search	100	8	598.5
Qdrant	upsert	2000	1	432.5
Qdrant	vector_search	2000	1	515.0
Qdrant	vector_search	2000	8	590.8

Table 14: Qdrant non-graph baseline: upsert/vector-search throughput at matched corpus size (N) and concurrency (c), same harness as Table 3.

K DURABILITY FAULT-INJECTION: METHODOLOGY AND FULL RESULTS

Section 3's ordering guarantee, that a crash during `add_note()` can only leave the graph index briefly behind the durable store, never ahead of it, is tested directly here against the live RustFS and FalkorDB services this paper's own results use, not argued from the code alone. The LLM controller is replaced with a fixed, instant stub returning a canned decision, so this experiment makes no network call to Ollama and cannot be confounded by model variance; every other component (embedding model, RustFS, FalkorDB) is the real, unmocked service.

Two failure points are injected, each over 30 independent trials against fresh graph namespaces. **Point A**: execution is interrupted immediately after the first RustFS write, before the first FalkorDB upsert, the core ordering claim. A trial passes if, afterward, RustFS holds the note durably while FalkorDB correctly does not yet contain the corresponding node (checked by a direct Cypher existence query, independent of the patched upsert call). **Point B**: one neighbor note is seeded normally, then a second note is added with evolution enabled and a canned response instructing a tag revision; execution is interrupted immediately after evolution's decision but before the resulting re-upsert/re-write pair (Section 3.3). A trial passes if both stores are left holding the same consistent pre-evolution snapshot, RustFS and FalkorDB both lack the revised tag, rather than a partial or divergent write.

Every trial in both settings passed: 30 of 30 for Point A, 30 of 30 for Point B, zero instances of FalkorDB holding a node RustFS never durably recorded, and zero instances of a partial post-evolution write reaching one store but not the other. This does not test every possible failure mode (a crash mid-network-write to either service individually is not reproducible by interrupting the calling Python process alone), but it directly confirms the two ordering boundaries `add_note()`'s own docstring claims, against the real deployed services rather than mocked call order alone.

L CONCURRENCY CORRECTNESS: METHODOLOGY AND FULL RESULTS

Section 3.1's shareability goal, that two agents can use one memory store concurrently without external locking, was previously measured only as raw throughput (Appendix J): FalkorDB's ops/sec scaling under concurrent load, isolated from LLM cost. That demonstrates performance, not correctness under real concurrent writes; this appendix tests the latter directly.

Two agents ran concurrently against one shared FalkorDB graph and RustFS bucket, each an ordinary `GraphMemorySystem` instance with evolution enabled, no additional locking or coordination beyond what the storage layer itself provides. Agent A used the host-level Ollama instance this paper's other results use; Agent B used a second, independent Ollama instance (a separate container, same `qwen3.6:latest` model), so neither agent's evolution calls could serialize behind the other's on a single shared LLM backend, isolating genuine write concurrency at the storage layer as the property under test. Each agent added 10 real notes with evolution enabled, interleaved via concurrent execution rather than run sequentially.

Both agents finished in 382.5s combined wall time. Afterward: all 20 notes were durably present in RustFS (0 missing); the shared FalkorDB graph contained exactly 20 `Memory` nodes, matching the expected count with neither a lost upsert nor a phantom duplicate from a concurrent `MERGE` race; and of the 32 `LINKED_TO` edges evolution created across both agents, every one resolved to two real notes from this run, none dangling or referencing the wrong agent's memory. No coordination beyond FalkorDB's and RustFS's own per-operation atomicity was needed to obtain this result, consistent with the stateless design Section 3.2 describes: correctness here follows from the stores' own guarantees, not from anything graphmem's own orchestration layer adds on top.

This is a small-scale, single-run demonstration (2 agents, 20 notes, one seed), not an exhaustive concurrency stress test; a genuinely adversarial test would target the same small set of neighbor notes from many agents at once to maximize contention, which this run's independent notes do not. That is direct future work.

M MEMBERSHIP-CHECK RATE: METHODOLOGY AND FULL RESULTS

Section 3.3 rules out a link to anything outside the shown neighbor set $N_k(x)$ by construction, but how often the LLM actually proposes a connection outside that set, the exact failure mode A-Mem's own position-based indexing bug allows, was not previously measured. This appendix reports that rate directly: `GraphMemorySystem` tracks it as `evolution_link_stats`, a running count of every "strengthen" connection the evolution LLM proposes against how many fall within the neighbor set it was actually shown, incremented in `process_memory()` at the same point the membership check itself runs.

Two real LoCoMo samples (788 conversation turns) were seeded through the With Evolution variant end to end, the same `qwen3:latest`, thinking-off backend every other result in this paper uses, no mocking of any kind. Across both samples, the evolution LLM proposed 1,240 connections; 1,238 (99.8%) fell within the shown neighbor set, and 2 (0.2%) did not. Those 2 are exactly the connections that would have become hallucinated links under A-Mem's original position-based indexing (Section 1) had graphmem's membership check not existed. The rate is low at this sample size, consistent with most evolution decisions genuinely referring to a retrieved neighbor rather than a confused one, but it is not zero: the membership check has a real, if infrequent, effect on this paper's own seeded corpora, not only a hypothetical one. A larger-scale measurement across all three benchmarks is direct future work, costed the same way Appendix D costs every other full-scale run in this paper.

N FULL LIMITATIONS DISCUSSION

graphmem's default (a local Ollama-served model) trades cloud cost and API-key requirements for whatever that model's reasoning and JSON-schema adherence happen to be. Section 4.4 is a concrete instance of this limitation mattering more than expected: a single, previously unexamined configuration choice changed both evolution latency and answer quality substantially, since disabling thinking removes the model's own natural stopping point on a JSON-schema-constrained call. The generation-length cap and request timeout that address this are calibrated for the Ollama backend specifically; the OpenAI-compatible backend was left unchanged, since an unsupported reasoning-effort value is a real API error for a non-reasoning OpenAI model rather than a no-op, and that backend is untested here.

A separate methodological point applies to every With Evolution-variant result reported in this paper. Evolution's link decisions are built directly from each note's derived summary and keywords (Section 3.3), so their availability at seeding time is not cosmetic for With Evolution: without them, every linking or revision choice is made from note content and a placeholder rather than the semantic metadata the evolution prompt is designed to use. No Evolution is unaffected either way, since it never invokes the evolution step and retrieves by embedding similarity alone, which depends only on note content. A dedicated rerun of With Evolution with this metadata available at seeding time, at the exact scope and settings behind Table 1 (same thinking-off setting, same question sets on all three benchmarks), was compared against the results reported there with the same paired bootstrap/permutation methodology Appendix E uses elsewhere. Table 15 reports the result: none of the three deltas reach significance. LongMemEval's point estimate moved the most, but at $n{=}58$ its interval is wide and crosses zero, the same underpowered-sample pattern Appendix E already reports for other LongMemEval comparisons. This metadata has a real, identified causal path into With Evolution's link decisions; measured at full scale, its availability does not change With Evolution's reported F1 by a statistically significant amount on any benchmark tested, so Table 1's numbers stand as reported.

Benchmark	n	**mean diff.**	**95% CI**	p
LoCoMo	1986	-0.0008	$[-0.0139, +0.0124]$	0.913
BEAM	272	-0.0062	$[-0.0259, +0.0135]$	0.536
LongMemEval	58	$+0.0382$	$[-0.0274, +0.1089]$	0.290

Table 15: With Evolution's F1 change when seeding-time content-analysis metadata is available, vs. the results reported in Table 1 (10,000 resamples/permutations each, list-position pairing verified exact on all three benchmarks).