MemForest: An Efficient Agent Memory System with Hierarchical Temporal Indexing

Han Chen
National University of Singapore
Singapore, Singapore
chenhan@u.nus.edu

Zining Zhang
National University of Singapore
Singapore, Singapore
zzn@nus.edu.sg

Wenqi Pei
National University of Singapore
Singapore, Singapore
wenqi_pei@u.nus.edu

Bingsheng He
National University of Singapore
Singapore, Singapore
dcsheb@nus.edu.sg

Ming Wu
Zero Gravity Labs
United States
ming@0g.ai

Jason Zeng
Zero Gravity Labs
United States
jason@0g.ai

Michael Heinrich
Zero Gravity Labs
United States
michael@0g.ai

Wei Wu
Zero Gravity Labs
United States
wei@0g.ai

Hongbao Zhang
Zero Gravity Labs
United States
peter@0g.ai

ABSTRACT

Memory is a fundamental component for enabling long-context LLM agents, supporting persistent state across interactions through a continuous serve-and-update lifecycle. Despite substantial prior work, existing systems suffer from significant maintenance overhead due to two key limitations: coarse-grained state management and inherently sequential update pipelines. In particular, updates are often tightly coupled with LLM inference and require full-state rewrites, leading to poor scalability and growing latency as memory accumulates. To address these challenges, we present **MemForest**, a memory framework that reformulates agent memory as a write-efficient temporal data management problem. MemForest breaks the sequential bottleneck via parallel chunk extraction, decoupling memory construction into concurrent, independent operations. To further eliminate coarse-grained maintenance, we introduce *MemTree*, a hierarchical temporal index that organizes memory as time-ordered trees rather than flat global summaries. This design replaces full-state rewrites with localized per-node updates, reducing maintenance cost to the affected tree paths while naturally preserving temporally evolving states. We evaluate MemForest on two long-context memory benchmarks, LongMemEval-S and LoCoMo. On LongMemEval-S, MemForest achieves the best overall performance among stateful baselines, reaching 79.8% pass@1 accuracy while sustaining a memory construction throughput approximately 6× higher than state-of-the-art approaches including EverMemOS.

KEYWORDS

LLM agents, agent memory, persistent memory, temporal indexing, hierarchical retrieval, write-efficient memory

1 INTRODUCTION

Large language model (LLM) agents are increasingly expected to sustain personalized and stateful behavior across interactions that span days, weeks, or months [21, 24, 31, 40]. This requirement arises in applications such as conversational assistants, long-lived task agents, and interactive social agents, where useful behavior depends on preserving user preferences, prior commitments, and accumulated experiences over time. This, in turn, requires an efficient and effective memory system that transforms interaction streams into a structured memory state that remains useful as evidence accumulates and user state evolves. Recent memory systems have made substantial progress in managing long-context interactions through hierarchical structures, online/offline consolidation, and temporal graphs [3, 7, 11, 15, 25]. At the same time, recent database systems work has begun to treat LLM+retrieval workloads as first-class data systems, optimizing retrieval–inference pipelining, cache reuse, and persistent vector infrastructures for LLM applications [1, 12, 29, 36]. In this paper, we focus on improving the efficiency and effectiveness of persistent agent memory systems under this systems perspective.

Current memory systems can be viewed as having three core functions: *extraction*, which converts raw interactions into persistent memory records; *retrieval*, which fetches relevant context for downstream response generation; and *maintenance*, which updates, consolidates, and restructures existing knowledge over time [7, 39]. While prior work has substantially improved retrieval quality, storage organization, and query-time reasoning, their online write paths still commonly rely on synchronous extraction, consolidation, or profile-style maintenance over existing memory state [3, 7, 11, 15]. In realistic deployments, the dominant bottleneck shifts to these write-heavy paths, driven by synchronous LLM inference overhead and repeated full-state updates. As illustrated in Figure 1, the dominant latency in representative systems stems from *extraction* and *maintenance*, rather than *retrieval*.

We identify two structural bottlenecks behind this inefficiency. First, existing architectures often embed the LLM directly within the write critical path of *extraction* and *maintenance*. Because the model must synchronously adjudicate every new dialogue chunk—extracting, summarizing, reconciling, or rewriting it against existing memory—the process is forced into a largely serial execution.

Code: https://github.com/Concyclics/MemForest

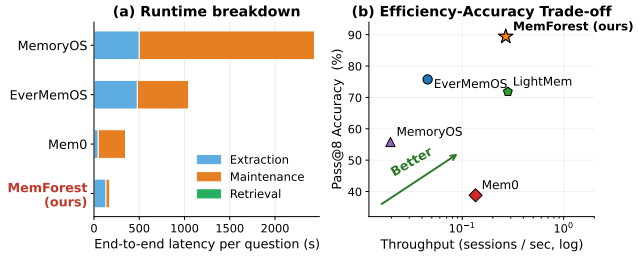

Figure 1: Long-context memory efficiency on LongMemEval-S with Qwen3-30B. (a) Write-heavy extraction and maintenance dominate latency. (b) MemForest improves the update-throughput–accuracy frontier. Main results use pass@1; pass@1–8 curves are in Appendix F.

This creates a severe latency bottleneck that worsens as interaction frequency increases. For example, systems such as EverMemOS rely on write-time semantic processing and consolidation, which improves memory quality but also places substantial LLM work directly on the update path [11]. Second, current systems often operate at a coarse granularity, routinely requiring the model to perform full-state rewrites of compact hot states such as user profiles or global summaries [3, 11, 15, 21]. Even when only minor new evidence arrives, the system must reread and rewrite the entire memory object. As memory accumulates, this imposes a maintenance cost and latency floor that scales with maintained-state size rather than with newly arrived evidence.

However, improving write efficiency alone is not sufficient, because long-context agent memory is inherently temporal [9, 18, 25, 32]. User states evolve, facts are revised, and older information often remains necessary for complex reasoning. For example, if a user first lived in Boston, later moved to New York, and then relocated to San Francisco, a memory system should support not only the current-state query of where the user lives now, but also historical and transition queries such as where the user lived before New York and when the move occurred. We therefore frame long-context agent memory as a write-efficient temporal data management problem, in which persistent memory must remain incrementally maintainable while preserving historical state evolution [2, 6]. The core challenge is to overcome the trade-off between write efficiency and faithful temporal memory representation: a system must minimize the serial delays of *extraction* and the state-size-dependent costs of *maintenance* in order to maximize update throughput, while rigorously preserving historical states for long-context reasoning in order to maximize answer accuracy.

In this paper, we propose **MemForest**, a memory architecture designed around this write-efficient temporal objective. MemForest combines parallel chunk extraction, canonical fact consolidation, and *MemTree*—a hierarchical temporal index that materializes scoped memory as time-ordered trees rather than flat records or repeatedly rewritten profiles. MemForest adopts a similar high-level intuition to write-optimized indexing in database systems, where update costs are reduced by avoiding repeated rewrites of compact indexed state, as exemplified by LSM-trees [20], although the maintained object here is persistent agent memory rather than key-value state. Parallel chunk extraction dismantles the serial *extraction* bottleneck by decoupling and processing new interactions concurrently. Canonical fact consolidation repairs the semantic fragmentation inherently introduced by such parallelization. To optimize *maintenance*, MemTree replaces full-state rewrites with localized per-node updates and lazy summary regeneration, so that index-maintenance cost scales with the affected tree paths and distinct dirty nodes rather than with the total accumulated memory size. At query time, MemForest leverages this structure to perform coarse-to-fine *retrieval*, navigating from broad interval summaries down to precise leaf-level evidence [5, 27, 28].

We evaluate MemForest on two long-context memory benchmarks, LongMemEval-S and LoCoMo [18, 32]. On LongMemEval-S, MemForest achieves the strongest overall pass@1 result among the evaluated stateful baselines, reaching 79.8% answer accuracy while sustaining a write throughput about 6× higher than EverMemOS, the strongest stateful baseline. On LoCoMo, MemForest remains competitive but mixed: its advantages are clearest on temporally structured long-context question answering, while broader multi-hop compositional reasoning remains a setting where broader-context baselines can still help. This efficiency matters because, in long-horizon deployments, *extraction* and *maintenance* costs are paid repeatedly as new sessions arrive rather than once as offline preprocessing. Overall, MemForest improves the memory substrate by accelerating write operations, explicitly preserving temporal state evolution, and exposing retrieval across multiple granularities.

Our contributions are threefold:

- We identify serial LLM-in-the-loop extraction and the state-size-dependent latency of full-state maintenance rewrites as the two dominant structural limitations of long-context agent memory systems.
- We introduce MemForest, a memory architecture that resolves these bottlenecks by combining parallel extraction with hierarchical temporal indexing, enabling localized updates, variable-granularity retrieval, and a persistent, queryable, and temporally evolving memory substrate under continuous writes.
- We show that this architectural shift improves the speed–accuracy trade-off on LongMemEval-S, where MemForest is the strongest among the evaluated stateful baselines, while remaining competitive on LoCoMo with substantially reduced write-path cost across both benchmarks.

The remainder of this paper is organized as follows. Section 2 introduces the workload model and problem formulation for long-context agent memory. Section 3 presents the system overview of MemForest and its core structure, MemTree. Section 4 provides the design and implementation details of its extraction, retrieval, and maintenance workflows. Section 5 evaluates MemForest on LongMemEval-S and LoCoMo. Section 6 provides ablation studies and analysis of the key design choices. We review related work in Section 7, and conclude this paper in Section 8.

2 PROBLEM FORMULATION

2.1 Workload Model

We model an agent memory workload as an online, time-ordered session stream. After observing T sessions, the system state is defined over the finite stream prefix

$$\mathcal{D}_T = (S_1, S_2, \ldots, S_T), \tag{1}$$

where T denotes the number of sessions received so far. Each session S_t is a bounded interaction segment, such as one conversation or one task episode. It consists of a sequence of turns

$$S_t = (u_{t,1}, u_{t,2}, \ldots, u_{t,n_t}), \tag{2}$$

where each turn $u_{t,i}$ is a timestamped user or assistant utterance, and n_t is the number of turns in session S_t.

The key systems issue is that new dialogue is not automatically usable memory. In persistent memory systems, a new session usually has to pass through a write path: key information is extracted, existing memory state is updated or reconciled, and access artifacts such as summaries, embeddings, or indexes are refreshed. Only after this pipeline advances the maintained memory to a stable version can the new information be reliably used by future retrieval and response generation. Thus, memory freshness is governed by the critical path required to incorporate new dialogue, rather than only by the amount of dialogue that has arrived.

Recent agent memory systems maintain memory by structured memory documents, vector-indexed and token-compressed fact stores, or direct search over raw interaction history [3, 7, 11, 15, 19, 25]. Despite their different organizations, their workflows can often be decomposed into three stages: *extraction*, which converts newly arrived interactions into memory records; *maintenance*, which updates, merges, reorganizes, or refreshes existing memory state; and *retrieval*, which recalls relevant memory for downstream response generation [7, 15, 39]. Figure 2 illustrates this common workflow.

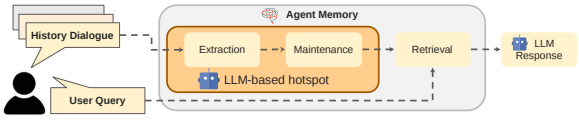

Figure 2: A generic workflow of agent memory systems. New dialogue goes through *extraction*; *maintenance* updates the maintained memory state; future queries trigger *retrieval*.

2.2 Temporal Scope

We use *temporal scope* as the abstraction for organizing long-horizon memory around an evolving target. A temporal scope groups time-ordered evidence about that target. For state-bearing targets, such as a user's residence, health condition, project status, or relationship with an entity, the scope induces a state trajectory over time. For broader targets, such as a dialogue session or a recurring scene, the scope preserves a chronological evidence timeline rather than a single state variable.

