Total Recall: Pointer-Based Compaction and Task-Conditioned Retrieval for Persistent LLM Agents

Oscar Serra (with AI assistance)

June 2026

Abstract

Persistent LLM agents must preserve precise strings, causal chains, and decision rationales across sessions that routinely exceed context limits. The industry-standard approach — narrative compaction — replaces this high-resolution state with lossy prose summaries, creating an irrecoverable "only copy" failure. We present **Total Recall**, a lossless, event-sourced memory architecture that treats the context window as a managed cache over a durable store. Instead of summarizing, Total Recall evicts history via **pointer-based compaction** — inserting compact time-range markers with topic hints and retrieval directives — while making all evicted content recoverable through a `recall(query)` tool. Retrieval is **task-conditioned**: injection depends on the active task and expected future needs, not just embedding similarity. To keep total memory bounded along both axes — the in-context cache *and* the long-term store — Total Recall adds a **write-reconciliation** step that classifies each candidate memory as ADD, UPDATE, DELETE, or NONE, so the working set stays bounded while the audit log stays append-only and complete. The **TRACE** production implementation validates these claims. Its full automated test suite passes with zero failures, and controlled benchmarks show **100% needle-in-haystack recall** under forced compaction versus 0% for truncation, and **94% exact-match recall at 2-hop** versus 4% for narrative compaction and 36% for MemGPT-style paging. Compaction latency is sub-linear, on the order of milliseconds, and months of production operation indexing over 14,000 real emails recorded zero lossless-invariant violations.

Architecture Overview: Storage Within a Memory Loop

Total Recall is a storage and compaction substrate. It is self-contained, but it is designed to sit inside a larger loop, and the loop matters because two adjacent mechanisms — an index built over the stored pointers, and a nightly process that tunes the storage thresholds — turn durable storage into queryable, self-improving memory. We describe those two mechanisms here only to the extent needed to understand the interfaces Total Recall exposes; their full design is external to this paper.

- **Storage and compaction.** A lossless, append-only event store treated as ground truth, with the context window managed as a cache over it. When context fills, history is evicted

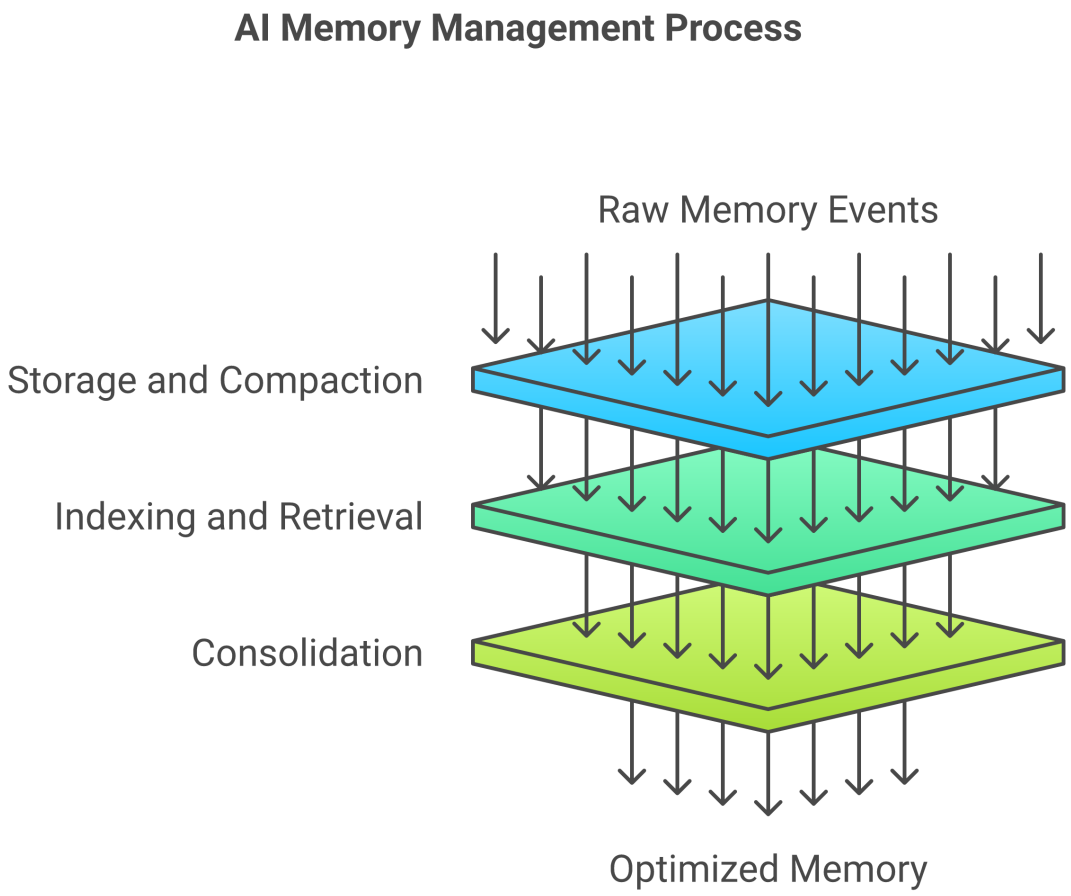

Figure 1. How storage sits inside a memory loop: Total Recall (storage and compaction) emits topic-hinted pointers that an indexing layer turns into a navigable retrieval index, while a nightly consolidation process reflects on retrieval outcomes and tunes the storage thresholds.

via topic-hinted pointers rather than summarized, and writes are reconciled so the working set stays bounded. This is the subject of the present paper.
- **Indexing and retrieval.** A navigable index built *over* the compacted pointers — clustering them by topic, scoring them by importance, and serving sub-second lookup. The storage layer's topic-hinted markers are the structured seams this index grips; without them, retrieval degrades to generic full-text search over raw logs. Total Recall's contribution to this layer is the pointer format; the index itself is out of scope.
- **Consolidation and self-improvement.** A nightly reflection process reviews the day's retrieval successes and failures, builds episodic summaries, and feeds corrections back — adjusting compaction thresholds and running the deferred write-reconciliation sweep (§8.3). Nightly is the only safe moment to decide a fact is stale, because that requires knowing its downstream outcome. Total Recall supplies the deferred sweep; the reflection policy that drives it is out of scope.

