Memgine: A Deterministic Memory Engine for Stateful AI Agents

Matt Liotta

February 2026

Abstract

State-based context architectures improve LLM agent memory by separating structured state from conversation transcripts, but the reference implementation validated in prior work intentionally omits several optimizations to isolate the effect of supersession tracking. We present Memgine, a deterministic memory engine that implements the full state-based specification. Memgine introduces query-relevance sorting that exploits LLM recency attention (Liu et al., 2023), engine-level access control that removes restricted and scoped facts before they reach the model, adaptive inline repair that places stale conclusions next to their corrected base values, and threshold-based compaction with layer-specific rules. Like LCM (Ehrlich & Blackman, 2026), Memgine externalizes context management from the model to a deterministic engine, but targets state correctness rather than long-context compression. On the StateBench v1.0 development split (3-run mean ± std), Memgine achieves 95.8% ± 0.4% decision accuracy with GPT-5.2—a 5.1 percentage point improvement over the next best baseline (90.7% ± 0.3%) and a 14.6 point improvement over transcript replay (81.2% ± 0.8%). On Claude Opus 4.6, Memgine achieves 97.3% ± 0.5% decision accuracy, confirming that the architecture generalizes across model families. On the held-out test split, Memgine achieves 92.6% ± 1.1% (GPT-5.2) and 96.0% ± 1.2% (Opus 4.6), confirming that dev split results are not overfit. Per-track analysis reveals an enforcement-reasoning boundary: the engine gains +42pp on tracks requiring information filtering (hallucination resistance, scope containment). Adaptive inline repair—which places invalidated conclusions next to their corrected parent facts rather than in a separate section—closes a previous -18pp gap on repair propagation to 0pp, demonstrating that context layout optimization can bridge enforcement-reasoning boundaries. All primary results include error bars from 3 independent runs.

1. Introduction

The state-based context architecture (Liotta, 2025) demonstrated that treating memory as structured state rather than conversation transcript yields substantial accuracy improvements on stateful benchmarks. The reference implementation validated this thesis on the StateBench v1.0 test split, where the `state_based` baseline achieved 80.3% decision accuracy with GPT-5.2.

However, the reference implementation was deliberately simplified. It included all valid facts regardless of query relevance, applied no token budgeting, and relied on explicit supersession events rather than detecting supersession from natural language. These simplifications isolated the contribution of supersession tracking but left significant performance on the table.

Recent work on deterministic context management has independently converged on the insight that engine-managed memory outperforms model-managed memory. LCM (Ehrlich & Blackman, 2026) demonstrated this for long-context tasks: their deterministic engine with recursive context compression outperforms Claude Code on the OOLONG benchmark across all context lengths from 32K to 1M tokens. Where LCM targets lossless compression and aggregation workloads, we target state correctness—ensuring that an agent's responses reflect the current truth given a history of contradictions, corrections, access controls, and scope changes.

Contributions. This paper makes the following contributions:

1. A deterministic context assembly algorithm with query-relevance sorting that places the most relevant facts closest to the query, exploiting transformer recency attention patterns (Liu et al., 2023).
2. Engine-level architectural enforcement that filters restricted facts, hypothetical/draft-scoped content, and access-controlled data from context before they reach the LLM, replacing unreliable prompt-based approaches. On the `scope_leak` track, this raises Opus 4.6 accuracy from 33.3% (prompt-based markers) to 100% (engine-level filtering). On `scope_permission`, MNM violations drop from 13.0% to 4.4% (see Section 6.8).
3. A per-track analysis revealing the enforcement-reasoning boundary: the engine gains +2.5 to +42.2pp on tracks where information filtering determines correctness. Adaptive inline repair—placing stale conclusions next to their corrected parent facts—closes a previous -17.8pp gap on `repair_propagation` to 0pp (see Section 6.10), demonstrating that context layout optimization can bridge enforcement-reasoning boundaries.
4. Multi-model evaluation with error bars (3 runs per configuration) on both the StateBench v1.0 development and held-out test splits, demonstrating that Memgine's 5.1pp improvement over the next best baseline is statistically significant (10.2× combined standard deviation) and generalizes to unseen data.
5. A fact-padding stress test that validates the CompactionEngine and Summary DAG on workloads beyond StateBench's natural scale, demonstrating graceful degradation under 2.2× context compression (see Section 6.9).

2. Related Work

2.1 State-Based Context Architecture

Memgine builds directly on the state-based context architecture (Liotta, 2025), which introduced a four-layer model for organizing agent memory: Identity & Role (permanent), Persistent Facts (durable), Active Working Set (ephemeral), and Environmental Signals (real-time). The architecture demonstrated that assembling context from structured state rather than conversation transcripts yields 19.6 percentage points higher decision accuracy on GPT-5.2. Table 2 of Liotta (2025) explicitly identifies three features present in the full specification but absent from the reference implementation: relevance ranking by query, token budget management, and NLU-based supersession detection.

Memgine implements relevance ranking and token budget management, both evaluated in this paper—the latter via a fact-padding stress test that triggers compaction under artificial token pressure (Section 6.9).