We use *evidence item* as an abstract memory-bearing unit: it may be a raw dialogue chunk, an extracted fact, or a maintained memory record, depending on the system. Its *temporal anchor* is the timestamp or time interval inherited from the source session turns. Formally, a scope σ contains an evidence sequence ordered by these temporal anchors:

$$E_\sigma = (e_{\sigma,1}, e_{\sigma,2}, \ldots, e_{\sigma,m_\sigma}). \tag{3}$$

For state-bearing scopes, this ordered evidence may define what is true for the scope at different times. For example, a residence scope may contain evidence that Bob lived in Boston, later moved to Davis, and then moved to Miami. The scope is not merely a bag of facts or a single latest-state summary; it is a temporally organized trajectory of evidence and state changes.

Existing memory systems [3, 7, 11, 15, 19, 25] usually encode such scopes in one of two static forms. One option is to store different time points as independent memory records and retrieve them with embeddings. This preserves local evidence, but semantic similarity does not encode temporal order, predecessor relations, or transition logic. Another option is to consolidate the scope into a mutable text state, such as a profile sentence, summary, or core-memory document. This avoids scattered retrieval, but turns the scope into a hot read-modify-write object. As new evidence accumulates, the text must either grow, making future retrieval and maintenance more expensive, or be compressed, removing intermediate states and transition evidence. These choices create both retrieval errors and write-path bottlenecks, which we analyze next.

2.3 Limitations of Existing Memory Systems

We analyze existing systems through the temporal-scope abstraction. Let a touched scope contain N existing evidence items $E_\sigma = (e_{\sigma,1}, \ldots, e_{\sigma,N})$, and let an incoming session add M new items $\Delta E_\sigma = (\Delta e_{\sigma,1}, \ldots, \Delta e_{\sigma,M})$. The write path must make $E_\sigma \leftarrow E_\sigma \oplus \Delta E_\sigma$ queryable, where $\oplus$ is the system-specific append, merge, update, or materialization operation.

Table 1 reports the dominant dependent write critical path, assuming a constant number of retrieved candidates per new item. The table is intended as a high-level comparison of where prior state appears on the write dependency chain. For MemForest, chunk-level extraction is parallel and independent of existing memory state, so its dependency depth is constant with respect to the touched scope size N under bounded chunk size and sufficient concurrency. The remaining dependent step is the post-extraction local MemTree update: routed records are inserted into scoped temporal trees, and derived artifacts are refreshed only along affected dirty paths. Section 4.2 shows that these dirty paths can be refreshed in parallel and that the dependent path is bounded by tree height, $O(\log N)$. Baseline dependency analysis appears in Appendix B.

2.3.1 Independent Evidence and Wrong-Time Retrieval. One common design stores items in E_σ as independent memory records and retrieves them with embeddings. This preserves local evidence, but embedding similarity is not a temporal relation: it does not encode order, supersession, or predecessor links between states. For a residence scope where Bob lived in Boston, then Davis, and later Miami, the query "Where did Bob live before moving to Miami?" requires the evidence immediately preceding the Miami transition. A record with stronger lexical overlap or higher recency can be ranked above this true predecessor, producing wrong-time retrieval. The system may therefore answer "Boston" because it retrieves an

Table 1: Write critical path for memory maintenance adding M new evidence items to a touched scope or memory object with N existing items.

System	Critical Path	Consequence
Mem0 [3]	$O(M)$	Sequential LLM updates over retrieved evidence
MemoryOS [15]	$O(M+N)$	Ordered promotion and hot-state rewrite
EverMemOS [11]	$O(M)$	Per-unit consolidation into maintained memory
LightMem [7]	$O(M+N)$	Buffer writes plus global consolidation
MemPalace [19]	$O(M)$	Fast append, but no temporal maintenance
MemForest	$O(\log N)$	Local MemTree update with parallel refresh

older residence record, or "Miami" because it retrieves the latest residence record, even though the correct answer is "Davis." The same issue can affect write-time maintenance: fact-store systems such as Mem0 [3] retrieve old records before deciding whether a new evidence item should be added, merged, updated, or deleted; retrieving the wrong point in E_σ can merge non-adjacent states or overwrite historical evidence.

2.3.2 Mutable Scope States and Accumulative Maintenance. Another common design consolidates E_σ into a mutable state s_σ, such as a profile, summary, or core-memory document. Each write updates this state as

$$s_\sigma^{(i)} = \textsc{Update}(s_\sigma^{(i-1)}, \Delta E_\sigma^{(i)}), \tag{4}$$

so LLM-based maintenance serializes later writes behind earlier generated states. This creates a growing-or-compressing dilemma: keeping all evidence makes prompts and maintenance cost grow with N, whereas compression can discard intermediate states and transition evidence. This pattern appears across systems: Mem0 [3] uses LLM-based update decisions over retrieved records; MemoryOS [15] maintains ordered promotion and profile-like states; EverMemOS [11] depends on streaming boundary decisions; LightMem [7] uses buffer-triggered extraction and global consolidation queues. MemPalace [19] avoids these write bottlenecks by appending raw chunks, but it also avoids structured temporal maintenance.

2.3.3 Why These Failures Matter. These two designs lead to complementary failures. A mutable latest-state summary can answer current-state lookup, but may remove evidence needed for historical-state and transition queries. Independent evidence records may preserve local facts, but semantic retrieval alone may select the wrong time point [9, 18, 25, 32].

Consider three sessions with evidence:

- **May 2023:** Bob moves from Boston to Davis.
- **July 2024:** Bob moves from Davis to Miami.
- **January 2025:** Bob buys a house in Miami.

A current-state query, "Where does Bob live now?", can be answered from a compact profile: "Miami." In contrast, "Where did Bob live before moving to Miami?" requires the intermediate Davis state. A profile-style memory may answer "Miami" or fall back to "Boston" after compressing away the transition, while an unordered record store may retrieve the most recent or most semantically similar residence fact instead of the true predecessor. This failure mode is common in long-horizon workloads: In LongMemEval-S [32], knowledge-update and temporal-reasoning questions, which directly require reasoning over changed or time-indexed states, account for 15.6% and 26.6% of the benchmark, respectively; multi-session questions add another 26.6% where evidence is distributed across sessions. In LoCoMo [18], temporal questions account for 42.3%, and multi-hop questions account for 16.2%, often requiring evidence to be composed across a long dialogue history.

2.4 Problem Formulation

Given the online session-stream prefix $\mathcal{D}_T$, our goal is to maintain a persistent memory substrate $\mathcal{M}_T$ that turns new sessions into queryable memory with low cost while preserving temporally evolving state. We focus on three requirements.

Low-latency memory construction. New sessions should become queryable after a short write path. When an incoming session produces new evidence for one or more temporal scopes, the update cost should depend primarily on the new evidence and the affected scopes, rather than on repeatedly rewriting hot summaries or serially adjudicating a large mutable memory state. Since many memory updates invoke LLMs, the system should avoid unnecessary LLM calls and token usage caused by repeatedly rereading or regenerating accumulated state.

Temporal-scope fidelity. The maintained memory should preserve time-local evidence, historical states, and state transitions within each temporal scope. This is necessary not only for current-state lookup, but also for knowledge updates, multi-session recall, and temporal reasoning, where latest-state summaries may forget intermediate states and unordered records may retrieve evidence from the wrong time point.

Localized maintenance. Writes should affect only the temporal scopes and access artifacts touched by new evidence. Such locality reduces the write critical path and also enables efficient rematerialization or migration when memory policies, indexes, or tree configurations change.

MemForest addresses these requirements through three design choices. Canonical facts serve as stable write units, making new evidence mergeable without repeatedly rewriting a mutable profile. Persistent memory state is separated from derived access artifacts, so summaries, embeddings, and index rows can be regenerated selectively. Each temporal scope is materialized as a MemTree: leaves preserve time-local evidence, internal nodes summarize contiguous intervals, and writes touch only affected paths. The next section describes how these are realized in the MemForest architecture.

3 MEMFOREST ARCHITECTURE

Section 2 formulates long-horizon agent memory as maintenance over an online session-stream prefix $\mathcal{D}_T$. The goal is to turn newly arrived sessions into queryable memory with low write-path cost, while preserving temporally evolving state within each temporal scope. MemForest addresses this goal with a shared memory substrate and a scoped temporal index. The shared substrate separates persistent memory state from derived access artifacts, and the temporal index materializes each scope as a MemTree.

Figure 3 shows the architecture. A new session is first processed by parallel extraction and normalized into canonical facts, which

serve as stable write units. These facts are then routed to session, entity, and scene scopes and inserted into the corresponding MemTrees through local, height-bounded update paths. Retrieval first recalls relevant trees and then browses within their temporal hierarchies from coarse interval summaries to leaf evidence. Maintenance edits the persistent state locally and regenerates only affected derived artifacts. Thus, MemForest reduces construction latency through the combination of parallel fact extraction and short MemTree write paths: extraction avoids a single serialized LLM pass over the entire session, while MemTree materialization avoids rewriting an accumulated memory object after extraction. At the same time, scoped MemTrees preserve temporal-scope fidelity, and the persistent/derived separation enables localized maintenance.

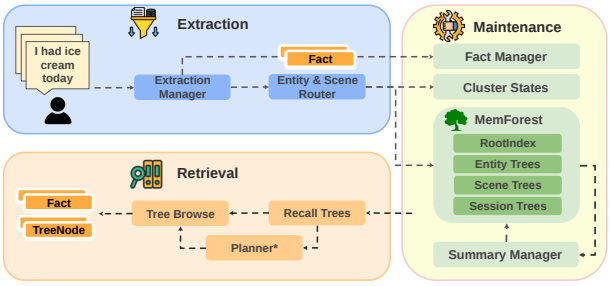

Figure 3: MemForest architecture. Sessions are extracted into canonical facts, routed to scoped MemTrees, and maintained through selective refresh of derived artifacts. Retrieval recalls relevant trees and browses from interval summaries to leaf evidence. The planner is optional.

3.1 Shared Memory Substrate

To shorten the write path, MemForest does not commit new dialogue by rewriting a global profile, a user document, or a compact latest-state summary. Instead, it maintains a shared memory substrate with two layers: persistent state and derived access artifacts. The persistent state is the source of truth. It contains canonical facts, scope assignments, MemTree structure, and source-session references. Derived artifacts include interval summaries, node embeddings, and root-index rows used for retrieval. These artifacts are generated from persistent state and can therefore be refreshed selectively after local edits.

The stable write unit in this substrate is the *canonical fact*. A canonical fact represents one temporally anchored piece of memory with retrieval-ready text, source references, entity mentions, topical signals, and a temporal anchor inherited from the source session. This choice is important for low-latency construction. Parallel extraction may produce fragmented local outputs, but the write path does not need to decide immediately how to rewrite an accumulated memory state. Instead, extracted outputs are first normalized into canonical facts and then inserted into affected scopes. This makes new evidence mergeable and routeable without repeatedly rereading the entire history of the scope.

The same substrate also supports localized maintenance. Since summaries, embeddings, and index rows are derived from canonical facts and tree structure, a later update only needs to invalidate and regenerate the derived artifacts whose dependency paths intersect the affected scopes. This separation is the systems counterpart of the temporal-scope model in Section 2.2: persistent state preserves the time-local evidence, while derived artifacts provide efficient access to that evidence at different granularities.