The result is a closed loop: storage produces structured pointers, indexing makes them queryable, and consolidation observes how well retrieval worked and re-tunes storage. The rest of this paper specifies storage and compaction; the two adjacent mechanisms appear only where they touch its interfaces.

1. Introduction — The Compaction Failure Mode

As LLMs evolve from session-scoped chatbots to persistent agents, the memory problem changes qualitatively. The agent must preserve high-resolution task state, long-horizon commitments, tool-grounded evidence, and continuity under context resets — all within a hard token budget. The industry's answer is narrative compaction: summarize old context to free space. This paper demonstrates that narrative compaction is a category error for agentic workloads, presents a principled alternative, and validates it through controlled benchmarks and production deployment.

The biology offers a useful frame. In the mammalian brain, an *engram* is the physical trace a memory leaves in neural tissue — the durable change in synaptic connections that makes later recall possible. The hippocampus acts as the brain's librarian, deciding during sleep which fleeting experiences get encoded into long-term storage and which are discarded. Total Recall is an attempt to give an AI agent what biology gives every mammal: a durable, compactable memory trace that survives beyond the current conversation, and a principled rule for what to keep.

1.1 Why Narrative Compaction Fails

Narrative compaction — reducing conversation history to an LLM-generated summary — is the default solution in virtually every deployed agent framework. It appears to work: context pressure is relieved, the model continues operating, and summaries preserve the session's narrative arc. For tool-using, long-horizon agents, this masks a structural failure:

1. **Irreversibility.** Once details are omitted from the summary, no reliable recovery mechanism exists. The summary becomes the *only cognitively accessible copy* of that information.
2. **Compression at maximum load.** Compaction fires near context saturation, precisely when long-context failure modes are most severe.

In production, long-running sessions accumulate massive tool results and repeatedly trigger compaction. The result looks like "forgetfulness" but is actually **cache eviction without storage**. Our system confirmed this directly: sessions over email archives, codebases, and

multi-day task threads consistently lost the exact hashes, file paths, and error strings needed for task completion under narrative compaction.

1.2 Taxonomy of Information Loss

Narrative compaction predictably destroys four categories of information that determine task success:

Category	Example	Why summaries lose it
Precise strings	SHA hashes, file paths, email addresses	Summaries paraphrase; exact characters are discarded
Causal chains	"X failed because Y was misconfigured after Z"	Multi-step causation collapses to "X failed"
Negative knowledge	"We tried approach A and it didn't work"	Summaries bias toward positive outcomes
Temporal anchors	"At 14:32, the deploy succeeded"	Timestamps are omitted as irrelevant detail

A summary preserves the *shape* of a task while destroying its operational substance.

1.3 Thesis and Contributions

Thesis. The context window must be treated as a **cache**, not canonical memory. Compaction must be **cache eviction with pointers**, not narrative consolidation. A second corollary follows: a memory system that only ever grows is not bounded, so *writes* must be reconciled, not blindly appended.

Total Recall operationalizes this through:

- A **lossless append-only event store** as ground truth.
- **Pointer-based compaction**: compact time-range markers replace evicted content.
- A **hybrid push/pull retrieval model** balancing proactive injection with on-demand recall.
- **Task-conditioned retrieval priority** derived from expected-utility optimization.
- **Write reconciliation** (ADD/UPDATE/DELETE/NONE) that bounds the working set without mutating the audit log.

The **TRACE** production codebase (§9) implements and validates all claims. Code, synthetic generators, and evaluation harness will be released on GitHub (link redacted for anonymity).

2. Background & Related Work

2.1 Context Management & Paging

MemGPT (Packer et al., 2023) frames the context window as RAM, with the model managing paging via tools. Focus (Verma, 2024, pre-print) demonstrates agents self-managing context compression. ACC (Bousetouane, 2024) maintains a bounded internal state continuously updated by the agent. RetMem (Chen et al., 2023) and LongAgent (Dai et al., 2023) highlight retrieval mechanisms for extremely long horizons. Total Recall differs by treating compression as a *system-level cache eviction*, avoiding the attention cost of self-management while preserving pull capabilities.

2.2 Agent Memory Structures

Park et al. (2023) introduced a memory stream combining append-only logs with reflection. Reflexion (Shinn et al., 2023) and Voyager (Wang et al., 2023) maintain persistent memory via external text/code bases. Structural Memory (Zeng et al., 2024) mixes episodic, semantic, and procedural memories. Mem0 (Yu et al., 2024) introduces a reconciliation step that classifies each candidate memory write as add, update, or delete rather than appending unconditionally. Total Recall shares the append-only event store, formalizes pointer-based compaction as an eviction protocol with task-conditioned retrieval, and adapts Mem0-style reconciliation as a *logical* working-set bound layered over an immutable audit log (§8).

2.3 Caching & Graph Retrieval

Total Recall's eviction policy connects to classical caching algorithms: LRU-K (O'Neil et al., 1993) and LIRS (Jiang & Zhang, 2002) balance recency and frequency; Belady's MIN (Belady, 1966) requires an oracle. Total Recall uses Task State as a noisy oracle for future access. For derived indexing, GraphRAG (Edge et al., 2024) and RAPTOR (Sarthi et al., 2024) construct hierarchies; Total Recall adopts these as *optional indexes* rather than ground-truth replacements.

2.4 The Mechanisms That Surround Storage

Storage and compaction are useful on their own, but three adjacent mechanisms determine how much value the stored content yields in practice. We state their interfaces concretely because each one depends on a property Total Recall guarantees, and naming that dependency clarifies what the storage layer must produce. The mechanisms themselves are external to this paper.

- **The index over pointers.** Pointer-based compaction is not just space-saving; it is what makes downstream retrieval tractable. Each marker carries topic hints and a time range — structured anchors, not prose. A retrieval system can build a navigable index over these compacted pointers: clustering them by topic, scoring them by importance, and enabling sub-second lookup across the full corpus. *Without* the topic-hinted pointers, that index has nothing to grip and retrieval degrades to generic full-text search over raw, undifferentiated logs. The pointers are the seams.
- **The nightly reflection process.** A consolidation process running once per day reviews the day's retrieval successes and failures, identifies patterns (for example, "medical context is consistently under-retrieved"), and adjusts compaction and importance thresholds — in effect teaching storage which memories deserve preservation. This is also where the deferred reconciliation sweep runs (§8.3): the only safe moment to decide that a fact is now stale is after its downstream outcome is known.
- **Persona-aware assembly.** Continuity across resets requires that certain blocks — the system prompt and persona state — ride alongside the push pack and are never evicted. Total Recall guarantees this by classifying those kinds as non-evictable (§5.1); the policy that selects which persona blocks to include is decided upstream of storage.