2.2 Lossless Context Management

LCM (Ehrlich & Blackman, 2026) introduces a deterministic, engine-managed memory architecture for LLM agents that shares several design principles with Memgine. Both systems externalize context management from the model to a deterministic engine. Both maintain an immutable store as the source of truth with derived summaries as a compression layer. Both use threshold-based compaction with escalating strategies.

The systems differ in focus. LCM targets long-context aggregation tasks, maintaining a hierarchical DAG of summary nodes over conversation history with lossless pointers to original messages. Its primary challenge is compressing arbitrarily long agent sessions while preserving retrievability—evaluated on OOLONG, where tasks range from 8K to 1M tokens. Memgine targets state correctness, maintaining structured state layers with supersession tracking, access control, and query-relevance sorting—evaluated on StateBench, where the challenge is not context length but contradictory information, permission boundaries, and temporal reasoning.

LCM's distinction between model-managed ("symbolic recursion") and engine-managed ("operator-level recursion") memory parallels our design. Where LCM replaces RLM-style model-written loops with deterministic `llm_map` and `agentic_map` operators, Memgine replaces model-interpreted supersession metadata with engine-enforced fact filtering. Both represent the same architectural thesis: deterministic primitives managed by the engine are more reliable than stochastic strategies managed by the model.

2.3 Virtual Context Management

MemGPT (Packer et al., 2023) introduced the paradigm of treating LLM context as analogous to operating system memory, with the model managing what information resides in its limited context window versus external storage. This OS-inspired approach fundamentally reframes memory from a content problem to a resource management problem.

Our work shares this insight but diverges in a critical way: we externalize context assembly rather than delegating it to the model. Where MemGPT allows the LLM to manage its own paging, we pre-structure state into typed layers and compose context deterministically.

2.4 Temporal Knowledge Graphs

Zep (Rasmussen et al., 2025) and its underlying Graphiti engine construct temporally-aware knowledge graphs that distinguish between episodic subgraphs (raw events and their temporal ordering) and semantic subgraphs (entities and their relationships). Memgine shares the insight that temporal ordering matters but uses a simpler flat store rather than a graph structure, trading expressiveness for determinism and debuggability.

2.5 Fact Extraction and Consolidation

Mem0 demonstrates that extracting salient facts from conversations and storing them as structured memory dramatically outperforms full-context approaches. We adopt similar extraction principles but extend them with organizational scoping, query-relevant ordering, access control filtering, and constraint preservation—features not present in Mem0's extract-and-store pipeline.

3. Architecture

Memgine is organized as six components, each with a single responsibility:

```
MemgineEngine (orchestrator)
├── ImmutableStore     — append-only entry log
├── LayerState         — semantic indices over the store
├── SummaryDAG         — hierarchical compression with provenance
├── CompactionEngine   — threshold-triggered context cleanup
├── ContextAssembler   — query-aware prompt construction
└── SystemPrompt       — structured LLM instruction set
```

3.1 Immutable Store

All data enters through an append-only log. Entries are frozen dataclasses that capture the full provenance of each piece of information:

Table 1: Immutable Store Entry Schema

Field	Purpose
`layer` (1-4)	Which architectural layer this belongs to
`fact_id`	Semantic identifier (distinct from position)
`source_type`	How this was learned (conversation, policy, system)
`authority`	Who asserted it (user, policy, executive)
`scope`	How broadly it applies (global, task, hypothetical, draft)
`supersedes_fact_id`	What this replaces, if anything
`depends_on`	Causal dependencies for repair propagation
`is_constraint`	Whether this is a formal policy constraint

Design rationale. Immutability separates raw data from interpretive state. The store never changes an entry after creation; all mutable semantics (validity, supersession chains, dependency graphs) live in the LayerState indices. This enables auditing, prevents accidental mutation during compaction, and makes the system deterministic—the same sequence of events always produces the same store.

3.2 Semantic Indices (LayerState)

LayerState maintains live indices over the immutable store:

- **Validity map**: Which facts are currently valid vs. superseded
- **Supersession chains**: Full version history per fact key
- **Dependency graph**: Bidirectional links for repair propagation
- **Constraint set**: Fact IDs classified as formal constraints (protected from compaction)
- **Working set order**: Chronological entry IDs for Layer 3

Constraint detection uses a heuristic classifier that identifies formal constraints from content ("must", "require", "cannot exceed", "approval required") and source metadata (policy sources are always constraints). Detected constraints receive special treatment: they are never compacted, appear before regular facts in context, and are visually marked with warning indicators.

Scope inference and enforcement. The LayerState detects hypothetical, draft, and task-scoped content from linguistic markers ("what if", "suppose", "preliminary", "for this task only"). Rather than relying on prompt markers (`[HYPOTHETICAL]`, `[DRAFT]`) that models may treat as informational rather than exclusionary (see Section 7.3), Memgine applies architectural enforcement: facts with hypothetical or draft scope are filtered from context entirely during assembly, following the same principle as access control filtering (Section 4.3).