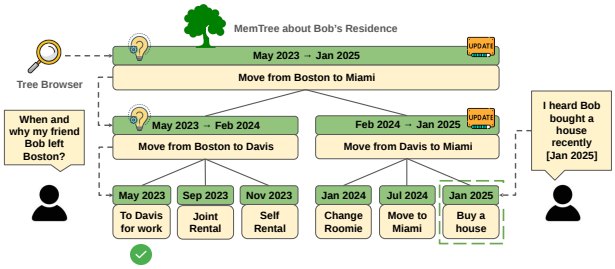

Figure 4: MemTree materializes one temporal scope as a time-ordered hierarchy: leaves preserve local evidence, internal nodes summarize intervals, and the root supports coarse recall. The same structure supports local insertion, dirty-path refresh, and hierarchical retrieval.

3.2 MemTree: Scoped Temporal Index

To preserve temporal-scope fidelity, MemForest materializes each temporal scope σ as a MemTree $\mathcal{T}_\sigma$. A MemTree is a balanced temporal hierarchy: leaves store time-local evidence in temporal order, internal nodes summarize contiguous intervals, and the root provides a coarse representation for forest-level recall. Figure 4 illustrates this design.

MemTree avoids the two failure modes in Section 2.3. Unlike an unordered fact store, it makes predecessor, successor, and interval relations explicit through the leaf order. Unlike a mutable latest-state summary, it does not overwrite intermediate states when new evidence arrives: older states remain represented as leaves or interval summaries even if the root summary reflects the current overall trajectory. Thus, in the residence example from Section 2.3.3, retrieval can browse toward the interval immediately before the Miami transition, instead of relying only on lexical similarity or recency.

MemForest uses three complementary families of MemTrees. A *session tree* preserves the chronology of one source session and provides a fallback view over original interaction context. An *entity tree* groups evidence about a recurring subject, such as a person, project, preference, or state-bearing object. A *scene tree* groups evidence belonging to a semantically coherent situation or topic that may involve multiple entities. These trees are not redundant copies of the same memory: session trees preserve source order, entity trees support subject-centered state evolution, and scene trees capture broader multi-entity context.

MemTree also unifies read and write locality. On the write side, new evidence enters as time-ordered leaves, and only affected ancestor paths need to be refreshed. On the read side, retrieval starts from root and interval summaries and descends only where finer evidence is needed. MemTree is therefore the core architectural abstraction rather than merely an index implementation: it connects

temporal fidelity, coarse-to-fine retrieval, and localized maintenance in the same structure.

3.3 System Workflows

The three runtime workflows jointly realize the requirements in Section 2.4, but they are not independent pipelines. Ingestion produces the persistent state that retrieval uses, retrieval relies on the temporal organization maintained by MemTree, and maintenance keeps derived access artifacts consistent with local edits.

Session ingestion. When a new session S_t arrives, MemForest partitions it into short extraction units, extracts local memory candidates in parallel, canonicalizes the outputs into stable facts, routes those facts to affected scopes, and materializes them into the corresponding MemTrees. The key architectural property is that a new session becomes queryable through local insertion rather than through a global memory rewrite. This directly targets low-latency memory construction: the critical path depends on the incoming evidence and the affected scopes, rather than on the full accumulated state of a mutable profile or summary.

Query-time retrieval. Given a query, MemForest first performs forest-level recall over root representations. The retrieved unit at this stage is a tree, not an atomic fact. This keeps scope-level grouping intact during coarse pruning. MemForest then browses inside the selected trees, moving from interval summaries to more specific child nodes and finally to leaf-level evidence. This workflow preserves temporal-scope fidelity at query time: rather than flattening all memory into an unordered vector pool, the retriever can navigate the temporal hierarchy of a scope and recover evidence from the appropriate interval.

Lifecycle maintenance. Beyond normal ingestion, MemForest supports incremental addition, merge, and targeted deletion. These operations edit persistent state first and regenerate only affected derived artifacts. For example, a merge updates canonical facts or scope assignments before refreshing summaries and embeddings on affected paths; a deletion uses source-session references to identify derived leaves and then refreshes only invalidated ancestors. This realizes localized maintenance because writes affect only touched scopes and access paths. It also enables efficient migration when memory policies, index settings, or tree configurations change: the system can regenerate selected derived artifacts from persistent state without replaying the entire session stream $\mathcal{D}_T$.

Overall, MemForest satisfies the requirements in Section 2.4 by assigning a clear role to each architectural component. Canonical facts make newly extracted evidence stable and mergeable, which shortens the write path. MemTrees preserve time-local evidence, historical states, and transitions within each scope. The separation between persistent state and derived artifacts bounds maintenance to affected scopes and dirty access paths. The next section describes how these architectural choices are implemented in the write path, query path, and lifecycle maintenance procedures.

4 DESIGN AND IMPLEMENTATION

Section 3 and Figure 3 describe the MemForest workflow: sessions are extracted into canonical facts, routed to temporal scopes, materialized as MemTree leaves, refreshed through dirty paths, and later accessed by forest recall and tree browse. This section instantiates that workflow through the write path, query path, and lifecycle maintenance procedures. Parallel extraction reduces front-end LLM latency, canonical facts provide stable write units, MemTree materialization shortens the post-extraction critical path, and dirty-path refresh localizes maintenance.

Running Example. We refer back to the residence example in Section 2.3.3: memory already records that Bob moved from Boston to Davis, a new session states that he moved from Davis to Miami in July 2024, and a later query asks where he lived before moving to Miami. We use this update to illustrate how MemForest extracts, routes, updates, and later browses temporal evidence.

4.1 Write Path: Extraction and Canonicalization

This subsection implements the first stages in Figure 3: session-to-chunk extraction and canonical-fact construction. Given a newly arrived session $S_t = (u_{t,1}, u_{t,2}, \ldots, u_{t,n_t})$, MemForest partitions it into fixed-size extraction chunks

$$C(S_t) = \{c_{t,j}\}_{j=1}^{\lceil n_t/b \rceil}, \qquad c_{t,j} = (u_{t,(j-1)b+1}, \ldots, u_{t,\min(jb,n_t)}) \quad (5)$$

where b is the chunk size in number of turns. We use $b = 2$ by default: two-turn chunks preserve enough local context for fact extraction while keeping extraction calls short and parallelizable. We report the chunk-size sweep in Appendix C. Chunks are processed independently up to the concurrency budget, avoiding a single serialized LLM pass over the full session.

Each extraction call returns memory candidates with source references, temporal anchors, entity mentions, and topical signals. Since chunk-local extraction may produce overlapping outputs, MemForest canonicalizes candidates before indexing: it normalizes surface forms, merges duplicates, and stores the resulting canonical facts in the Fact Manager. Canonical facts provide the bridge between parallel extraction and local MemTree materialization. They make extracted evidence stable and routeable without requiring the system to rewrite an accumulated memory object.

4.2 Scope Routing and Local MemTree Update

This subsection implements the routing stage in Figure 3. After canonicalization, MemForest routes each fact to relevant temporal scopes and emits scoped update records for the maintenance mechanism. Session scope is determined by the source session. Entity scope is induced from normalized entity labels. Scene scope is induced from topical signals and maintained with lightweight cluster states, such as centroids and member-fact identifiers. This routing stage does not require additional LLM calls after extraction. Let $\mathcal{F}_t$ be the facts extracted from S_t. Routing produces records

$$\mathcal{R}_t = \{(\sigma, r) \mid r \in \mathcal{F}_t,\ \sigma \in R(r)\}, \quad (6)$$

where $R(r)$ is the set of scopes touched by record r. Entity and scene trees use canonical facts as leaves. Session trees use source dialogue cells as leaves so that the system keeps a high-fidelity fallback channel to the original interaction.

Although Figure 3 places tree update under maintenance, normal ingestion invokes the same maintenance mechanism immediately after routing. The mechanism separates eager structural edits from lazy semantic refresh. Structural edits attach new leaves, update

placement maps, and split or rebalance the tree if needed. A placement map records which tree leaves are derived from each canonical fact or dialogue cell, enabling later merge and deletion. Semantic artifacts, including summaries, node embeddings, and root-index rows, are refreshed lazily. After inserting a leaf, MemForest marks only its ancestors dirty. Repeated dirty marks are coalesced, so nearby writes refresh a shared ancestor only once. Dirty nodes are then grouped by level and refreshed bottom-up; nodes at the same level, and nodes from different trees, can be processed in parallel.

We number leaves as level 0 and increase levels toward the root, so this loop refreshes dirty nodes bottom-up. Algorithm 1 summarizes this maintenance-backed local update. For a balanced k-ary MemTree with N leaves, the height is $h = \lceil \log_k N \rceil$. A structural insertion marks one leaf-to-root path dirty, so a touched path has $O(\log N)$ dependent refresh depth. When a batch touches multiple paths, the total work grows with the number of distinct dirty nodes, but dirty nodes at the same level and dirty paths in different trees or scopes can be refreshed in parallel. Thus, the post-extraction wall-clock critical path is bounded by the deepest affected tree path rather than by the number of touched paths or the full accumulated memory size. NodeIndex supports node-level embedding retrieval, and RootIndex supports forest-level recall; both are derived artifacts regenerated from dirty summaries.

In the example, the Miami evidence is inserted after the Davis evidence in the Bob entity tree and the residence-related scene tree. Only the ancestor paths above the inserted leaves are marked dirty. MemForest does not rewrite a profile sentence such as "Bob lives in Miami," nor does it resummarize Bob's entire residence history.

Algorithm 1: Local MemTree Update with Lazy Refresh

Input : memory substrate $\mathcal{M}$, routed records $\mathcal{R}_t$
Output: updated persistent state and refreshed derived artifacts

$\mathcal{B} \leftarrow \text{GroupByTree}(\mathcal{R}_t)$
foreach $(\mathcal{T}_\sigma, R_\sigma) \in \mathcal{B}$ **do**
  foreach $r \in \text{SortByTime}(R_\sigma)$ **do**
    $\ell \leftarrow \text{CreateLeaf}(r)$
    $\text{AttachAndRebalance}(\mathcal{T}_\sigma, \ell, r.time)$
    $\text{UpdatePlacementMap}(r, \mathcal{T}_\sigma, \ell)$
    $\text{MarkDirtyAncestors}(\ell)$
$\mathcal{A} \leftarrow \{\mathcal{T}_\sigma \mid (\mathcal{T}_\sigma, R_\sigma) \in \mathcal{B}\}$
$\mathcal{U} \leftarrow \text{CollectDirtyNodesByLevel}(\mathcal{A})$
for $\lambda \leftarrow 0$ **to** $\text{MaxLevel}(\mathcal{U})$ **do**
  foreach $v \in \mathcal{U}[\lambda]$ ***in parallel*** **do**
    if $\text{IsLeaf}(v) \wedge \text{TreeType}(v) \in \{\textsc{entity}, \textsc{scene}\}$ **then**
      $v.summary \leftarrow \text{Passthrough}(v.payload)$
    else if $\text{IsLeaf}(v)$ **then**
      $v.summary \leftarrow \text{SummarizeCellText}(v.payload)$
    else
      $v.summary \leftarrow \text{SummarizeChildren}(v.children)$
    $v.dirty \leftarrow \text{false}$
foreach $v \in \text{AffectedIndexNodes}(\mathcal{A})$ ***in parallel*** **do**
  $v.embedding \leftarrow \text{Embed}(v.summary)$
  $\text{NodeIndexPut}(v)$
$\text{RefreshRootRows}(\mathcal{M}, \mathcal{A})$
return $\mathcal{M}$

4.3 Query Path: Forest Recall and Tree Browse

The query path is designed to avoid the wrong-time retrieval failure described in Section 2.3.1. MemForest does not retrieve only independent facts from a flat vector pool. Instead, it first recalls relevant trees and then browses inside their temporal hierarchies.