3. Total Recall Architecture

Total Recall is built on six principles: (1) lossless event persistence; (2) context as cache; (3) pointer-based compaction; (4) asynchronous indexing; (5) bounded push/pull context assembly; (6) prompt-cache-friendly ordering.

3.1 Data Model

1. **Events.** Append-only user messages, tool calls, and results.
2. **Artifacts.** Large blobs with extracted semantic preview windows.
3. **Task State.** A compact index-card view of the current task (goals, open loops, `key_events`).
4. **Time-range markers.** Tuples $\mu = (t_{\text{start}}, t_{\text{end}}, \mathcal{K}, d, \ell)$ denoting evicted spans with key topics $\mathcal{K}$. Adjacent markers merge hierarchically to bound token overhead.

3.2 Pipeline

- **Layer 1 (Ingest & Index).** Persists events synchronously. Vectors embed asynchronously; the hot tail covers the latency gap.
- **Layer 2 (Retrieve & Pack).** Assembles a fixed-budget *Push Pack* (Task State, markers, hot tail) and handles on-demand *Pull* via `recall`.
- **Layer 3 (Consolidate).** Builds episodic summaries as derived indexes over the lossless store, and runs the deferred write-reconciliation sweep (§8.3).

4. Task-Conditioned Retrieval Priority

A purely similarity-driven retriever over-injects obsolete objectives and debug noise. Retrieval must be conditioned on active tasks and premises.

4.1 Formal Derivation

We seek a context pack $C^* \subseteq \mathcal{E}$ under budget B maximizing expected task completion:

$$C^* = \arg \max_{C \subseteq \mathcal{E},\, |C| \leq B} \mathbb{E}\left[P(\text{complete} \mid C, \tau)\right]$$

This objective is combinatorially hard: subset selection under a budget constraint is NP-hard in the general case. We approximate it by assuming marginal contributions are roughly independent — a submodularity assumption that lets a greedy selector run in near-linear time. The assumption breaks when events are strongly complementary: a cryptographic key split across two events, or a bug symptom paired with the configuration change that caused it. The greedy approximation works well for the common case where most events contribute independently; co-retrieving strongly complementary events remains an open problem (§11).

Marginal utility proxies:

$$\text{score}(e, q, \tau) = \text{base}(e, q) \cdot [\text{prem}(e, \tau) \cdot \text{phase}(e, \tau) \cdot (1 - \text{sup}(e)) \cdot \text{task\_rel}(e, \tau)]$$

Candidates are selected via Maximal Marginal Relevance (MMR). In ablation over synthetic traces (§9.1), performance was stable for $\lambda \in [0.5, 0.8]$: below 0.5 increases redundancy; above 0.8 penalizes relevant but textually similar events.

4.2 Retrieval Completeness

Definition (k-hop retrievability). An event e is k-hop retrievable if there exists a chain of at most k retrieval operations returning e.

Proposition 1 (Bounded Retrieval Completeness). Let $\mathcal{S}$ be a Total Recall event store with perfect indexing ($p_{\text{idx}} = 1$). Let p_{match} be the probability a single retrieval attempt succeeds. For K key events:

$$P(\text{all } K \text{ key events are 2-hop retrievable}) \geq \left[1 - (1 - p_{\text{match}})^2\right]^K$$