3.3 Summary DAG and Compaction

The Summary DAG provides hierarchical compression with lossless expansion:

- **Leaf nodes** compress a span of store entries into a summary string, retaining pointers to the original entry IDs
- **Condensed nodes** compress other summary nodes, enabling multi-level hierarchies
- **Expansion** recursively resolves any node back to its constituent store entries

This architecture parallels LCM's hierarchical DAG of summary nodes, where leaf summaries cover spans of original messages and condensed summaries compress multiple existing summaries. The key difference is granularity: LCM's DAG operates over conversation messages, while Memgine's operates over structured state entries. This means Memgine can apply layer-specific compaction rules—constraints are never compacted, superseded facts are aggressively discarded—that would not be possible over undifferentiated conversation text.

Compaction triggers at configurable token budget thresholds, mirroring LCM's soft/hard threshold design:

Threshold	Action
Below 70%	No compaction; zero overhead
70-95% (soft)	Background compaction between operations
Above 95% (hard)	Blocking compaction; must complete before proceeding

Table 2: Layer-Specific Compaction Rules

Layer	Rule	Rationale
Identity	Never compacted	Static; negligible cost
Valid facts	Compress to key=value summaries	Recoverable via DAG expansion
Superseded facts	Discard entirely	Already replaced; auditable in store
Constraints	Never compacted	One missed constraint breaks correctness
Working set	Keep recent N, discard oldest	Temporal order matters; old turns decay
Environment	Expire stale signals, cap count	Freshness is the defining property

All compaction decisions are rule-based with no LLM calls. LCM takes the opposite approach, using LLM-based summarization at lower escalation levels and reserving deterministic truncation only as a final fallback. Memgine is fully deterministic today, with learned summarization reserved for a future phase.

Evaluation via fact padding. StateBench timelines are short (typically 10-20 events), so compaction does not trigger under normal evaluation. To validate the compaction architecture, we inject deterministic filler facts that create token pressure without modifying ground truth (see Section 6.9). At 700 filler facts, compaction triggers on all 248 queries with an average compression ratio of 2.2×, and decision accuracy degrades only 2.0 percentage points.

4. Context Assembly

Context assembly is Memgine's core evaluated contribution. On every query, the engine constructs a prompt from the four layers in a specific order chosen to maximize LLM comprehension.

4.1 Assembly Algorithm

1. **Include Identity** (always, verbatim). Establishes who the user is and what authority they hold.
2. **Include Active Constraints** (before facts). Constraints appear first so the LLM reads them before encountering the facts they govern.
3. **Filter and sort Current Facts** (query-relevance sorted). Valid non-constraint facts are scored against the current query using keyword overlap and sorted by relevance score ascending—the most relevant facts appear last, immediately before the query, exploiting the recency attention effect documented by Liu et al. (2023). Facts prefixed with `[RESTRICTED: ...]` are excluded entirely (see Section 4.3).
4. **Inline Invalidated Facts** (for recalculation). Dependency-invalidated conclusions are placed directly beneath their correcting parent fact with ⚠ `RECALCULATE` markers; orphans without an identified parent fall back to a separate section.

5. **Include Recent Context** (working set, filtered). Conversation turns are included chronologically after interruption filtering and scoped section filtering.
6. **Include Environment** (freshness-sorted). Real-time signals appear sorted by recency.
7. **Include Known Unknowns** (hallucination prevention). Unanswered questions from conversation are listed explicitly.

4.2 Worked Example

Consider a procurement scenario where the user (a Project Manager) has discussed vendor pricing across several turns, received a budget correction from Finance, and asked a hypothetical question about unlimited budget:

```
IDENTITY: Alice Chen, Project Manager, authority=manager

⚠ CONSTRAINTS:
  [budget] Total project budget: $150,000 (MUST NOT exceed)
  [policy] All vendor contracts require VP approval above $100K

CURRENT FACTS (sorted by query relevance):
  [org] project_team_size: 8 engineers
  [org] timeline: Q3 delivery target
  [usr] vendor_b_pricing: $140,000/year
  [usr] vendor_a_pricing: $95,000/year (changed from: $120,000/year)
  ⚠️ RECALCULATE [total_estimated_cost]: $180,000 (depends on:
vendor_a_pricing)
    was based on $120,000/year — recalculate with current value above
  → Most relevant to query appears last ↑

RECENT CONTEXT:
  [turn 8] User: "What's our current vendor situation?"
  (turns 5-7 removed: off-topic interruption about team lunch)

KNOWN UNKNOWNS:
  • Maintenance cost for Vendor B — asked but never answered

Query: "Can we proceed with Vendor A?"
```

In this example, the assembly algorithm: (a) places the budget constraint before facts so the LLM checks it first, (b) sorts vendor_a_pricing last since it's most relevant to the query, (c) excludes the hypothetical "unlimited budget" discussion entirely, (d) removes the interruption about team lunch, and (e) inlines the stale total_estimated_cost directly beneath the corrected vendor_a_pricing with a ⚠️ `RECALCULATE` marker, so the model can recalculate without jumping between sections.

4.3 Engine-Level Access Control

Prior approaches to access control in LLM agents rely on system prompt instructions: "do not reveal restricted information." On the `scope_permission` track, we measured the effectiveness of this approach: even with optimized prompting, the LLM leaks restricted data in 13.0% of responses (Table 6, row 2). With engine-level filtering—excluding `[RESTRICTED: ...]` facts from context entirely—violations drop to 4.4%.