Given a query q, forest recall builds a candidate tree set from two signals. Root recall retrieves trees whose root summaries are close to the query, capturing scope-level relevance. Fact-to-tree recall first retrieves atomic facts and then maps them back to the trees in which they appear, recovering trees whose relevance is concentrated in local evidence. MemForest ranks the union pool as

$$C(q) = \text{TopK}_{T \in C_{\text{root}}(q) \cup C_{\text{fact}}(q)} \text{score}(q, T), \tag{7}$$

where $\text{score}(q, T)$ combines root-summary similarity and the best matched fact similarity for tree T. This compact union recall keeps the broad scope signal from roots while retaining lexical, entity-specific, and date-specific cues from facts.

After forest recall, MemForest browses inside the recalled trees. Browse starts from root and interval summaries, descends to promising child nodes, and stops when it reaches leaf-level evidence. The retrieved leaves are resolved back to canonical facts or source dialogue cells, reranked, and assembled into the final answer context.

MemForest supports two browse modes. The embedding-only mode scores candidate child summaries with embedding similarity and follows the best branches. This mode has low online latency and is useful when retrieval cost must be minimized. The LLM-guided mode asks an LLM to choose the next branch from visible child summaries. In the high-accuracy setting, LLM-guided browse can be paired with a planner: given the query and the recalled root summaries, the planner creates a targeted subquery for each tree. The planner is a traversal controller rather than an answer generator; it only makes tree browse more targeted.

In the running example, the query "Where did Bob live before moving to Miami?" may recall both the Bob entity tree and the residence-related scene tree. Flat embedding retrieval may select the latest Miami fact or the older Boston fact. MemForest instead browses the temporal hierarchy and moves to the interval immediately preceding the Miami transition, recovering Davis. This is the query-time counterpart of MemTree's temporal-scope fidelity.

4.4 Lifecycle Maintenance and Update Locality

This subsection implements the maintenance stage in Figure 3. Maintenance is invoked both by normal ingestion, which inserts newly routed records into MemTrees, and by lifecycle operations such as merge, deletion, and migration. In all cases, MemForest edits persistent state first and then regenerates only derived artifacts whose dependency paths intersect the affected scopes.

Merge. When two memory states are merged, MemForest first reconciles canonical facts and scope assignments. Matching scopes are merged by combining their affected MemTrees and refreshing only touched subtrees. Unmatched trees can be copied directly because their persistent state and derived artifacts remain valid.

Delete. When a dialogue segment is retracted, the session registry identifies the canonical facts, dialogue cells, and tree leaves. MemForest removes those leaves, updates placement maps, and marks

invalidated ancestors dirty. Summaries, embeddings, and root rows are then regenerated only along affected paths.

Migration and re-materialization. When memory policies, index settings, or tree configurations change, MemForest does not need to replay the entire session stream $\mathcal{D}_T$. The persistent substrate remains the source of truth, and selected derived artifacts can be regenerated from canonical facts, scope assignments, and tree structure. External memory can also be imported as another forest and integrated through the same merge path.

The Bob example also illustrates this locality. If the July 2024 session is deleted or corrected, the session registry identifies the Miami transition fact and its derived leaves. MemForest removes or updates those leaves and refreshes only the invalidated ancestor paths in the session, entity, and scene trees. It does not regenerate unrelated scopes or replay $\mathcal{D}_T$.

This locality is the main systems claim of the implementation. MemForest does not claim that every part of memory construction is logarithmic: LLM extraction still depends on the incoming session and is accelerated through parallel chunk processing. The narrower post-extraction claim is that materialization and semantic refresh are bounded by affected scopes, dirty paths, and distinct dirty nodes rather than by the full accumulated memory state. This is what allows MemForest to avoid repeated full-state rewrites while keeping temporal evidence queryable.

5 EVALUATION

We evaluate MemForest as an end-to-end persistent memory system for long-context agents. The evaluation follows the requirements in Section 2.4. We first measure whether MemForest shortens the write path that turns new dialogue into queryable memory. We then check whether its query-time overhead is acceptable. Finally, we evaluate whether the same memory substrate preserves answer quality and supports post-build maintenance.

5.1 Experimental Setup

Execution setting. All systems are evaluated under a persistent-memory setting. For each benchmark instance, dialogue history is first transformed into queryable memory through each method's native write path, rather than flattened into a single offline prompt. This reflects the deployment setting of long-horizon assistants, where interaction history accumulates over time and new memory must become queryable after each write.

All methods are evaluated with two open-source LLM backbones, `Qwen3-4B-Instruct-2507` and `Qwen3-30B-A3B-Instruct-2507`, and use `Qwen3-Embedding-0.6B` [34, 38] as the unified embedding model for retrieval and indexing. Each model is served on a dedicated NVIDIA H100 GPU using vLLM 0.18.0 with FlashAttention [4]. Unless otherwise stated, we report both 4B and 30B settings.

Benchmarks and methods. We evaluate on two conversational memory benchmarks: LongMemEval-S [32] and LoCoMo [18]. LongMemEval-S contains 500 question-specific memory instances across six categories: *single-session-user*, *single-session-preference*, *single-session-assistant*, *knowledge-update*, *multi-session*, and *temporal-reasoning*. LoCoMo contains 10 long multi-session dialogue samples with 1,986 questions across *single-hop*, *multi-hop*, *open-ended*, *temporal*, and *adversarial* types.

Table 2: Write-path latency and token cost on LongMemEval. Speedup is normalized to the slowest method in each model setting; larger is better.

	30B setting			4B setting		
Method	Time (s)	Tokens	Speedup	Time (s)	Tokens	Speedup
MemForest	**178.0**	1.143M	**13.7×**	**136.9**	1.106M	**12.4×**
Mem0	353.2	294.1K	6.9×	314.9	289.6K	5.4×
EverMemOS	1048.8	1.455M	2.3×	790.9	1.466M	2.1×
MemoryOS	2439.9	527.8K	1.0×	1695.1	524.5K	1.0×

Unless otherwise stated, **MemForest** denotes the default configuration with union recall, final top-$k = 10$, and planner-guided tree browsing. **MemForest (emb)** uses the same persistent memory substrate but replaces planner-guided browsing with embedding-only tree descent. These two variants expose higher-accuracy and lower-latency retrieval over the same maintained memory state.

We compare against EverMemOS [11], LightMem [7], MemoryOS [15], MemPalace [19], and Mem0 [3]. EverMemOS, MemoryOS, and Mem0 are the primary write-path baselines because they maintain persistent memory through explicit write-time processing. LightMem and MemPalace are included as answer-quality references: LightMem uses a compression-oriented pipeline with an auxiliary small model, while MemPalace indexes raw conversation content without a separate stateful memory write path.

Metrics and alignment. For efficiency, we report write-path wall-clock time and total LLM token usage required to construct queryable memory. We also report query-time retrieval and answer-generation latency. For answer quality, we use pass@1 accuracy as the primary metric and use `DeepSeek-V3.2` [16] as the LLM judge with benchmark-specific grading prompts in Appendix A. The detailed pass@1–8 curves are reported in Appendix F.

For fair comparison, all methods use the same serving setup and answer backbones, and all retrieval-based systems are evaluated under the same final retrieval budget of top-$k = 10$. We preserve each baseline's native workflow whenever possible. EverMemOS uses its agentic retrieval pipeline with evidence sufficiency checking and iterative reformulation; for memory construction, we enable episode and event-log extraction, keep foresight/profile extraction disabled, and keep clustering enabled. LightMem uses its original compressed pipeline with `LLMLingua-2` [23], topic segmentation, and local vector-store retrieval. Mem0 uses its original `infer=True` extraction pipeline. MemPalace is evaluated in raw retrieval mode. MemoryOS uses its three-level memory architecture with `top_k_sessions=10` and `retrieval_queue_capacity=10`.

5.2 Write-Path Efficiency

We first evaluate the main systems target of MemForest: the write path. A persistent memory system repeatedly turns newly arrived dialogue into queryable state. If this path is serialized through long-context LLM calls or repeated global rewrites, memory freshness is bounded by construction latency rather than by retrieval quality.

Table 2 shows that MemForest substantially shortens memory construction under both builder sizes. With the 30B builder, MemForest constructs memory in **178.0s** per question, corresponding to a **13.7×** speedup over MemoryOS. With the 4B builder, it takes

Table 3: Query-time latency breakdown in seconds.

	30B setting			4B setting		
Method	Retrieval	Answer	Total	Retrieval	Answer	Total
MemForest	2.600	2.031	4.60	2.400	1.948	4.30
MemForest (emb)	0.542	1.645	2.19	0.474	1.949	2.42
Mem0	0.027	0.196	0.22	0.030	0.261	0.29
MemoryOS	0.442	1.846	2.29	0.492	0.611	1.10
EverMemOS	2.254	6.232	8.49	2.106	8.860	10.97

136.9s, corresponding to a **12.4**× speedup. MemForest is also consistently faster than EverMemOS and Mem0, showing that the gain does not depend on a particular backbone.

The latency reduction comes from the two write-path design choices in Sections 3 and 4. Parallel chunk extraction avoids a single serialized LLM pass over the full session, while MemTree materialization replaces repeated global memory rewrites with scoped insertion and dirty-path refresh. As a result, the critical path is determined by short extraction calls and affected tree paths, rather than by the accumulated size of a profile-like memory object.

Token cost reflects a latency–cost trade-off. MemForest uses more tokens than Mem0 and MemoryOS because it turns a few long state-dependent prompts into many short parallel calls. Much of this overhead is repeated prompt prefixes and can be amortized by prefix caching. MemForest therefore optimizes memory freshness and wall-clock write latency, not raw token minimization.

This distinction is important for interpreting the efficiency–quality frontier. Mem0 has lower token cost, but its answer quality is much lower in Sections 5.4 and 5.5. EverMemOS is accurate on several categories, but pays much larger construction cost. MemForest moves the frontier by keeping strong answer quality while making persistent memory construction substantially faster.

5.3 Query-Time Latency

Table 3 shows that MemForest provides two query-time operating points. The embedding-only mode keeps online latency low, taking **2.19s** in the 30B setting and **2.42s** in the 4B setting. This is comparable to MemoryOS and much faster than EverMemOS. The default planner-guided mode increases total latency to **4.60s** and **4.30s**, reflecting the additional tree-browse LLM calls used for higher answer quality, while still remaining faster than EverMemOS.

The extra cost is concentrated in retrieval rather than answer generation. This matches the design: planner-guided browsing is a retrieval refinement layer, not a heavier answer-generation workflow. Mem0 remains the fastest online baseline because its retrieval path is minimal, but the accuracy results below show that this speed comes with a large quality loss. Overall, query-time latency is modest compared with write-path construction time, supporting our focus on memory freshness and write-path latency.

5.4 Main Results on LongMemEval

We next examine whether the write-efficient memory preserves answer quality. Table 4 reports pass@1 accuracy on LongMemEval-S; detailed pass@1–8 curves are reported in Appendix F.

MemForest achieves the best overall accuracy under both answer backbones, reaching **70.4%** with 4B and **79.8%** with 30B. The embedding-only variant remains close, reaching **67.6%** and **78.4%**, and still outperforms all external baselines overall. This shows that most of the gain comes from the persistent memory substrate rather than from adding a heavy query-time agent.

Table 4: pass@1 accuracy on LongMemEval. Abbreviations: SS = single-session, Pref. = preference, Asst. = assistant, K-Upd. = knowledge-update, Temp. = temporal-reasoning.