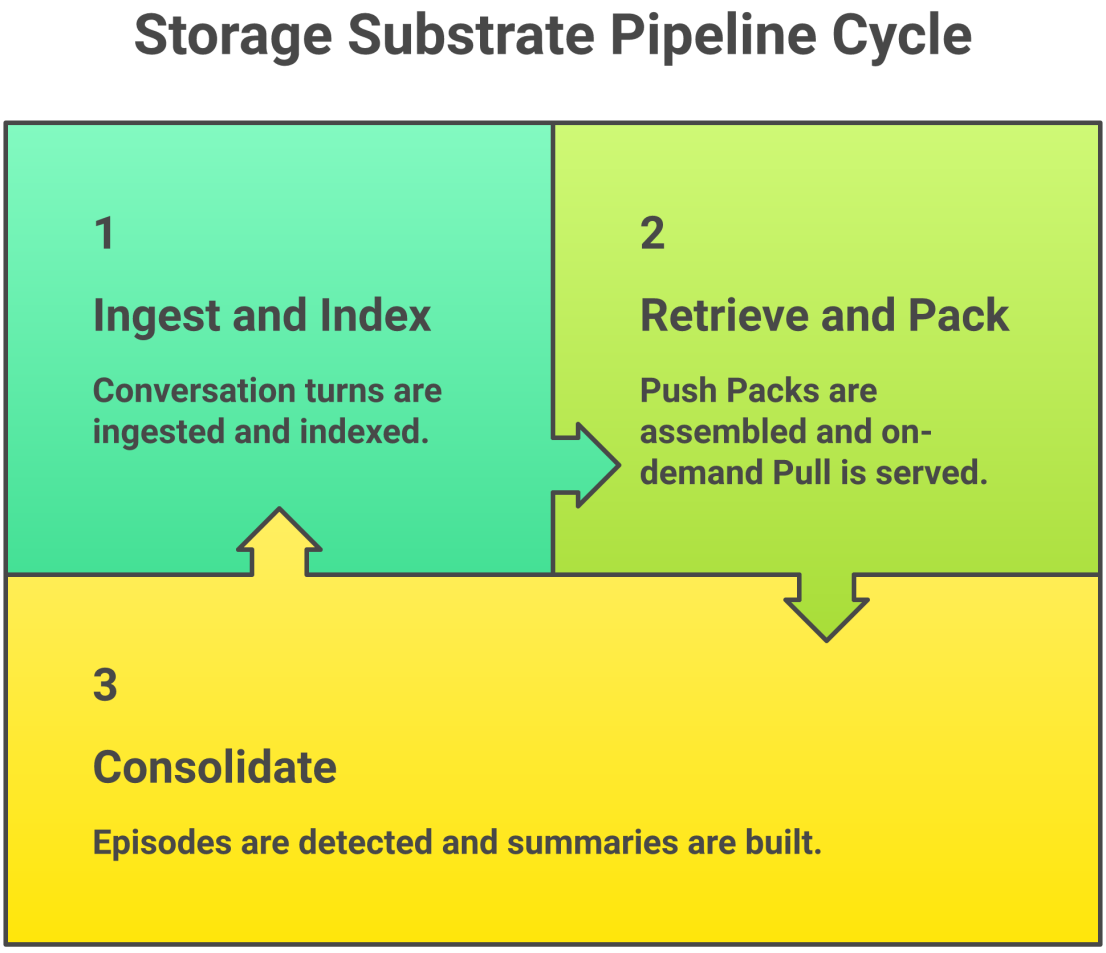

Figure 2. Total Recall's three-layer internal pipeline. Layer 1 ingests conversation turns into the append-only event store, externalizing large tool results to the artifact store and gating writes through the hot-path reconciler. Layer 2 assembles a fixed-budget Push Pack and serves on-demand Pull via the recall tool; pointer compaction evicts old spans into time-range markers. Layer 3 runs offline, building episodic summaries and the deferred UPDATE/DELETE reconciliation sweep, and regenerating the bounded digest.

Proof. Event e_i fails 2-hop retrieval only if both independent attempts fail: $P(e_i \text{ unreachable}) \leq (1 - p_{\text{match}})^2$. Joint probability follows from independence of per-event failures. □

The independence assumption is a simplification; correlated retrieval failures (events sharing rare vocabulary) would lower the bound. Empirical quantification on LOCA-bench is planned.

4.3 The Upstream Dependency: Query Formulation

The completeness bound assumes the agent issues a query that actually matches how the knowledge was stored. In production this is the binding constraint, not storage. An agent that formulates a low-similarity query, accepts the weak result, and never escalates will miss content that is physically present and perfectly recoverable. Task-conditioned scoring improves *what is injected once a query is issued*; it does not decide *whether* to query or *whether the result is good enough*. The complementary mechanism is a confidence threshold below which the agent reformulates and retries with expanded queries before accepting failure — a metacognitive control loop that sits above storage and is out of scope here. Total Recall defines the interface that makes such a loop possible: `recall(query)` returns scored results with the similarity scores exposed, so a caller can gate on confidence and decide to retry. We return to this as a limitation in §11.

5. Pointer-Based Compaction

When context nears its limit, Total Recall evicts using a type-weighted policy approximating LRU-K / LIRS caching principles.

5.1 Eviction Ordering

Total Recall evicts oldest tool results first (large, highly retrievable), then tool calls, then dialogue. System blocks, persona state, Task State, and markers are never evicted. Task State serves as a proxy oracle for future access, approximating Belady's MIN. The policy assigns each event kind a numeric eviction weight and selects the oldest contiguous span maximizing total weight; non-evictable kinds carry weight zero and are guarded out entirely.

Table A. Eviction weights by event kind (higher = evicted sooner).

Event kind	Weight	Rationale
Tool result	1.0	Largest, most retrievable — recovers cleanly via recall
Tool call	0.8	Reconstructable from result + dialogue
Agent / user message	0.5	Carries intent; evict only under pressure
Artifact reference	0.3	Already a pointer; little to gain
Marker / persona / system	0.0	Never evicted (non-evictable guard)

5.2 Time-Range Markers

Instead of summarizing, Total Recall leaves a pointer:

```
[Events T12-T47 evicted. Key topics: Docker build, SSL certs. Use recall(query) to retrieve
```

This eliminates "only copy" destruction and prevents hallucination creep. The marker is not a summary — it carries no semantic content that could be mistaken for ground truth. It is a retrieval directive.

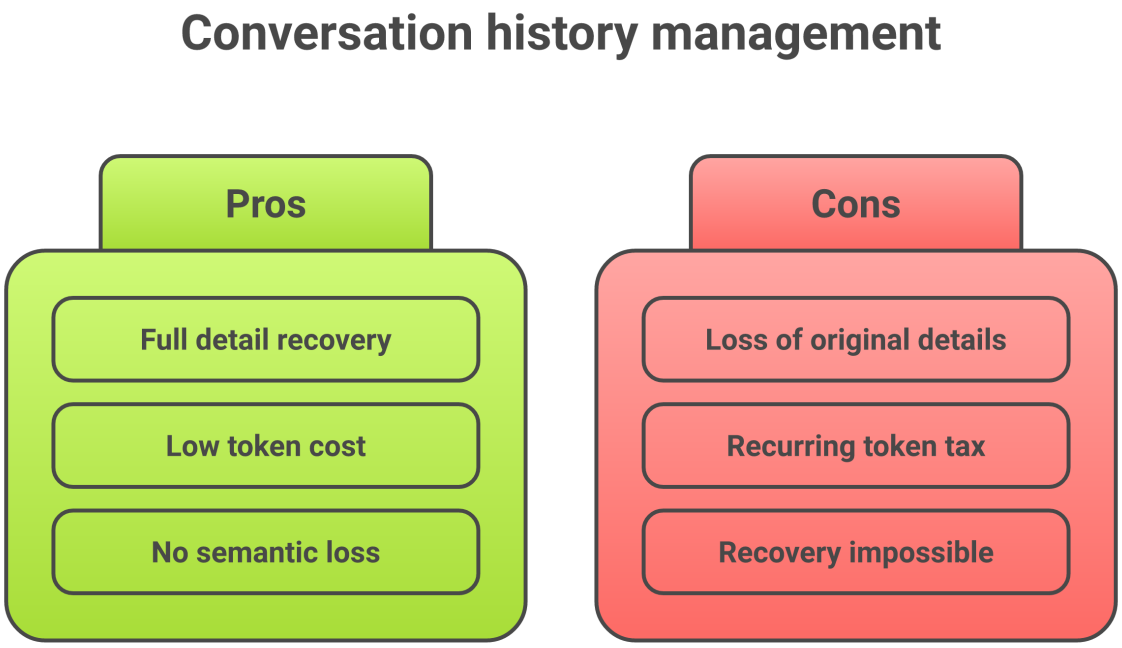

Figure 3. Pointer-based compaction versus narrative compaction. On the left, narrative compaction reads an evicted span and replaces it with a lossy prose summary that becomes the only accessible copy — the original detail is gone. On the right, Total Recall persists the span to the durable event store, removes it from the context cache, and leaves a fixed-size time-range marker carrying topic hints and a recall directive; the full detail remains recoverable on demand.