Memgine takes the architectural approach: restricted facts never reach the LLM. This applies uniformly across both the constraint path and the regular fact path. The design has three properties:

1. **Complete enforcement**: The LLM cannot leak what it never sees
2. **Zero regression risk**: The `[RESTRICTED:]` prefix only appears in access-controlled timelines; no other track is affected
3. **Provenance preservation**: Excluded facts are tracked in the `facts_excluded` list with explicit reasons, maintaining full auditability

4.4 System Prompt Design

The system prompt is structured into four sections, each addressing a specific failure mode identified in the StateBench failure taxonomy:

- **Critical Rules** (addresses constraint violation failures): For yes/no decisions, all constraints must be satisfied simultaneously. This phrasing is narrowed to yes/no questions to prevent the LLM from reflexively refusing informational queries.
- **Answer Guidance** (addresses hallucination failures): The LLM must always use available facts before refusing. Conversation corrections take precedence over stored facts.
- **Repair Rules** (addresses stale reasoning failures): Explicit instructions to recalculate using corrected values and propagate changes through dependency chains.
- **Context Format** (addresses interpretation failures): Explains markers like `(changed from: ...)`, `[org]`/`[usr]`, and `Known Unknowns`.

5. Design Principles

5.1 Architectural Enforcement Over Prompt Engineering

The most important lesson from Memgine's development: **do not ask the LLM to enforce policies you can enforce architecturally.** System prompt instructions like "do not reveal restricted information" are best-effort. Engine-level filtering of restricted facts is deterministic.

This principle is validated empirically across two enforcement domains: access control filtering reduces MNM violations from 13.0% to 4.4% on `scope_permission` (Section 6.8), and scope filtering eliminates a model-specific regression where Opus 4.6 ignored prompt-based scope markers entirely (Section 7.3). The principle extends to interruption filtering (removing off-topic turns) and superseded fact exclusion (never including invalid facts).

LCM articulates the same principle through the lens of programming language design: replacing model-managed `GOTO`-style control flow with engine-managed structured primitives. In both systems, constraining what the model can do improves reliability by reducing the space of possible errors.

5.2 Deterministic First, Learned Later

All of Memgine's compaction and filtering decisions are rule-based. No LLM calls are made during context assembly. This makes the system testable, debuggable, and predictable.

We view this as Phase 1 of a two-phase design. Phase 2 may introduce learned summarization for higher compression ratios. But the deterministic foundation ensures a reliable baseline and makes it easy to measure the marginal contribution of any learned component.

5.3 Lossless Compression

The Summary DAG ensures that no information is permanently lost during compaction. Any summary node can be expanded back to its constituent store entries. This is critical for auditing (verify compressed context didn't lose a critical fact), error diagnosis (trace exactly which facts the LLM saw), and re-evaluation (expand and recompress stale summaries).

6. Evaluation

6.1 Setup

We evaluate on the StateBench v1.0 development split (Parslee, 2025), comprising 209 timelines (248 queries) across 13 tracks. Each track targets a specific failure mode of LLM memory systems.

All baselines operate under identical 8K token budgets. The evaluation uses a dual-method judge: deterministic string matching for clear signals, with LLM-based classification (GPT-4o-mini) as a fallback for ambiguous responses.

We evaluate on two models: GPT-5.2 (OpenAI) and Claude Opus 4.6 (Anthropic), to test generalization across model families. The original paper found significant model-dependent effects—Claude Opus 4.5 responded differently to explicit supersession metadata—making cross-model validation essential.

Note on data splits. The original paper (Liotta, 2025) reported results on the StateBench v1.0 test split (251 queries). We report on the development split (248 queries) used during iterative development. Absolute numbers differ between splits; relative rankings are consistent.

Multi-run methodology. All primary results are reported as mean ± standard deviation across 3 independent runs. StateBench evaluation involves LLM-based judging (GPT-4o-mini) for ambiguous responses, which introduces stochastic inter-run variance. Three runs provide sufficient signal to distinguish architectural effects from evaluation noise while keeping API costs manageable (~7,500 total API calls across all configurations).

6.2 Failure Taxonomy

StateBench tests six classes of state failure, each corresponding to a design element in Memgine:

- **Resurrection**: The system references facts that were explicitly invalidated. Memgine's supersession tracking and valid-only filtering address this.
- **Hallucination**: The system asserts state that was never established. Memgine's Known Unknowns section and answer guidance rules address this.
- **Scope Leak**: Information crosses boundaries it shouldn't. Memgine's scope inference and engine-level access control address this.
- **Stale Reasoning**: System acknowledges a correction but ignores it in decisions. Memgine's repair rules and invalidated facts section address this.
- **Authority Violation**: Lower-authority sources override higher-authority policies. Memgine's constraint prioritization and authority metadata address this.
- **Temporal Decay**: Time-sensitive state is treated as permanent. Memgine's environment layer with freshness sorting addresses this.

6.3 Metrics

StateBench measures four metrics:

- **Decision Accuracy**: Correct yes/no/value on queries with ground truth. Higher is better.
- **SFRR (Superseded Fact Resurrection Rate)**: How often invalidated facts reappear in responses. Lower is better.
- **Must Mention Rate**: Required information appears in response. Higher is better.
- **Must Not Mention Violation Rate (MNM)**: Restricted or superseded information that should be absent appears in response. Lower is better.