Method	Overall	SS-User	SS-Pref.	SS-Asst.	K-Upd.	Multi-Session	Temp.
Qwen3-4B-Instruct-2507							
MemForest	**70.4**	**95.7**	63.3	82.1	70.5	57.1	66.9
MemForest (emb)	67.6	90.0	50.0	80.4	65.4	55.6	**67.7**
EverMemOS	56.2	80.0	50.0	67.9	62.8	51.1	41.4
LightMem	60.4	87.1	**66.7**	19.6	**71.8**	**57.9**	57.9
MemoryOS	32.0	50.0	20.0	**87.5**	34.6	18.8	13.5
MemPalace	35.4	60.0	20.0	7.1	52.6	37.6	25.6
Mem0	32.8	69.1	26.7	10.7	47.4	28.6	24.1
Qwen3-30B-A3B-Instruct-2507							
MemForest	**79.8**	**97.1**	**76.7**	80.4	75.6	**73.7**	79.7
MemForest (emb)	78.4	94.3	70.0	78.6	76.9	69.9	**81.2**
EverMemOS	66.2	87.9	50.0	60.4	**86.7**	62.1	52.5
LightMem	67.0	94.3	**76.7**	28.6	73.1	64.7	65.4
MemoryOS	50.0	71.4	43.3	**87.5**	55.1	34.6	36.8
MemPalace	44.0	61.4	36.7	12.5	59.0	50.4	34.6
Mem0	40.2	75.7	50.0	10.7	44.9	41.4	27.8

The category breakdown matches the design goal. MemForest is especially strong on categories that require kept evolving evidence across sessions, such as multi-session and temporal-reasoning questions. This supports the claim that MemTrees preserve useful temporal structure while still enabling efficient construction.

The comparison between the two MemForest variants clarifies the role of planner-guided browsing. The planner improves overall accuracy under both model sizes, with the clearest gains on single-session-user, single-session-preference, knowledge-update, and multi-session questions. However, embedding-only browsing is slightly stronger on temporal-reasoning in both settings, suggesting that many temporal queries already align well with the explicit time-ordered structure of MemTree. Thus, planner-guided browsing is useful as a refinement layer, while the temporal memory substrate remains the main quality driver.

Compared with external baselines, EverMemOS is strongest on knowledge-update, LightMem remains competitive on user/preference questions, and MemoryOS is strongest on single-session-assistant. However, none of these baselines is as stable across categories as MemForest. The LongMemEval results therefore support the central claim: MemForest improves write-path efficiency without sacrificing end-to-end answer quality.

5.5 Main Results on LoCoMo

Table 5 reports pass@1 accuracy on LoCoMo, a multi-session dialogue benchmark with more entangled evidence than LongMemEval.

LoCoMo is the harder benchmark for MemForest, but the results remain competitive and balanced. EverMemOS achieves the best overall accuracy, leading MemForest by 4.2 points under 4B and by only 1.2 points under 30B. MemForest is therefore the closest system to the overall best baseline, while substantially outperforming the remaining methods.

The category breakdown shows that MemForest is balanced on most standard categories, while adversarial questions remain a harder case. EverMemOS is stronger on broad multi-hop and temporal questions, while MemForest is strongest on open-ended

Table 5: pass@1 accuracy on LoCoMo.

Method	Overall	Single-Hop	Multi-Hop	Open-Ended	Temporal	Adversarial
Qwen3-4B-Instruct-2507						
MemForest	62.1	72.3	54.8	**54.2**	79.7	29.6
MemForest (emb)	61.4	70.6	55.1	**54.2**	78.7	29.2
EverMemOS	**66.3**	**73.4**	**72.6**	50.0	**86.0**	23.8
LightMem	41.0	44.3	32.7	21.9	60.9	11.7
MemoryOS	48.4	40.1	25.2	27.1	65.6	**42.4**
MemPalace	37.3	46.1	28.3	21.9	51.6	14.6
Mem0	19.3	17.0	9.3	15.6	23.9	20.0
Qwen3-30B-A3B-Instruct-2507						
MemForest	68.4	**78.0**	67.3	**65.6**	83.1	35.9
MemForest (emb)	66.9	75.2	62.3	61.5	84.4	33.2
EverMemOS	**69.6**	76.2	**86.6**	62.5	**88.1**	19.7
LightMem	36.2	34.8	32.7	30.2	50.3	14.1
MemoryOS	50.9	41.1	29.9	40.6	66.5	**44.8**
MemPalace	40.5	47.9	32.4	22.9	56.2	15.9
Mem0	18.6	16.7	9.0	30.2	23.3	15.2

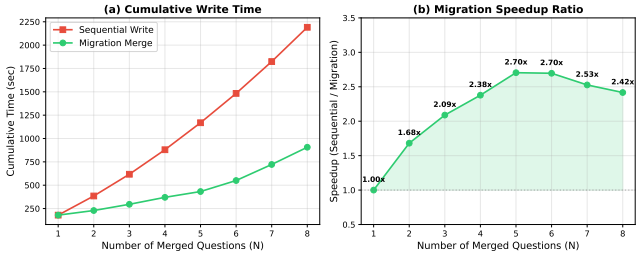

Figure 5: Migration efficiency on progressively merged LongMemEval instances. Left: cumulative maintenance time under sequential write and migration merge. Right: speedup of migration relative to sequential write.

questions under both model sizes and on single-hop questions under 30B. The embedding-only variant remains close to planner-guided MemForest, confirming that most of the gain comes from the maintained memory substrate. Overall, MemForest does not achieve the best score in every category, but it is consistently strong and balanced while retaining the write-path advantage in Section 5.2. This supports our central claim that MemForest improves the efficiency–quality frontier for persistent temporal memory.

5.6 Efficient Migration

Beyond initial construction, a persistent memory system should support maintenance after memory has already been built. We evaluate memory migration, where already materialized memory states are merged directly instead of replaying all raw sessions through the write path.

We build a migration workload from LongMemEval by progressively combining multiple question instances into one memory state. Since instances come from different users, this is a synthetic stress workload that simulates a shared or multi-source memory store. We compare *sequential write*, which reprocesses all sessions into a growing memory, with *migration merge*, which directly merges already materialized MemForest states. This experiment measures memory scale and wall-clock merge cost; it does not replay question answering on the merged state.

Figure 5 shows migration is consistently faster than sequential write. It exceeds **2**× speedup from N=3, peaks at **2.70**× around N=5–6, and remains above **2.4**× at N=8. The gain comes from state reuse: migration avoids repeated extraction over already processed histories and avoids rebuilding unaffected trees from scratch.

The speedup does not come from collapsing the memory state. Sequential write and migration merge produce states of comparable scale: across all merged sizes, the number of facts differs by less than 1%, and the number of trees differs by at most about 8%. Small differences are expected because fact extraction, tree-summary refresh, and scope routing involve LLM-based or heuristic decisions.

Among the evaluated baselines, we did not identify a directly comparable merge interface: they support continued linear writes, but not direct merge across already constructed memory states. Migration is therefore a lifecycle advantage of MemForest in settings where memory must be transferred, synchronized, or combined across instances, such as expert memory transfer [26] and distributed memory construction [10, 14]. Overall, the experiment shows that canonical facts and scoped temporal trees make post-build maintenance reusable and localized, extending the write-path efficiency argument beyond initial construction. Detailed memory-scale statistics are reported in Appendix E.

6 ABLATION STUDY

The main results show that MemForest improves answer quality and write-path efficiency. We now isolate which design choices are responsible for these gains. The ablations follow the two main mechanisms introduced in Sections 3–4. First, we study MemTree write-path scalability, focusing on lazy dirty-path refresh, level-parallel maintenance, and the branching factor k. Second, we study retrieval accuracy on a controlled LongMemEval subset, isolating the role of the three temporal tree views and the difference between embedding-only and LLM-guided tree browsing.

6.1 Write-Path Scalability

We first study the write path, since MemForest's main systems contribution is not merely faster extraction, but the ability to keep long-horizon memory maintenance local as memory grows. The key question is therefore whether MemTree maintenance remains scalable under increasing tree size, and how the MemTree branching factor k, i.e., the maximum number of children summarized by one internal node, affects both efficiency and summary quality.

Lazy maintenance reduces redundant LLM work and improves scalability. Figure 6(a) compares eager per-insert summary regeneration against MemForest's batch mark-dirty refresh. Batch refresh substantially reduces LLM summary calls, and the reduction becomes larger as the number of facts per tree grows. This shows that lazy maintenance is not a small implementation optimization: it prevents repeated writes from triggering redundant summary regeneration along overlapping ancestor paths.

The same locality benefit appears in parallel execution. Figure 6(c) shows that level-parallel flush yields larger speedups for larger trees, because bigger trees expose more same-level nodes that can be refreshed concurrently. In other words, the gain from MemTree maintenance improves with data scale rather than disappearing at larger workloads.

A moderate branching factor k balances efficiency and summary quality. Figures 6(d) and 6(e) study the MemTree branching factor k. Increasing k makes trees shallower and can reduce maintenance

depth, but it also increases the number of children summarized in each call. Per-call summary recall remains high up to about $k \approx 16$, after which it drops more sharply, indicating a soft summary-capacity knee. End-to-end root recall also peaks at moderate k values rather than at the extremes: small k deepens the summarization chain and accumulates cascade loss, while overly large k makes each summary too flat and lossy. These results justify MemForest's moderate branching-factor defaults rather than treating k as an arbitrary tuning knob.

Extraction remains the dominant cost, but MemTree maintenance does not become the bottleneck. Figure 6(b) profiles MemTree build time as the number of facts per tree grows, while Figure 7 gives the per-question write-path breakdown of wall-clock time, token usage, and LLM call counts under both 30B and 4B build backbones. Extraction dominates time and token usage, identifying it as the main optimization target. By contrast, tree building contributes only a modest fraction of wall-clock time, despite issuing many short LLM calls for node summaries. This is an important systems result: MemTree introduces temporal structure and local maintenance without turning the temporal index itself into the new bottleneck.

We defer the chunk-size operating-point study for extraction to Appendix C, since the main focus of the ablation section is the MemTree design itself rather than the extraction preset.

6.2 Retrieval Accuracy on LongMemEval Subset

We study retrieval accuracy on a 60-question LongMemEval subset, constructed by sampling 10 questions from each of the six question types. To reduce the influence of single-sample answer variance, we report pass@8 in the main text and defer the full question IDs to Appendix D. These experiments are retrieval diagnostics rather than replacements for the full-benchmark results in Section 5.

We focus on two questions. First, are multiple temporal tree views needed? Second, after candidate trees have been recalled, how much does tree browsing improve evidence access? We use `llm+planner` to denote the default planner-guided LLM tree browse used by MemForest in Section 5, and `emb` to denote the embedding-only tree descent used by MemForest (emb).

No single tree view is sufficient by itself. Table 6 compares different tree-family combinations under the same `llm+planner` browse setting. Two patterns are clear. First, no single tree family reaches the best performance on its own. Among the single-view variants, `session only` is the strongest at **85.0%** pass@8, while `scene only` drops to **81.7%** and `entity only` falls much further to **63.3%**. This is expected: session trees preserve raw dialogue and therefore provide the highest-fidelity standalone fallback, but they do not by themselves give the most effective structured retrieval view.

Second, combining temporal views is what yields the strongest performance. In particular, `entity+scene` reaches **86.7%**, matching the full `entity+scene+session` system. This result is important because it shows that the core MemTree design is already highly effective even without relying on the raw-dialogue session tree: the structured temporal views alone can recover the best retrieval accuracy. We nevertheless retain the session tree in the full system design, because it preserves an explicit high-fidelity fallback channel and ensures that raw conversational evidence remains available when needed. We therefore view the three tree families as complementary rather than redundant: entity and scene trees carry most of the structured retrieval workload, while session trees remain a deliberate final backstop.