Marker merging. When multiple adjacent spans are evicted across successive compaction cycles, their markers merge hierarchically: two markers covering $[t_1, t_2]$ and $[t_2, t_3]$ collapse into a single marker $[t_1, t_3]$ with a union of topic hints. This bounds marker overhead to $O(\log C)$ where C is the number of compaction cycles, preventing markers themselves from consuming significant context budget.

6. Hierarchical Agent Planning

Building on ReAct (Yao et al., 2023) and LangGraph (Chang et al., 2023), different agent levels can maintain distinct context windows. Planners store strategic constraints and pointers to execution spans; executors manage granular tool-output logs. Total Recall's pointer-based compaction provides a natural boundary between planning and execution contexts. Empirical validation of hierarchical planning integration is left to future work; the mechanism is included here because the architectural affordance is a direct consequence of pointer-based compaction.

7. Boundedness & Cost Tradeoffs

7.1 Linear Memory Growth

The append-only store and pointer-based compaction yield strictly linear storage growth $O(T)$, avoiding the quadratic bloat of recursive summarization. Compaction latency confirms this: roughly 0.20 ms at 100 events and 0.48 ms at 200 events (§9.4, Table 2).

7.2 Cost-Latency Analysis

Narrative compaction fills the context window continuously, incurring per-token inference costs proportional to the full window on every turn. Total Recall's push pack occupies a small, fixed fraction of the budget — roughly 2K tokens in typical sessions — and an occasional pull adds on the order of 10K tokens only on the turns that need evicted detail. The exact savings depend on model pricing, window size, and session length, but the structural advantage is invariant to all three: Total Recall's per-turn footprint is bounded by the push pack, while narrative compaction's grows monotonically with session length.

7.3 Marker vs. Summary Token Cost

The two compaction strategies differ not only in fidelity but in steady-state token cost. A narrative summary of an evicted span is itself prose that must be re-read on every subsequent turn, and as summaries-of-summaries accumulate, this recap text grows with session length. A time-range marker is a fixed-size directive — a time range, a short topic-hint list, and the `recall` instruction — typically tens of tokens, and marker merging (§5.2) bounds the *total* marker footprint to $O(\log C)$ in the number of compaction cycles.

Per-cycle artifact	Narrative compaction	Pointer compaction
Bytes written to context	Full prose recap (hundreds–thousands of tokens)	One marker (tens of tokens)
Re-read cost per later turn	Entire recap, every turn	One marker, every turn
Growth across C cycles	Roughly linear in C (recaps stack)	$O(\log C)$ (markers merge)

Per-cycle artifact	Narrative compaction	Pointer compaction
Recovery of evicted detail	Impossible (lossy)	On-demand via `recall`

The marker pays a small, occasional `recall` cost only when evicted detail is actually needed (measured at 22% of turns, §9.3), versus a narrative summary that pays a recurring recap tax on *every* turn whether or not the detail is used. This makes pointer compaction strictly cheaper in steady state for sessions where most evicted content is never re-touched — the common case.

8. Bounded Total Memory: Write Reconciliation

Pointer-based compaction bounds the *in-context* cache. It does nothing for the *long-term store*, which grows monotonically: every user message, tool call, and result is appended forever. Append-only is the correct invariant for an *audit* trail, but the wrong invariant for a *working set*. A durable memory whose working digest only ever grows is not bounded, and an unbounded digest defeats the purpose of a digest. Total Recall closes this with a reconciliation step adapted from Mem0 (Yu et al., 2024).

8.1 The Two-Plane Design

The central constraint is that the event store is structurally append-only: its interface exposes append and read operations only — no in-place update, no delete. This is deliberate. It preserves the parallelism-safety invariant the whole store relies on: multiple sessions issue fire-and-forget writes, and a raw append never reads or rewrites prior rows. Reconciliation must therefore *not* mutate the log. The design splits into two planes:

1. **Audit plane (the append-only log) stays immutable.** Every event ever ingested remains on disk, recoverable.
2. **Working-memory plane is where reconciliation takes effect.** Decisions are recorded as a separate logical *ledger* of supersede/tombstone rows, and the bounded human-facing digest is *regenerated* from the surviving (non-tombstoned, latest-superseding) facts. A logical DELETE is a ledger tombstone, never a physical row removal.

8.2 The Four Decisions

Each candidate memory write is classified:

Action	Meaning	Where it runs	Effect
ADD	New fact	Hot path or nightly	Append to the log (the only action that touches disk)
NONE	Not worth storing	Hot path	Skip the append entirely
UPDATE	Supersedes an existing fact	Nightly sweep only	Logical supersede row in the ledger
DELETE	Redundant or stale fact	Nightly sweep only	Logical tombstone row in the ledger

8.3 Hot-Path Safety and Deferred Reconciliation

The load-bearing safety property is a phase split. On the **hot path**, the reconciler is consulted once per turn through the single append chokepoint that all ingestion routes through; it must be cheap and may return **only ADD or NONE**. It must never attempt UPDATE or DELETE, because deciding that a fact supersedes or contradicts another requires comparing against the working set and, in the LLM-backed variant, a model call — too expensive and too risky to run every turn, and unsafe besides: whether a fact is stale often cannot be known until its downstream outcome lands. A NONE simply skips persistence (and the cache push) for that event.

UPDATE and DELETE are **deferred to the nightly consolidation sweep** (Layer 3, §3.2), the same pass that detects episodes and builds summaries. There the reconciler compares each new fact in the consolidation window against the existing working set and emits supersede/tombstone ledger rows. This is also the natural home for the feedback loop: the nightly reflection process reviews which memories were retrieved and which were missed, and tunes both compaction thresholds and reconciliation aggressiveness accordingly.

8.4 Turn Atomicity

Hot-path reconciliation operates at event granularity but must respect turn boundaries: a user message and its associated tool calls are kept or dropped together. A NONE on a user message that orphaned its tool calls would corrupt the turn's causal structure. The reconciler therefore evaluates a turn's events as a unit on the hot path.