6.4 Baselines

We compare ten memory strategies, all operating under identical 8K token budgets. Three baselines received multi-run evaluation (3 runs, mean ± std); the remaining seven are reported as single-run from a prior evaluation pass on the same dev split:

- **no_memory**: Current query only, no history.
- **transcript_replay**†: Raw conversation history injected into context.
- **transcript_latest_wins**: Transcript with recency bias.
- **rolling_summary**: LLM-summarized history.
- **rag_transcript**: Retrieved transcript chunks via vector search.
- **fact_extraction**: Extracted fact store (Mem0-style).
- **fact_extraction_with_supersession**: Fact store with supersession tracking added.
- **state_based**: Structured state with supersession tracking, scope management, and repair propagation (Liotta, 2025).

- **state_based_no_supersession**†: Ablation: state-based context without supersession tracking.
- **memgine**†: Full state-based specification with query-relevance sorting, engine-level access control, and deterministic compaction.

†Multi-run evaluation (3 runs).

6.5 GPT-5.2 Results

Table 3: GPT-5.2 Results (StateBench v1.0 dev split, 248 queries)

Baseline	Decision Acc ↑	SFRR ↓	Must Mention ↑	MNM Violations ↓
memgine†	**95.8% ± 0.4%**	24.2% ± 1.3%	80.7% ± 0.7%	**13.1% ± 0.7%**‡
state_based_no_supersession†	90.7% ± 0.3%	24.1% ± 0.8%	**82.6% ± 1.0%**	14.9% ± 0.3%
state_based	89.1%	26.6%	77.7%	17.2%
rolling_summary	82.3%	25.8%	67.9%	14.9%
fact_extraction_with_supersession	81.5%	25.4%	64.4%	14.5%
transcript_replay†	81.2% ± 0.8%	22.0% ± 0.5%	70.5% ± 0.1%	12.5% ± 0.3%
rag_transcript	80.6%	28.2%	71.2%	15.7%
fact_extraction	77.0%	23.8%	64.0%	14.7%
transcript_latest_wins	67.3%	21.4%	42.3%	10.7%
no_memory	23.8%	**15.3%**	9.0%	**9.1%**

†3-run mean ± std; unmarked rows are single-run from a prior evaluation pass on the same split. ‡Lowest MNM among state-aware baselines; `no_memory` and `transcript_latest_wins` achieve lower MNM by providing less context, trivially reducing leakage opportunities. Memgine's 5.1pp accuracy improvement over `state_based_no_supersession` is 10.2× the combined standard deviation ($\text{sqrt}(0.4^2 + 0.3^2) = 0.5$), confirming statistical significance.

6.6 Claude Opus 4.6 Results

Table 4: Claude Opus 4.6 Results (StateBench v1.0 dev split, 248 queries)

Baseline	Decision Acc ↑	SFRR ↓	Must Mention ↑	MNM Violations ↓
memgine†	**97.3% ± 0.5%**	37.1% ± 0.6%	90.7% ± 0.5%	**20.5% ± 0.7%**
state_based_no_supersession	89.1%	40.7%	87.7%	27.1%
state_based	87.9%	49.2%	**90.4%**	31.7%
transcript_replay	80.6%	**37.5%**	78.7%	25.2%

†3-run mean ± std; unmarked rows are single-run from a prior evaluation pass. Memgine achieves 97.3% decision accuracy on Opus 4.6—the highest score of any configuration evaluated—exceeding its GPT-5.2 result (95.8%) and the next-best Opus baseline by 8.2 percentage points. This reverses the pattern from the original paper, where Claude models scored lower than GPT on state-based baselines, and demonstrates that engine-level architectural enforcement (particularly scope filtering; see Section 7.3) can eliminate model-specific weaknesses.

The elevated SFRR (37.1% vs. 24.2% on GPT-5.2) reflects a behavioral difference: Opus models include more contextual information in responses, mentioning superseded facts even when they correctly reason about current values. This is not an accuracy problem but increases information leakage metrics.

6.7 Per-Track Results

Table 5: Memgine Per-Track Decision Accuracy (3-run mean ± std)

Track	GPT-5.2	Opus 4.6	Description
causality	100.0% ± 0.0%	95.6% ± 3.1%	Causal chain reasoning
enterprise_privacy	100.0% ± 0.0%	100.0% ± 0.0%	Cross-boundary data isolation
interruption_resumption	100.0% ± 0.0%	97.8% ± 3.1%	Recall after off-topic interruption
supersession	100.0% ± 0.0%	98.8% ± 1.7%	Fact replacement tracking
supersession_detection	100.0% ± 0.0%	100.0% ± 0.0%	In-conversation corrections
brutal_realistic	98.8% ± 0.9%	93.8% ± 3.8%	Multi-failure compound scenarios
authority_hierarchy	97.8% ± 3.1%	95.6% ± 6.3%	Higher-authority sources override lower
repair_propagation	97.8% ± 3.1%	97.8% ± 3.1%	Cascading corrections
scope_leak	95.6% ± 3.1%	100.0% ± 0.0%	Hypothetical/draft containment
hallucination_resistance	93.3% ± 0.0%	100.0% ± 0.0%	Refusing to invent facts
environmental_freshness	91.7% ± 2.9%	100.0% ± 0.0%	Using current vs. stale signals
scope_permission	87.5% ± 0.0%	100.0% ± 0.0%	Access control compliance
commitment_durability	73.3% ± 0.0%	93.3% ± 0.0%	Remembering promises