Table 6: Tree-family ablation on the LongMemEval subset. We report 30B pass@8 under `llm+planner` browse.

Variant	30B pass@8
entity+scene+session	**86.7**
entity+scene	**86.7**
entity+session	85.0
session only	85.0
scene+session	83.3
scene only	81.7
entity only	63.3

Planner-guided LLM browse is the strongest high-accuracy mode. We compare retrieval variants on the same subset: flat top-10 fact retrieval without MemTree structure (`flat-10`), root-only recall that uses each tree root as a summary state without hierarchical browse (`root-only`), embedding-only browse (`emb`), planner-guided embedding browse (`emb+planner`), LLM-guided browse (`llm`), and planner-guided LLM browse (`llm+planner`). This comparison complements the query-path design in Section 4.3 and the full-benchmark results in Section 5 by isolating how much the temporal tree structure and browse policy each contribute.

The first result is that neither flat retrieval nor root-only recall is sufficient. `flat-10` reaches **78.3%**, while `root-only` is lower at **73.3%**. Together, these two baselines show that neither directly retrieving a flat top-10 fact set nor stopping at coarse root summaries can match the retrieval power of temporal MemTree browse. The system must both preserve temporal scopes and descend within them to reach answer-bearing evidence.

The second result is that browse itself matters, and that planner-guided LLM browse is the strongest high-accuracy setting. Embedding-only browse already improves substantially over both flat and root-only baselines, reaching **85.0%**, pure LLM-guided browse reaches **88.3%**, and the best result of **90.0%** comes from planner-guided LLM browse. Once candidate trees have been recalled, planner-generated per-tree subqueries make browse more targeted by using each tree's root summary to specialize the search intent. This is especially useful in the LLM-guided setting, where the browser can exploit not only semantic similarity but also more structured intent such as temporal focus, state transitions, or finer-grained disambiguation. The added planner cost is acceptable here, because a single LLM call steers the subsequent per-tree LLM browse calls.

By contrast, planner guidance does not help embedding browse: `emb+planner` drops to **83.3%**. Under embedding-only descent, the rewritten query is reduced to a vector-similarity signal. Unlike LLM-guided branch selection, embedding similarity cannot reliably preserve richer browse intent, such as time ranges, before/after relations, or transition constraints; it mostly captures semantic relatedness. In that setting, the extra planner call adds noticeable cost without a comparably targeted retrieval benefit. For this reason, MemForest exposes two operating modes: a lightweight embedding-only mode for latency-sensitive deployments and a planner-guided LLM mode for the high-accuracy setting.

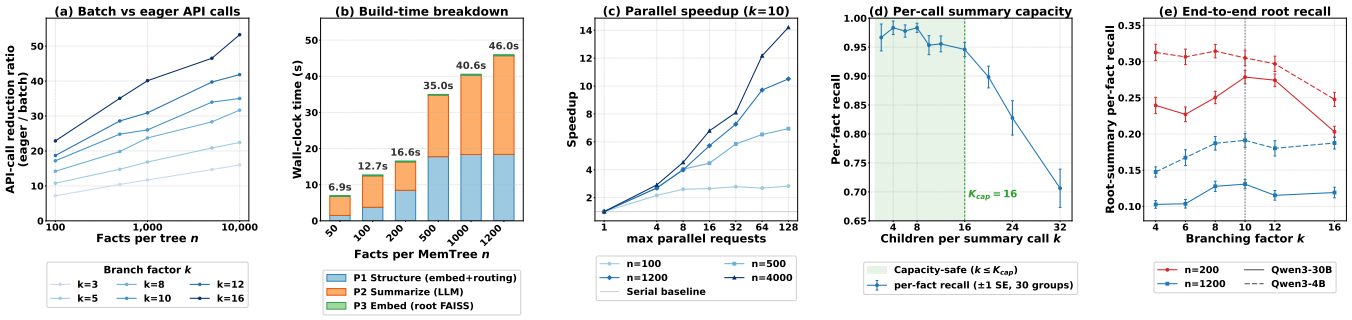

Figure 6: Write-path scalability diagnostics for MemTree. (a) Batch mark-dirty refresh reduces LLM summary calls relative to eager per-insert summary regeneration. (b) MemTree build time grows moderately with the number of facts per tree, showing that tree maintenance remains lightweight. (c) Level-parallel flush yields larger speedups on larger trees, showing improved scalability with data size. (d) Per-call summary capacity remains stable up to a moderate branching factor before dropping. (e) End-to-end root recall peaks at moderate branching factors, motivating the default k settings used in MemForest.

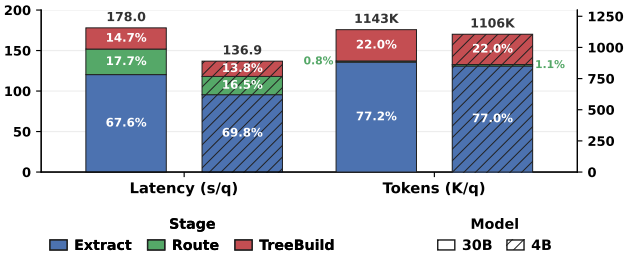

Figure 7: Per-question write-path breakdown for MemForest with Qwen3 30B and 4B. Maintenance cost remains modest because of local updates and parallel construction.

Table 7: Retrieval and browse ablation on the 60-question LongMemEval subset. We report 30B pass@8 only; `flat-10` retrieves the top-10 facts directly without tree recall or browse.

Variant	30B pass@8
flat-10	78.3
root-only	73.3
emb	85.0
emb+planner	83.3
llm	88.3
llm+planner	**90.0**

7 RELATED WORK

Existing memory systems for LLM agents differ mainly in how memory is constructed, how persistent state is organized, and how that state is maintained as interactions accumulate. We review the work most related to MemForest along these three dimensions.

Memory Construction. A common direction is to build long-term memory by extracting information from interactions and storing it as persistent records. MemForest differs in that it treats the construction path as the primary systems target: it parallelizes extraction over short windows, consolidates outputs into canonical facts, and materializes indexes from that canonical state. Representative prior systems take construction as a given pipeline instead. SeCom studies how memory units should be constructed and retrieved for conversational agents [22]. Mem0 emphasizes practical long-term memory for personalization and retrieval [3]. LightMem and MemoryOS introduce staged memory handling across short-, mid-, and long-term components, separating online use from offline consolidation and managed updates [7, 15]. Context-dependent memory frameworks also highlight the value of modular memory handling across different interaction contexts [8]. These systems show the benefit of explicit memory construction over full-history prompting, but mainly treat construction as a pipeline that produces records, profiles, or tiered stores rather than as a resource to be parallelized.

Memory Organization. Another line of work studies how memory should be organized after it is written. MemForest is aligned with this line, but differs in the representation it materializes: it organizes canonical facts into per-scope temporal trees so that retrieval can proceed from root summaries to leaf evidence within an explicit temporal hierarchy, rather than centering the design on recollection workflows or dynamically selected structures. EverMemOS models memory as a lifecycle of semantic consolidation and recollection [11]. A-Mem explores agentic memory organization, while H-Mem, HiMem, and Adapt investigate hybrid, hierarchical, or adaptive structures for long-context agents [13, 17, 33, 35, 37]. These systems move beyond flat memory stores and show that retrieval quality depends strongly on memory structure.

Memory Maintenance. Another important question is how memory should evolve after it is written. MemForest differs from prior work by making a canonical, mergeable memory state with temporal update semantics as its primary design goal: canonical facts are persistent state, summaries are derived artifacts regenerated incrementally, and this separation supports temporal preservation, incremental reorganization, and migration without replaying raw sessions. RMM treats memory as a reflective process, refining what is retained and how stored memory is later used [30]. At the other end of the design space, fidelity-first systems such as MemPalace preserve raw or near-verbatim history and defer abstraction to query time [19]. These approaches make different trade-offs between write-time processing and query-time reasoning, but neither centers on canonical, locally maintainable persistent state. This systems view is related in spirit to write-optimized and temporal indexing, which avoid repeated hot-state rewrites and keep historical versions queryable [2, 6, 20]. MemForest adapts this intuition

to LLM-derived memory, where both facts and summaries must remain incrementally maintainable.

8 CONCLUSION

We presented MemForest, a long-context agent memory system that reframes persistent memory as a write-efficient temporal data management problem. Our key contribution is to address the shared write-path bottleneck by introducing canonical facts as stable write units, decoupling persistent state from derived artifacts, and organizing memory as MemTree, a scoped temporal index that enables localized updates instead of global maintenance. This design shifts post-extraction maintenance away from full-memory rewrites toward affected scopes, dirty paths, and distinct dirty nodes, while preserving explicit temporal structure for current, historical, and transition queries. Experiments on LongMemEval-S show that MemForest achieves the best overall accuracy (79.8%) among stateful baselines while significantly reducing write-path latency, and on LoCoMo it performs strongly on open-ended and temporally grounded queries while remaining competitive on other categories. Overall, our results highlight that scalable long-context memory systems should be designed around write efficiency, explicit temporal preservation, and localized maintainability under continuous updates.

REFERENCES

[1] Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. 2025. Cache-Craft: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation. *Proc. ACM Manag. Data* 3, 3, Article 136 (June 2025), 28 pages. https://doi.org/10.1145/3725273

[2] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An asymptotically optimal multiversion B-tree. *The VLDB Journal* 5, 4 (1996), 264–275.

[3] 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).

[4] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*.

[5] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[6] Ramez Elmasri, Gene TJ Wuu, and Yeong-Joon Kim. 1990. The time index: An access structure for temporal data. In *Proceedings of the 16th International Conference on Very Large Data Bases*. 1–12.

[7] Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. 2026. LightMem: Lightweight and Efficient Memory-Augmented Generation. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=dyJ0GWpjJB

[8] Pengyu Gao, Jinming Zhao, Xinyue Chen, and Long Yilin. 2025. An efficient context-dependent memory framework for llm-centric agents. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: Industry Track)*. 1055–1069.

[9] Yubin Ge, Salvatore Romeo, Jason Cai, Raphael Shu, Yassine Benajiba, Monica Sunkara, and Yi Zhang. 2025. Tremu: Towards neuro-symbolic temporal reasoning for llm-agents with memory in multi-session dialogues. In *Findings of the Association for Computational Linguistics: ACL 2025*. 18974–18988.

[10] Tooraj Helmi. 2025. Decentralizing AI Memory: SHIMI, a Semantic Hierarchical Memory Index for Scalable Agent Reasoning. *arXiv preprint arXiv:2504.06135* (2025).

[11] Chuanrui Hu, Xingze Gao, Zuyi Zhou, Dannong Xu, Yi Bai, Xintong Li, Hui Zhang, Tong Li, Chong Zhang, Lidong Bing, et al. 2026. EverMemOS: A Self-Organizing Memory Operating System for Structured Long-Horizon Reasoning. *arXiv preprint arXiv:2601.02163* (2026).

[12] Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, and Beng Chin Ooi. 2025. HAKES: Scalable Vector Database for Embedding Search Service. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3049–3062.

[13] Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. 2025. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 32779–32798.

[14] Paul Jackson and Jane Klobas. 2008. Transactive memory systems in organizations: Implications for knowledge directories. *Decision support systems* 44, 2 (2008), 409–424.

[15] Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. 2025. Memory os of ai agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 25972–25981.

[16] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556* (2025).

[17] Mingfei Lu, Mengjia Wu, Feng Liu, Jiawei Xu, Weikai Li, Haoyang Wang, Zhengdong Hu, Ying Ding, Yizhou Sun, Jie Lu, et al. 2026. Choosing How to Remember: Adaptive Memory Structures for LLM Agents. *arXiv preprint arXiv:2602.14038* (2026).