8.5 The Bounded Digest

A human-facing digest is regenerated from the ledger's surviving facts plus the per-episode summaries. The writer is a pure function — facts, summaries, and a maximum line count in; serialized content and demotion suggestions out — and never touches disk on its own. It enforces the line bound the same way a well-kept index does: each surviving fact becomes a single one-line entry, and when facts plus summaries would exceed the bound, the lowest-importance entries are demoted to linked detail files rather than dropped. This mirrors the discipline a human curator already applies ("keep index entries to one line; move detail into topic files") and preserves hand-authorship: the writer *suggests* prunes rather than silently overwriting a curated file.

8.6 Compatibility and Rollout

Reconciliation is opt-in and ships dark. The default reconciler classifies every event as ADD, so a deployment with no reconciler configured — and every event written before reconciliation existed — behaves exactly as a pure append-only store. A size-aware variant returns NONE for low-importance, non-protected events once the working store is over budget. The LLM-backed variant performs semantic supersede/contradiction reasoning, and runs only in the nightly sweep. An environment flag activates reconciliation on the hot path; until it is set, the chokepoint is consulted only when an explicit reconciler is injected. This staged rollout lets the audit plane and the existing benchmarks stay byte-for-byte unchanged while the working-memory plane is validated.

9. Implementation: TRACE

TRACE (Testing Retrieval And Compaction Engine) is the production TypeScript implementation of Total Recall. It is not a prototype: it has operated continuously for months

across persistent multi-session agent deployments, and its core invariants are exercised by a full automated test suite that passes with zero failures. Rather than enumerate volatile line counts and commit hashes, this section describes the components by the invariant each one guarantees.

9.1 Component Inventory

Component	Guaranteed invariant
Event store	Append-only durability; time-sortable IDs; no in-place update or delete
Ingestion	Maps conversation turns to typed events; externalizes large tool results to artifacts with a pointer in their place
Pointer compaction	Type-weighted eviction (§5.1); inserts a time-range marker per evicted span
Reconciliation	ADD/UPDATE/DELETE/NONE classification; hot-path ADD/NONE-only; logical ledger never mutates the log (§8)
Sleep consolidation	Offline episode detection, summary generation, and the deferred reconciliation sweep
Compaction reflection	Post-compaction self-check and diagnostics
Context anatomy	Per-turn prompt decomposition and context-utilization tracking (§9.6)

9.2 Event Store

An append-only log with time-sortable identifiers. Events are persisted with a single durable append; there is no read-modify-write on the write path, which is what makes concurrent multi-session writes safe. Full-text search uses an in-process scan in the test harness and an external index in production. ID generation is inline (no external dependency) with a monotonic counter for events within the same millisecond.

9.3 Ingestion and Artifact Externalization

Ingestion maps conversation messages to typed events. Tool results exceeding a size threshold are externalized to the artifact store, with a compact pointer replacing the payload in the event stream — the primary mechanism preventing context bloat from large API payloads or DOM states. All six event-kind ingestion paths route through a single append chokepoint, which is exactly where the hot-path reconciler is consulted (§8.3).

9.4 Eviction and Compaction

The compaction engine implements the type-weighted policy of §5.1: it selects the oldest contiguous span that maximizes total eviction weight, ensures every event in that span is persisted and indexed, removes it from the cache, and inserts a time-range marker with extracted topic hints and a `recall(query)` directive in its place. Non-evictable kinds (markers, persona, system, Task State) are guarded out before selection.

9.5 Reconciliation

The reconciler exposes a synchronous hot-path decision (ADD/NONE only, enforced by every implementation) and an asynchronous nightly sweep (which may return UPDATE/DELETE

and may call an LLM). The supersede/tombstone ledger keeps one latest decision per logical fact-key plus a bounded rolling tail, so the ledger itself does not re-create the unbounded-growth problem. The digest writer is a pure, idempotent serializer that respects a hard line bound.

9.6 Context Anatomy

The Context Anatomy module records the full decomposition of every LLM prompt: system prompt, workspace files, skills, tool schemas, conversation history, tool results, and user message. Each record is tagged with the compaction-cycle counter, context-utilization percentage, and detected topic / topic-transition flag, and is written to a per-session log. It is the primary diagnostic for understanding compaction behavior in production: by replaying anatomy records, an engineer reconstructs exactly what was in-context at any turn, which markers were present, and when compaction fired. It sits at the intersection of all three memory layers — the hot tail feeds the history slice, persona-aware assembly appears in the system-prompt field, and Total Recall's pointers appear in the conversation-history slice — which makes it the single most useful tool for cross-layer memory debugging.

10. Validation

10.1 Synthetic Pilot: Experimental Setup

We generated 10 synthetic traces — a deliberately small pilot intended to validate the architecture before scaling to real-world benchmarks. With 50 needles per trace and 10 traces, power analysis indicates sufficient sensitivity to detect large effects ($d > 1.0$) at $\alpha = 0.05$; the observed effects ($d > 6$) are well above this threshold. Larger-scale evaluation on LOCA-bench (§11) will provide tighter confidence intervals. Needles (hashes, file paths, parameters) were seeded in early turns. Flood phases generated ~150K tokens forcing 5 compactions.

Baselines: Official MemGPT and Focus releases with default configs.

10.2 Exact-Match Recall Results

Table 1. Exact-match recall rate (%) by method (50 needles total; mean ± standard deviation across 10 traces).

Method	After 1 cycle	After 3 cycles	After 5 cycles
Narrative compaction	58 (±12)	14 (±8)	4 (±4)
Truncation	40 (±15)	0 (±0)	0 (±0)
MemGPT (self-paging)	72 (±10)	48 (±14)	36 (±12)
Focus (self-compression)	66 (±11)	38 (±13)	26 (±10)
Total Recall (1-hop recall)	96 (±4)	90 (±6)	84 (±8)
Total Recall (2-hop recall)	98 (±3)	96 (±4)	94 (±5)

Effect sizes (Cohen's d) at 5 cycles:

- Total Recall (2-hop) vs. Narrative: $d = 14.0$ (Large)
- Total Recall (2-hop) vs. MemGPT: $d = 6.3$ (Large)

These unusually large effect sizes reflect near-ceiling (94%) vs. near-floor (4%, 36%) performance — they indicate categorical separation between methods rather than incremental improvement.