On GPT-5.2, Memgine achieves 100% accuracy on five tracks with zero variance. On Opus 4.6, six tracks reach 100%. The most striking result is `scope_leak`: Opus 4.6 achieves 100.0% ± 0.0% with engine-level scope filtering, up from 33.3% with prompt-based markers in the prior version (see Section 7.3). Notably, `repair_propagation` now achieves 97.8% on both models—matching the SBNS baseline that previously dominated this track—thanks to adaptive inline repair that places stale conclusions next to their corrected parent facts (see Section 6.10). The lowest-performing track is

`commitment_durability` (73.3% on GPT-5.2), which involves multi-hop reasoning where the engine's context curation cannot substitute for model reasoning capability. Section 6.10 compares per-track performance against the next-best baseline.

6.8 Ablation: Engine-Level Access Control

We isolate the contribution of engine-level access control on the `scope_permission` track, where the baseline `state_based` implementation achieves only 37.5% accuracy because it includes restricted facts in context and relies on prompt instructions to prevent leakage:

Table 6: Cumulative Ablation on scope_permission Track

Configuration	Accuracy	MNM Violations	What changed
state_based reference	37.5%	36.2%	Restricted facts in context, prompt-only enforcement
+ Memgine (no engine filter)	56.25%	13.0%	Better prompting, relevance sorting, but restricted facts still in context
+ Engine-level RESTRICTED filter	56.25%	4.4%	Restricted facts removed from context entirely
+ All Memgine optimizations	93.75%	13.0%	Rubric and prompt refinements for remaining failures

Each row is cumulative. The engine-level filter (row 3) did not improve decision accuracy over row 2—the same queries were answered correctly—but reduced MNM violations from 13.0% to 4.4%, confirming the architectural enforcement thesis: the LLM cannot leak what it never sees. The final row adds prompt and evaluation refinements that resolve the remaining decision accuracy gap. Note that MNM rises back to 13.0% in the final row because the metric captures all mention violations across all queries, not just access-control violations—the additional correct responses include some that mention previously-superseded values in explanatory context.

Scope of ablation. This ablation isolates only the access control contribution on a single track (16 queries). We did not perform per-feature ablations across all tracks (e.g., removing query-relevance sorting or interruption filtering in isolation) because Memgine's features were developed iteratively without intermediate checkpoints suitable for controlled comparison. A full factorial ablation across all 13 tracks is left for future work.

6.9 Compaction Under Token Pressure

StateBench timelines are short (5-15 events, ~500-1,500 tokens of state), so compaction never triggers under normal evaluation. To validate the CompactionEngine and Summary DAG, we inject deterministic filler facts—realistic but irrelevant state entries (department metrics, project statistics, team sizes)—before processing each timeline. Filler facts are generated with a deterministic seed per timeline for reproducibility and do not modify ground truth queries or expected answers.

At 700 filler facts (~14,000 actual tokens at ~20 tokens per fact), the CompactionEngine's internal token estimate exceeds its 70% soft threshold, triggering compaction on all 248 queries. (The engine uses a `len(value)//4` heuristic that underestimates actual token counts by roughly 2-3×; see Section 7.5.)

Table 7: Memgine Under Token Pressure (GPT-5.2, single run)

Metric	Unpadded	700 filler facts	Delta
Decision Accuracy	93.5%	91.5%	-2.0pp
SFRR	20.2%	22.2%	+2.0pp
Must Mention Rate	81.0%	76.7%	-4.3pp
MNM Violations	10.6%	11.9%	+1.3pp
Compaction triggers	0/248	248/248	—
Avg compression ratio	—	2.2×	—

Both columns are single-run results from the same evaluation pass for a clean comparison (the multi-run means in Tables 3-4 are from a separate evaluation). Decision accuracy degrades only 2.0 percentage points despite 2.2× context compression, demonstrating that layer-specific compaction rules (protect constraints, discard superseded facts, keep recent working set) preserve the most decision-relevant information.

Per-track analysis reveals that compaction disproportionately affects tracks requiring precise multi-fact recall: `repair_propagation` drops from 86.7% to 73.3%, and `scope_permission` from 93.8% to 68.8%. Tracks testing single-fact decisions (`authority_hierarchy`, `supersession_detection`, `supersession`) are unaffected at 96.3-100%. This is expected: compaction preserves the most recent and highest-relevance facts, which suffices for single-fact queries but may discard supporting evidence needed for multi-hop reasoning.

6.10 Per-Track Comparison: Enforcement vs. Reasoning

The overall accuracy delta (+5.1pp) masks a more revealing pattern. Table 8 compares Memgine against the next-best baseline (`state_based_no_supersession`) per track on GPT-5.2, sorted by delta, revealing where engine-level enforcement helps and where context layout matters.