[18] 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 *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 13851–13870.

[19] milla-jovovich. 2026. *MemPalace*. https://github.com/milla-jovovich/mempalace

[20] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta informatica* 33, 4 (1996), 351–385.

[21] Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. MemGPT: towards LLMs as operating systems. (2023).

[22] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Xufang Luo, Hao Cheng, Dongsheng Li, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Jianfeng Gao. 2025. SeCom: On Memory Construction and Retrieval for Personalized Conversational Agents. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=xKDZAW0He3

[23] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, et al. 2024. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*. 963–981.

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

[25] 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).

[26] Alireza Rezazadeh, Zichao Li, Ange Lou, Yuying Zhao, Wei Wei, and Yujia Bao. 2025. Collaborative memory: Multi-user memory sharing in llm agents with dynamic access control. *arXiv preprint arXiv:2505.18279* (2025).

[27] Alireza Rezazadeh, Zichao Li, Wei Wei, and Yujia Bao. 2025. From Isolated Conversations to Hierarchical Schemas: Dynamic Tree Memory Representation for LLMs. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=moXtEmCleY

[28] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D Manning. 2024. Raptor: Recursive abstractive processing for tree-organized retrieval. In *The Twelfth International Conference on Learning Representations*.

[29] Ji Sun, Guoliang Li, James Pan, Jiang Wang, Yongqing Xie, Ruicheng Liu, and Wen Nie. 2025. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4951–4963.

[30] Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, et al. 2025. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 8416–8439.

[31] Zhenheng Tang, Xin He, Tiancheng Zhao, Fanjunduo Wei, Xiang Liu, Peijie Dong, Qian Wang, Qi Li, Huacan Wang, Ronghao Chen, et al. 2026. LLM Agent Memory: A Survey from a Unified Representation–Management Perspective. (2026).

[32] 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 *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=pZiyCaVuti

[33] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-Mem: Agentic Memory for LLM Agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=FiM0M8gcct

[34] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical

report. *arXiv preprint arXiv:2505.09388* (2025).

[35] Zihe Ye, Jingyuan Huang, Weixin Chen, and Yongfeng Zhang. 2026. H-Mem: Hybrid Multi-Dimensional Memory Management for Long-Context Conversational Agents. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7756–7775.

[36] Runjie Yu, Weizhou Huang, Shuhan Bai, Jian Zhou, and Fei Wu. 2025. AquaPipe: A Quality-Aware Pipeline for Knowledge Retrieval and Large Language Models. *Proc. ACM Manag. Data* 3, 1, Article 11 (Feb. 2025), 26 pages. https://doi.org/10.1145/3709661

[37] Ningning Zhang, Xingxing Yang, Zhizhong Tan, Weiping Deng, and Wenyong Wang. 2026. HiMem: Hierarchical Long-Term Memory for LLM Long-Horizon Agents. *arXiv preprint arXiv:2601.06377* (2026).

[38] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176* (2025).

[39] Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2025. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems* 43, 6 (2025), 1–47.

[40] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 38. 19724–19731.

A PROMPTS

A.1 LLM-as-Judge Prompts

LongMemEval Judge Prompt

Your task is to label an answer to a LongMemEval question as CORRECT or WRONG.
You will be given:
(1) question type
(2) question
(3) gold answer
(4) generated answer
Be generous with grading: (1) accept semantically equivalent answers; (2) accept equivalent relative and absolute time expressions; (3) accept more specific but consistent answers; (4) mark WRONG only for contradiction, missing key facts, answering a different question, or incorrect abstention.
Question type: {question_type}
Question: {question}
Gold answer: {gold_answer}
Generated answer: {generated_answer}
Return JSON only with {"label": "CORRECT"} or {"label": "WRONG"}.

LoCoMo Judge Prompt

Your task is to label an answer to a question as CORRECT or WRONG.
You will be given:
(1) question
(2) gold answer
(3) generated answer
Be generous with grading: (1) accept semantically equivalent answers; (2) accept equivalent relative and absolute time expressions; (3) accept more specific but consistent answers; (4) mark WRONG only for contradiction, missing key facts, non-answer, or incorrect abstention.
Question: {question}
Gold answer: {gold_answer}
Generated answer: {generated_answer}
Return JSON only with {"label": "CORRECT"} or {"label": "WRONG"}.

B DETAILED WRITE-PATH AND PARALLELISM ANALYSIS

This appendix expands Table 1. We use the same common write setting as Section 2.3: the touched memory object contains N existing records or state items, and the incoming session produces M new memory records. We treat the number of retrieved candidates per new record as a constant. The main text reports the dominant write critical path, while this appendix explains which parts of each pipeline are parallelizable and which parts remain on the dependency chain.

For MemForest, the table reports dependency depth rather than total touched work. Inserting M records into affected MemTrees touches $O(M \log N)$ nodes in total, but insertions across records, scopes, and same-level dirty nodes can be processed in parallel. The critical path is therefore bounded by the tree height, $O(\log N)$.

The main text reports the per-record critical path. Here we explain why each system falls into that class and separate total parallelizable work from dependency depth.

B.1 Independent Records and Mutable States

A temporal scope is commonly represented in one of two static forms. The first stores different time points as independent records:

$$e_{\sigma,j} \mapsto v_{\sigma,j}, \quad j = 1, \ldots, m_\sigma, \tag{8}$$

where $v_{\sigma,j}$ is an embedding or retrievable representation of evidence item $e_{\sigma,j}$. Retrieval can be parallelized, but semantic similarity does not encode predecessor, successor, or interval relations. A temporal transition query may therefore retrieve a record from the wrong time point.

The second form maintains one mutable state object:

$$s_\sigma^{(i)} = \textsc{LLMUpdate}(s_\sigma^{(i-1)}, \Delta_\sigma^{(i)}). \tag{9}$$

If the object grows with history, the i-th update may require processing the current state:

$$O\left(|s_\sigma^{(i-1)}| + |\Delta_\sigma^{(i)}|\right). \tag{10}$$

For a hot scope, this can make each triggered update proportional to the accumulated state size N. If the state is compressed to bound this cost, intermediate states and transition evidence may be lost.

B.2 Mem0

Mem0 maintains mutable memory records with embedding-based retrieval. For one new memory record r, the write path can be abstracted as

$$R = \textsc{Search}(r, K), \quad a = \textsc{LLMUpdate}(r, R), \quad S' = \textsc{Mutate}(S, a). \tag{11}$$

The per-record candidate set has size K, giving a per-record critical comparison path of $O(K)$. Embedding and search work can be parallelized across records, but the update action is generated against the current mutable memory state. If two new records retrieve the same old record, reordering the update calls can change whether the old record is added, merged, rewritten, or deleted. Thus, the candidate work is parallelizable, but the maintenance semantics remain state-dependent.

B.3 MemoryOS

MemoryOS maintains short-term, mid-term, and long-term memory states. The relevant dependency can be summarized as

$$\begin{aligned} Q' &= \textsc{AppendQueue}(Q, r), \\ P &= \textsc{PageUpdate}(Q'), \\ L' &= \textsc{ProfileUpdate}(L, P). \end{aligned} \tag{12}$$

Queue promotion, page continuity, heat state, and profile rewriting are ordered state updates. When a profile or summary is hot, updating it may require reading and rewriting the accumulated text state. In the worst case, the touched state has size N, giving a per-record triggered maintenance path of $O(N)$. Compression can reduce the state size, but does so by discarding detail from older evidence.

B.4 EverMemOS

EverMemOS constructs memory through streaming MemCell formation. For a new turn or record r_i, boundary detection can be written as

$$b_i = \textsc{Boundary}(H_{i-1}, r_i), \quad H_i = \textsc{Advance}(H_{i-1}, r_i, b_i), \tag{13}$$

where H_i is the unresolved history after processing r_i. Each step is $O(1)$ with respect to the number of existing memory records, but it is an ordered stream step: later boundaries cannot be computed without earlier boundary decisions. Post-boundary extraction and index construction can be parallelized after MemCells are available, but boundary formation remains sequential within one conversation.

B.5 LightMem

LightMem uses segmentation, buffer accumulation, extraction triggers, and optional consolidation. The online path can be summarized as

$$z_i = \textsc{BufferUpdate}(z_{i-1}, r_i), \quad E_i = \textsc{ExtractIfTriggered}(z_i). \tag{14}$$

The buffer update itself is an ordered step. When offline consolidation is triggered, the system builds candidate relationships from a global memory snapshot. A new or updated record may need to be compared against the existing memory pool, giving a triggered per-record path of $O(N)$ under the common notation. Worker threads can parallelize independent LLM calls, but the consolidation phase is not a deterministic scope-local update because it depends on shared candidate queues and shared entry mutations.

B.6 MemPalace

MemPalace follows an append-oriented raw-history path:

$$c = \textsc{Chunk}(r), \quad S' = \textsc{Append}(S, c). \tag{15}$$

For one new chunk or memory record, the write path is $O(1)$ with respect to existing memory size. This is highly parallelizable because there is no write-time semantic maintenance. The trade-off is that temporal transitions, contradiction handling, and cross-chunk composition are not maintained as structured state and must be handled at query time.

B.7 MemForest

MemForest represents each temporal scope as a MemTree. For one new canonical fact r, the write path is

$$\sigma = \textsc{Route}(r), \quad T'_\sigma = \textsc{Insert}(T_\sigma, r), \quad A_\sigma = \textsc{RefreshDirty}(T'_\sigma). \tag{16}$$

For a balanced k-ary MemTree with N records in the affected scope, the tree height is $h = \lceil \log_k N \rceil$. Inserting one record touches one leaf-to-root path:

$$O(h) = O(\log N). \tag{17}$$

For B new records, the total touched structural work is $O(B \log N)$. Dirty nodes in different scopes and dirty nodes at the same level can be refreshed in parallel after their children are available, so the dependency depth remains bounded by the tree height rather than by the size of a mutable profile or a global memory object.

B.8 Summary

Under a per-record view, the main contrast is where prior state appears on the write critical path. Memory-record systems such as Mem0 consult K retrieved candidates and then perform state-dependent update adjudication. Profile or summary systems can degenerate to $O(N)$ updates when hot states are repeatedly rewritten. Streaming segmentation systems may have $O(1)$ per-record steps, but those steps are ordered and cannot be freely parallelized within one stream. Raw-history systems have $O(1)$ append cost, but avoid structured temporal maintenance. MemForest instead bounds the per-record maintenance path by $O(\log N)$ through local MemTree insertion while preserving time-local evidence and interval summaries.

Table 8: Chunk-sweep study for raw-fact extraction on the assembled-long-session benchmark.

Preset	Ent-GR (%)	Facts/s	Token/fact
1-turn	100	7.40	868
2-turn	100	7.00	767
4-turn	100	1.90	811
8-turn	99	1.52	647
16-turn	94	1.37	682
32-turn	95	1.26	832
48-turn	88	0.62	1127
64-turn	72	0.46	1271
whole	55	0.52	2505

C CHUNK-SIZE DIAGNOSTIC FOR RAW-FACT EXTRACTION

We study extraction chunk size in a separate diagnostic setting, since this operating-point choice is not the main contribution of MemForest and is therefore deferred from the main ablation section. The purpose of this experiment is to examine how raw-fact extraction degrades as chunk granularity increases.

To create a controlled stress setting, we manually assemble long sessions by concatenating multiple original conversations, then run the same raw-fact extraction pipeline under different chunk presets. This setup is therefore a diagnostic stress test rather than the benchmark's native dialogue layout, and is intended to expose how the extraction pipeline behaves as chunk granularity is varied.