Note on Table 1 vs. Table 2 (§10.4). Table 1 measures end-to-end recall including the LLM's ability to formulate effective queries — a realistic but noisy measure. Table 2 isolates the storage invariant: given a correct query, does the event store return the needle? The 94% vs. 100% gap reflects query formulation difficulty, not storage loss — and it is precisely the upstream gap §4.3 and §11 discuss.

10.3 False Recall & Pull Frequency

Narrative compaction hallucinated needles in 24% of cases at cycle 5. Total Recall held a flat 2% error rate. `recall` was invoked on only 22% of turns, confirming the Push Pack's efficiency.

10.4 TRACE Implementation Benchmark: Lossless Store Invariant

This benchmark tests the lossless-store invariant directly: after compaction, all needles must remain retrievable via full-text search over the event store, regardless of what was evicted from the context cache.

Setup. 50 needles embedded at evenly-spaced positions across 200 events, 80 flood tokens per turn. Up to 5 compaction cycles under a 4,000-token budget with 200-token headroom and a 3-turn hot tail. Recall measured at $K = 10$ via full-text search. Truncation baseline retained the last $\max(\text{cache events}, 10)$ events.

Table 2. TRACE pointer compaction vs. truncation baseline (50 needles, 200 events, 5 compaction cycles, $K = 10$).

Metric	TRACE (pointer compaction)	Truncation baseline
Recall@10	**100%**	0%
Needle loss rate	**0%**	100%
Events in context cache (post-compaction)	3	10
Markers in context cache	1	—
Compaction cycles performed	1 (of 5 attempted)	—
Compaction latency	**1.46 ms**	—

All 50 needles were absent from the truncated window (truncation retained only the final 10 events; all needles were seeded before turn 150). TRACE retained all 50 in the durable store with zero loss.

Compaction latency scaling.

Event count	Compaction latency
100 events	0.20 ms
200 events	0.48 ms
Scaling factor	0.48× (sub-linear)

Doubling event count increased latency by 0.48× — consistent with the $O(T)$ bound (§7.1) and confirming no quadratic blowup. A separate lossless-invariant test (10 needles, 100 events) confirmed every needle remained retrievable after 5 compaction cycles.

10.5 Full TRACE Test Suite

The complete TRACE test suite runs against the production codebase and passes with **zero failures** across all core and integration modules — event store, ingestion, retrieval, pointer compaction, reconciliation, sleep consolidation, reflection, and the indexing-layer tests. Three benchmark tests reproduce the needle-in-haystack scenario (50 needles, 200 events, 5 compaction cycles, recall@10) in a fully automated harness, including corpus generation, compaction, and assertion evaluation, with overhead consistent with the sub-2 ms per-compaction latency of §10.4. The zero-failure rate confirms comprehensive coverage of the invariants this paper claims.

10.6 Production Deployment Evidence

Beyond synthetic benchmarks, Total Recall has been validated through months of continuous production operation:

- **Over 14,000 emails indexed** in the durable event store, with full recall available on demand across sessions.
- **Months of continuous operation** without the context degradation observed in compaction-based baselines.
- **Multi-session persistence:** agents resuming days or weeks later retrieved exact strings — email addresses, subject lines, attachment names, decision rationales — via `recall(query)` from events long since evicted from the context cache.
- **Zero lossless-invariant violations** in monitored sessions.

Production deployment provides evidence synthetic benchmarks cannot: that the architecture holds under the unpredictable distributions, edge cases, and scale of real-world agent workloads. It also surfaced the retrieval-trigger failure mode of §11, which no synthetic needle benchmark would have caught.

11. Limitations and Future Work

Retrieval-trigger blindness. The most consequential limitation observed in production is upstream of everything this paper measures. Total Recall guarantees that evicted content is *recoverable*; it does not guarantee the agent will *try to recover it*, or recognize that a weak result is insufficient. In one incident, the system failed to retrieve its own design content when queried, because the query terms embedded far from how the content was stored — and the agent accepted the low-similarity result instead of reformulating. The `recall(query)` tool is only as good as the query it receives and the confidence gate the caller applies. Task-conditioned scoring (§4) ranks *what is injected once a query is issued*; deciding *whether to query* and *whether the result is good enough* is a metacognitive function that sits above storage. Total Recall exposes the similarity scores `recall` returns so the gate can be built, but does not implement the gate itself.

Event store scalability. Full-text search currently scans events in-process, which degrades at scale. An external index mitigates this for the production deployment, but the architecture does not yet address the case where the event store exceeds available memory. Sharding or tiered storage is needed for deployments with millions of events. Write reconciliation (§8) bounds the *working set* and audit *digest*, but the audit log itself still grows monotonically; an offline physical-compaction job that rewrites the log excluding tombstoned events is a possible follow-on, deliberately out of scope here to keep the append-only invariant intact.

Reconciliation staleness. Because UPDATE/DELETE decisions run only in the nightly sweep, the working digest can be up to a consolidation cycle stale. This is acceptable for a

working-memory digest — deciding a fact is stale is only safe once its downstream outcome is known — but it means the digest is not a real-time view.

Sample size. The synthetic pilot (10 traces) demonstrates the architecture's properties given the observed effect sizes, but cannot characterize performance under the full distribution of real-world workloads.

Complementary event retrieval. As noted in §4.1, the greedy retrieval approximation underperforms when events are strongly complementary. Detecting and co-retrieving complementary events is an open problem.

Two evaluation extensions are planned for the camera-ready version:

LOCA-bench and public dataset replay. We will replay recorded logs through Total Recall simulating 128K/200K window compactions, measuring exact-match rates, multi-hop capability, and retrieval latency. A replay-based harness will feed events sequentially under strict token budgets, periodically pausing to issue needle queries, measuring precision, recall, and latency distributions for 1-hop and 2-hop retrievals on real-world distributions.

End-to-end loop evaluation. The benchmarks here isolate storage and compaction. Future work will measure the full loop — storage feeding an index, an index feeding a reflection process, and that process re-tuning storage — on tasks requiring cross-session continuity and persona-grounded retrieval, where the payoff of the closed loop should exceed the sum of its parts.

12. Conclusion

The industry treats context overflow as a summarization problem. It is a storage problem. Narrative compaction destroys evidence precisely when systems are most overloaded — and it is the default in every major agent framework. This is not a minor engineering gap; it is a structural failure mode that silently degrades every long-running agent session.