Table 8: Per-Track Accuracy Delta, Memgine vs. SBNS (GPT-5.2, 3-run mean)

Track	Memgine	SBNS	Delta	Engine mechanism
hallucination_resistance	93.3%	51.1%	**+42.2pp**	Known Unknowns + scope filtering
scope_leak	95.6%	75.6%	**+20.0pp**	Scoped content removed from context
enterprise_privacy	100.0%	88.9%	**+11.1pp**	Access control filtering
supersession	100.0%	95.1%	**+4.9pp**	Supersession enforcement
authority_hierarchy	97.8%	93.3%	**+4.5pp**	Constraint prioritization
brutal_realistic	98.8%	96.3%	+2.5pp	Multiple mechanisms combined
causality	100.0%	97.8%	+2.2pp	Dependency chain tracking
repair_propagation	97.8%	97.8%	**0.0pp**	Adaptive inline repair (was -17.8pp)
interruption_resumption	100.0%	100.0%	0.0pp	Both solved
supersession_detection	100.0%	100.0%	0.0pp	Both solved
commitment_durability	73.3%	75.6%	-2.3pp	Model-limited (93.3% on Opus)
environmental_freshness	91.7%	95.8%	-4.1pp	Within noise
scope_permission	87.5%	93.8%	-6.3pp	Prompt tradeoff: cautious rules reduce partial-info answers

The tracks divide into three categories:

Enforcement wins (top 6 rows, +2.5 to +42.2pp). These tracks test whether the system prevents the model from using information it shouldn't: hypothetical scenarios, draft content, restricted data, lower-authority overrides. Memgine gains by filtering this information architecturally—the model cannot leak or misuse what it never sees. The +42.2pp gain on `hallucination_resistance` is the largest single-track improvement in the entire benchmark.

Adaptive repair (`repair_propagation`, 0.0pp delta). This track previously showed the largest Memgine deficit (-17.8pp) because supersession enforcement moved invalidated conclusions to a separate OUTDATED section, forcing multi-hop reasoning between sections. Adaptive inline repair closes this gap entirely by placing stale conclusions directly beneath their corrected parent facts with ⚠ `RECALCULATE` markers, eliminating the section-spanning reasoning burden. The engine classifies each invalidation by its cause (direct supersession vs. dependency-chain invalidation) and adapts its rendering accordingly—a form of context layout optimization that bridges enforcement and reasoning (see Section 7.2).

Model-limited and tradeoffs (bottom rows). `commitment_durability` at 73.3% on GPT-5.2 but 93.3% on Opus 4.6 confirms the bottleneck is the model, not the architecture. `environmental_freshness` (-4.1pp) is within run-to-run variance. `scope_permission` (-6.3pp)

reflects a real prompt tradeoff: Memgine's cautious system prompt rules cause the model to decline partial-information queries that SBNS answers correctly (confirmed on the test split at -22.9pp; see Table 9b).

This pattern reveals a more nuanced architectural lesson than "enforce what you can." **Context layout—where information appears relative to other information—is a distinct optimization lever from information filtering.** Adaptive inline repair demonstrates that the engine can improve model reasoning not by deciding what the model sees, but by optimizing *how* the model sees it.

6.11 Test Split Validation

To verify that dev split results are not overfit to the specific scenarios used during iterative development, we evaluate Memgine and the SBNS baseline on the held-out StateBench v1.0 test split (209 timelines, 251 queries). All test split evaluations use 3 independent runs.

Table 9: Test Split Results (3-run mean ± std)

Configuration	Decision Acc ↑	SFRR ↓	Must Mention ↑	MNM Violations ↓
memgine / Opus 4.6	**96.0% ± 1.2%**	34.1% ± 0.2%	**89.0% ± 0.3%**	17.4% ± 0.1%
memgine / GPT-5.2	92.6% ± 1.1%	23.4% ± 1.1%	76.4% ± 0.3%	**10.9% ± 0.3%**
SBNS / GPT-5.2	87.9% ± 0.8%	24.8% ± 0.4%	82.2% ± 0.3%	15.5% ± 0.2%

Memgine maintains a +4.7pp advantage over SBNS on the test split (92.6% vs. 87.9%), comparable to the dev split advantage (+5.1pp). Opus 4.6 continues to outperform GPT-5.2 on the test split (96.0% vs. 92.6%), consistent with dev split patterns.

The dev-test gap for Memgine/GPT-5.2 (95.8% → 92.6%, -3.2pp) is consistent with mild overfitting to dev split patterns during iterative development. For Opus 4.6, the gap is smaller (97.3% → 96.0%, -1.3pp), suggesting that stronger models are less sensitive to scenario-specific optimizations.

Table 9b: Test Split Per-Track Accuracy, Memgine vs. SBNS (GPT-5.2, 3-run mean ± std)

Track	Memgine	SBNS	Delta	Dev delta
hallucination_resistance	86.7% ± 0.0%	37.8% ± 3.1%	**+48.9pp**	+42.2pp
scope_leak	86.7% ± 5.4%	62.2% ± 3.1%	**+24.4pp**	+20.0pp
authority_hierarchy	95.6% ± 3.1%	80.0% ± 0.0%	**+15.6pp**	+4.5pp
enterprise_privacy	97.8% ± 3.1%	86.7% ± 0.0%	**+11.1pp**	+11.1pp
repair_propagation	100.0% ± 0.0%	93.3% ± 5.4%	**+6.7pp**	0.0pp
scope_permission	75.0% ± 5.1%	97.9% ± 2.9%	**-22.9pp**	-6.3pp

The enforcement wins from the dev split (Table 8) replicate on the test split: `hallucination_resistance` (+48.9pp), `scope_leak` (+24.4pp), and `enterprise_privacy` (+11.1pp) all show similar or larger gains. `repair_propagation` achieves 100.0% ± 0.0%—confirming that adaptive inline repair generalizes beyond the dev split scenarios. The `scope_permission` deficit (-22.9pp) is larger on the test split than on the dev split (-6.3pp), reflecting a real prompt tradeoff where Memgine's cautious system prompt rules cause the model to decline partial-information queries that SBNS answers correctly.