We evaluate each setting using **Ent-GR** (Entity Gold-Range retention), which is a diagnostic retention metric rather than an end-to-end QA metric. For each question, we identify its gold supporting turn range and the key answer-bearing span within that range, such as an entity, date, number, or short attribute phrase. A question is counted as retained if at least one extracted fact produced from a chunk intersecting the gold range still preserves that key span after normalization.

Table 8 shows a clear constrained operating point. Whole-session extraction is both the least faithful and one of the least efficient settings, while chunk sizes beyond 8 turns show visible fidelity degradation. Very small chunks preserve answer-bearing information, but 2-turn extraction provides the best overall balance: it preserves full Ent-GR, remains near the best throughput regime, and improves token efficiency over 1-turn extraction. We therefore use 2-turn extraction as the default write-path setting in the main system.

D LONGMEMEVAL DIAGNOSTIC SUBSET FOR ABLATION

For the retrieval ablations in Section 6.2, we construct a balanced 60-question diagnostic subset by sampling 10 questions from each LongMemEval question type. Table 9 lists the full question IDs.

Table 9: Question IDs in the 60-question LongMemEval diagnostic subset (10 per type).

Question Type	Question IDs
Knowledge Update	01493427, 031748ae, 0977f2af, 18bc8abd, 1cea1afa, 4d6b87c8, 69fee5aa, 852ce960, a1eacc2a, f9e8c073
Multi-Session	1a8a66a6, 60bf93ed_abs, 73d42213, 81507db6, 88432d0a_abs, aae3761f, d682f1a2, e5ba910e_abs, gpt4_d84a3211, gpt4_f2262a51
Single-Session (Assistant)	0e5e2d1a, 1568498a, 16c90bf4, 561fabcd, 6222b6eb, 7161e7e2, 71a3fd6b, 7a8d0b71, 8cf51dda, c7cf7dfd
Single-Session (Preference)	06f04340, 0edc2aef, 195a1a1b, 1a1907b4, 1d4e3b97, 35a27287, 6b7dfb22, 75832dbd, a89d7624, caf03d32
Single-Session (User)	001be529, 311778f1, 545bd2b5, 5d3d2817, 60d45044, 94f70d80, af8d2e46, c5e8278d, caf9ead2, faba32e5
Temporal Reasoning	4dfccbf7, 982b5123, e4e14d04, gpt4_18c2b244, gpt4_1e4a8aec, gpt4_2487a7cb, gpt4_68e94287, gpt4_7a0daae1, gpt4_e072b769, gpt4_f420262d

E DETAILED MEMORY SCALE IN MIGRATION

Table 10 reports the memory scale produced by sequential write and migration merge in the migration experiment of Section 5.6. The purpose of this analysis is to verify that the observed migration speedup does not come from collapsing or discarding memory state. Both strategies produce memory states of comparable scale. Across all merged sizes, the number of facts differs by less than 1%, and the number of trees differs by at most about 8%.

Small differences are expected. The sequential-write baseline replays the sessions through the full write path, while migration merge reconciles already materialized states. Because fact extraction, tree-summary refresh, and scope routing involve LLM-based or heuristic decisions, the two procedures need not produce bit-identical forests. The key observation is that migration preserves a similar amount of persistent evidence and scoped tree structure while reducing maintenance time.

Table 10: Memory scale under sequential write and migration merge. The two strategies produce memory states of comparable scale, without evidence of systematic blow-up or collapse.

N	Seq Facts	Mig Facts	Seq Trees	Mig Trees
1	1,435	1,435	249	249
2	2,692	2,716	375	405
3	4,061	4,098	562	607
4	5,540	5,590	733	792
5	6,860	6,922	895	967
6	8,101	8,174	1,044	1,128
7	9,426	9,511	1,200	1,296
8	10,705	10,801	1,362	1,472

F DETAILED RESULT ON ACCURACY

We report full pass@k curves for $k = 1, \ldots, 8$ on both LongMemEval and LoCoMo. The main paper uses pass@1 as the primary metric because it best matches the default single-run deployment setting. The full curves are included here to show how each system behaves under larger decoding budgets.

On LongMemEval, MemForest remains the strongest system across the full pass@k range under both the 4B and 30B settings. The overall ranking is stable as k increases, and the same pattern largely holds for the main long-context categories, especially *multi-session* and *temporal-reasoning*. EverMemOS shows a steeper increase than several other baselines on some categories, especially under the smaller 4B answerer, which is consistent with the interpretation in the main text: its retrieved contexts often contain useful evidence, but that evidence is not always exploited reliably in a single attempt. Increasing the decoding budget recovers part of this missed evidence, but does not change the benchmark-level ranking.

On LoCoMo, the pattern is more nuanced. MemForest and EverMemOS remain close in overall accuracy across the full pass@k range, with EverMemOS generally staying slightly ahead at the benchmark level. Category-wise, MemForest is usually the second-best system on most question types, while EverMemOS is typically the strongest. The main exception is *adversarial*, where the ranking differs from the other categories. Under the 30B setting, *adversarial* is led by MemoryOS, with MemForest remaining second and EverMemOS behind. Under the 4B setting, the same ordering holds at smaller k, but EverMemOS overtakes MemForest as k increases. This indicates that additional decoding budget benefits EverMemOS more strongly on adversarial cases than on the benchmark overall.

Overall, the pass@k curves support the use of pass@1 as the main metric while clarifying how the systems respond to larger

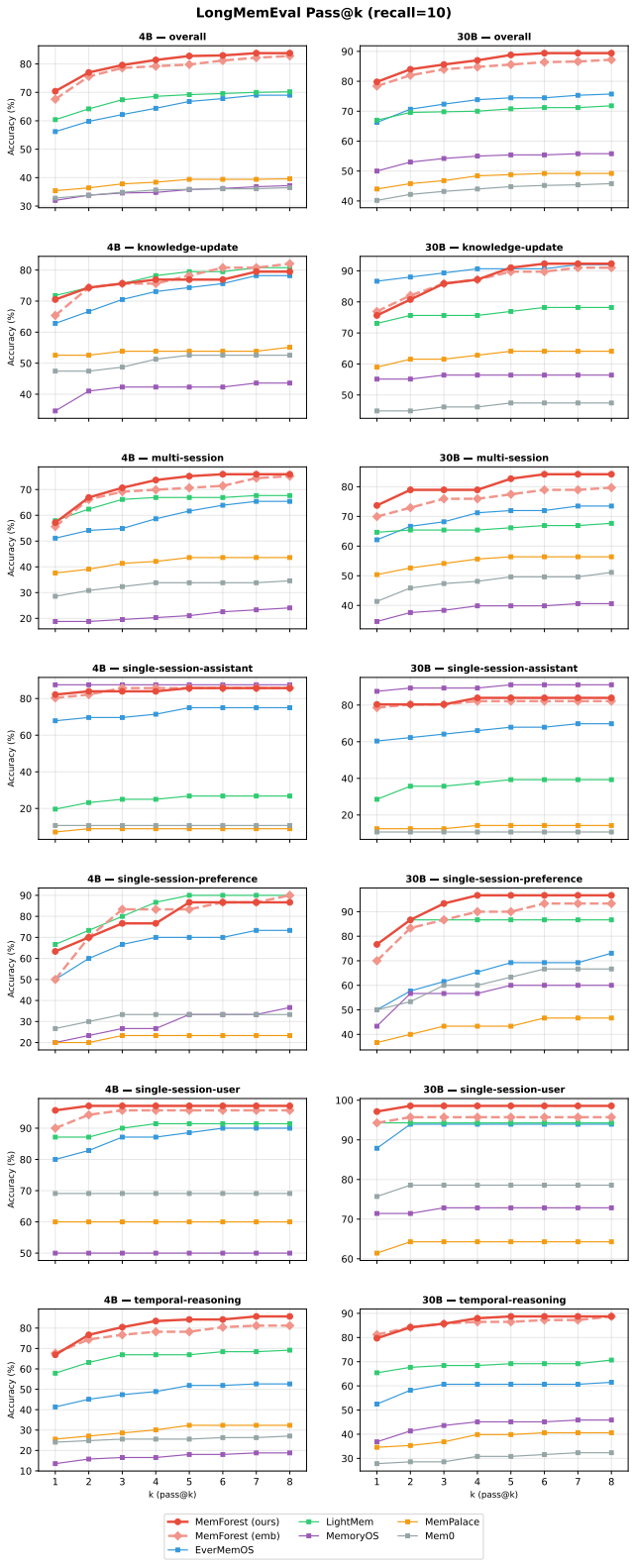

Figure 8: LongMemEval pass@k curves for $k = 1, \ldots, 8$ under the shared recall budget of top-10.

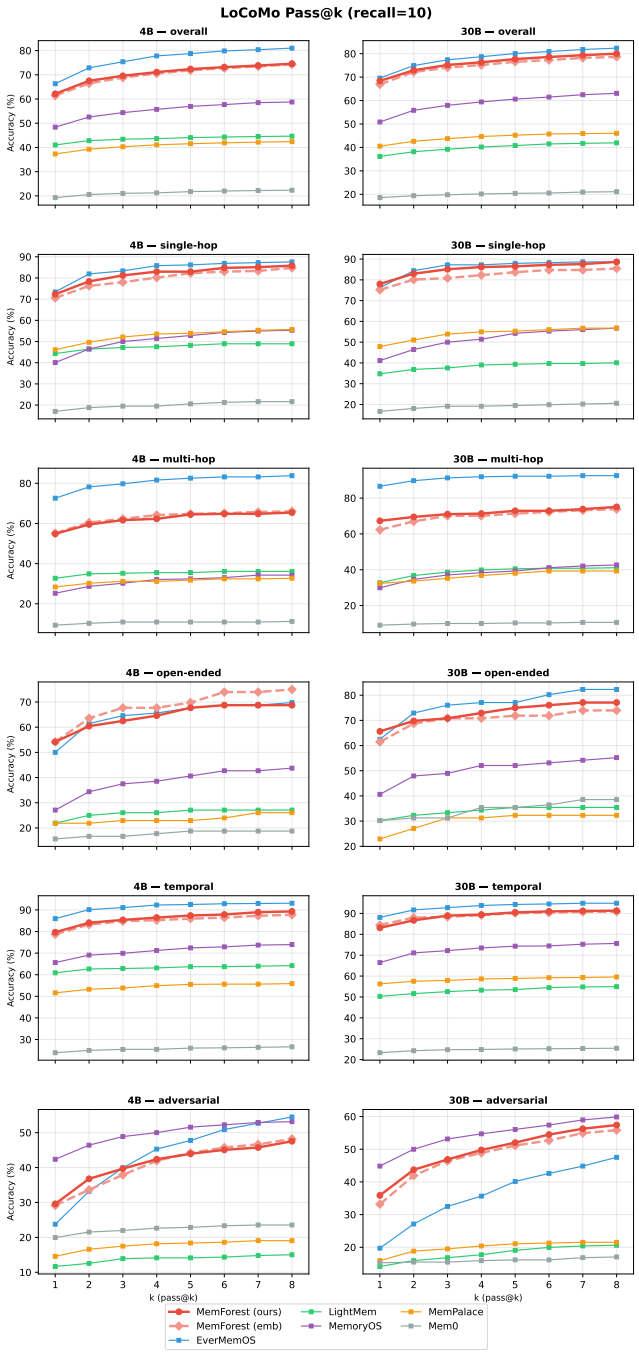

Figure 9: LoCoMo pass@k curves for $k = 1, \ldots, 8$ under the shared recall budget of top-10.

decoding budgets. On LongMemEval, MemForest remains consistently strongest across the full pass@k range. On LoCoMo, the benchmark-level ordering is closer, with EverMemOS generally leading and MemForest usually remaining a strong second-best system across most categories.