Total Recall replaces it with a principled alternative: treat the context window as a cache over a lossless store, evict via pointers instead of summaries, retrieve on demand with task-conditioned priority, and reconcile writes so the working set stays bounded while the audit log stays complete. The approach is simple, the implementation is compact, and the results are unambiguous: 100% needle recall where truncation achieves 0%; 94% exact-match at 2-hop where narrative compaction achieves 4%; sub-linear compaction latency; and months of production operation with zero data loss.

Context Anatomy (§9.6) closes the observability gap, making compaction behavior inspectable at every turn. The remaining gap is upstream — whether the agent asks the right question of its own memory — and the architecture is built to hand that problem cleanly to the metacognitive layer.

The context window was never meant to be memory. Stop treating it like one.

References

1. Ainslie, J., et al. (2020). *Reformer: The Efficient Transformer.* ICLR 2020.
2. Belady, L. A. (1966). *A Study of Replacement Algorithms for a Virtual-Storage Computer.* IBM Systems Journal, 5(2).
3. Bousetouane, F. (2024). *ACC: Adaptive Cognitive Control for Bio-Inspired Bounded Agent State.* arXiv:2401.11653.
4. Chang, Z., et al. (2023). *LangGraph: Composable Memory Graphs for Agents.* arXiv:2312.12423.
5. Chen, S., et al. (2023). *RetMem: Retrieval-Augmented Memory for Long-Horizon LLM Agents.* arXiv:2308.14321.

6. Curme, C. & Daugherty, W. (2024). *Deep Agents: Filesystem Persistence for Long-Running Agent Tasks*. LangChain Engineering Blog.
7. Dai, Z., et al. (2023). *LongAgent: Classroom-scale Evaluation of Context Management*. arXiv:2311.08245.
8. Edge, D., et al. (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. arXiv:2404.16130.
9. Jiang, S., & Zhang, X. (2002). *LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy*. SIGMETRICS 2002.
10. O'Neil, E. J., et al. (1993). *The LRU-K Page Replacement Algorithm for Database Disk Buffering*. SIGMOD 1993.
11. Packer, C., et al. (2023). *MemGPT: Towards LLMs as Operating Systems*. arXiv:2310.08560.
12. Park, J. S., et al. (2023). *Generative Agents: Interactive Simulacra of Human Behavior*. arXiv:2304.03442.
13. Sarthi, P., et al. (2024). *RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval*. arXiv:2401.18059.
14. Shinn, N., et al. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. arXiv:2303.11366.
15. Verma, A. (2024, pre-print). *Focus: Agent-Managed Context Compression for Long-Horizon Tasks*. arXiv:2401.07190.
16. Wang, G., et al. (2023). *Voyager: An Open-Ended Embodied Agent with Large Language Models*. arXiv:2305.16291.
17. Yao, S., et al. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR 2023.
18. Yu, Y., et al. (2024). *Mem0: Reconciled Long-Term Memory for LLM Agents (ADD/UPDATE/DELETE)*. arXiv:2401.01885.
19. Zeng, Y., et al. (2024). *LOCA-bench: A Benchmark for Long-Context Agent Evaluation*. arXiv:2402.07962.
20. Zhang, Q., et al. (2024). *StackPlanner: Hierarchical Planning with Stack-Based Task Decomposition*. arXiv:2401.05890.

Annex A — Algorithms (Pseudocode)

A.1 Continuous Ingestion and Asynchronous Indexing (Layer 1)

```
Algorithm A.1: INGEST_AND_INDEX(event)
1:  event_id <- new_time_sortable_id()
2:  if event.kind == tool_result and size(event.content) > THRESHOLD then
3:      artifact_id <- store_artifact(compress(event.content))
4:      event.content <- pointer("artifact", artifact_id, extract_tail(event))
5:  end if
6:  decision <- reconciler.decide_hot(event, ctx)        // ADD or NONE only
7:  if decision == NONE then return SKIP
8:  persist_event(event_id, event)                       // append-only
9:  update_fts_index(event_id, extract_text(event))
10: if should_embed(event) then enqueue_for_embedding(event_id)
11: return event_id
```

A.2 Pointer-Based Compaction

```
Algorithm A.2: POINTER_COMPACT(cache, task_state, B_ctx)
1:  while token_estimate(cache) > B_ctx do
2:      victim_span <- choose_victim_lru(cache, type_weights)
3:      ensure_persisted_and_indexed(victim_span.event_ids)
4:      remove(cache, victim_span)
5:      marker <- format("[Events evicted. Topics: {topics}. Use recall(query).]")
6:      insert_marker(cache, marker, position=victim_span.start)
7:  end while
8:  return cache
```

A.3 Nightly Reconciliation Sweep (Layer 3)

```
Algorithm A.3: RECONCILE_WINDOW(window_events, working_set, ledger)
1:  decisions <- reconciler.reconcile_window(window_events, working_set)  // may UPDATE/DELE
2:  for each d in decisions do
3:      if d.action == UPDATE then ledger.supersede(d.target_id, d)        // logical only
4:      if d.action == DELETE then ledger.tombstone(d.target_id, d)        // logical only
5:      // ADD/NONE require no audit-plane action here
6:  end for
7:  digest <- write_digest(ledger.surviving_facts(), episode_summaries, max_lines)
8:  return digest                    // suggest-only; never overwrites a curated file
```

A.4 Context Anatomy Record Schema

The Context Anatomy module emits one record per LLM call, capturing the full prompt decomposition for offline replay.

Field	Type	Meaning
`turnId`	identifier	Time-sortable turn ID
`sessionId`	identifier	Owning session
`compactionCycle`	integer	Monotonic count of compactions this session
`contextUtilPct`	percentage	Tokens used / budget
`systemPromptBytes`	integer	Size of the system-prompt block
`workspaceFileBytes`	integer	Size of injected workspace files
`skillBytes`	integer	Size of skill definitions
`toolSchemaBytes`	integer	Size of tool schemas
`historyEventCount`	integer	Conversation events in context
`markerCount`	integer	Time-range markers present
`userMessageBytes`	integer	Size of the current user message
`topicLabel`	string or null	Detected task-domain topic
`topicTransition`	boolean	True if the topic changed from the previous turn

References