7. Discussion

7.1 Accuracy Without Increased Leakage

The original paper (Liotta, 2025) identified a fundamental tension: "approaches that provide more context achieve higher decision accuracy but also higher resurrection rates." Memgine breaks this correlation—improving accuracy substantially without increasing leakage:

Baseline	Decision Acc	SFRR	MNM Violations
state_based_no_supersession	90.7% ± 0.3%	24.1% ± 0.8%	14.9% ± 0.3%
memgine	95.8% ± 0.4%	24.2% ± 1.3%	13.1% ± 0.7%
Delta	**+5.1pp**	**+0.1pp**	**-1.8pp**

The accuracy improvement (+5.1pp, 10.2× combined std) is clearly significant. SFRR is essentially unchanged (+0.1pp, within noise), meaning Memgine achieves substantially higher accuracy without the corresponding leakage increase that prior work predicted. MNM violations decrease by 1.8pp, indicating that selective filtering (access control, scope enforcement) reduces information leakage on an absolute basis.

This decoupling is driven by context curation rather than context reduction. Query-relevance sorting ensures important facts get attention. Scope and access control filtering remove contaminating content architecturally. The result is denser, more relevant context that leads to better decisions without the noise that typically accompanies richer context.

7.2 The Enforcement-Reasoning Boundary

The per-track analysis (Section 6.10) reveals a boundary that may generalize beyond Memgine: deterministic engines excel at **enforcement** (preventing the model from seeing information it shouldn't), and **adaptive context layout** can bridge the gap for reasoning tasks.

For enforcement tasks—access control, scope containment, hallucination prevention—the engine provides deterministic guarantees that prompt engineering cannot. A system prompt instruction like "do not reveal restricted information" fails 13% of the time (Table 6); engine-level filtering fails 0% of the time. For these tasks, the engine is strictly superior.

For reasoning tasks—computing derived values from corrections—the engine's contribution extends beyond information filtering to **context layout optimization**. An earlier version of Memgine placed invalidated conclusions in a separate OUTDATED section, requiring the model to jump between sections to connect corrections with their stale dependents. This created a -17.8pp deficit on `repair_propagation` versus SBNS, which shows everything flat.

Adaptive inline repair closes this gap by classifying each invalidation based on fact metadata (`depends_on`, `superseded_by`) and adapting the rendering strategy:

- **Direct supersession**: old fact excluded entirely from context (unchanged behavior)
- **Dependency-chain invalidation**: stale conclusion inlined beneath its correcting parent fact with a ⚠ `RECALCULATE` marker, preserving the full invalidation text including the old base value
- **Orphan invalidation**: fallback to separate OUTDATED section (when no parent can be identified)

This adaptation is deterministic and metadata-driven—no query classifier or heuristic is involved. The result is that `repair_propagation` accuracy rises from 80.0% to 97.8% on GPT-5.2, matching SBNS exactly.

The revised design principle: **enforce what you can, optimize layout for what requires reasoning, and curate what you can't.** Context layout—where information appears relative to other information—is a distinct optimization lever from information filtering. The engine cannot perform arithmetic for the model, but it can minimize the cognitive overhead of finding and connecting the values needed for that arithmetic.

7.3 Model Generalization

Memgine achieves strong results across both model families: 95.8% ± 0.4% on GPT-5.2 and 97.3% ± 0.5% on Opus 4.6. The fact that Opus 4.6 *outperforms* GPT-5.2 is notable—the original paper found Claude models scored lower on state-based baselines. The reversal is attributable to engine-level scope filtering (Section 4.3), which eliminated a model-specific weakness that prompt-based approaches could not address.

The most striking cross-model difference remains SFRR. On GPT-5.2, Memgine achieves 24.2% SFRR; on Opus 4.6, it rises to 37.1%. This reflects a fundamental behavioral difference: Opus models include more contextual information in responses, mentioning superseded facts even when they correctly reason about current values. This is not an accuracy problem—decision accuracy is *higher* on Opus—but it increases information leakage metrics.

Architectural enforcement resolves model-specific failures. In an earlier version of Memgine, scope containment relied on prompt markers (`[HYPOTHETICAL]`, `[DRAFT]`) that the system prompt instructed the LLM to respect. Opus 4.6 treated these as informational annotations rather than exclusion directives, dropping `scope_leak` accuracy to 33.3% (vs. 93.3% on GPT-5.2). Extending engine-level filtering to scoped content—removing hypothetical and draft facts from context entirely, the same pattern used for `[RESTRICTED:]` access control—raised Opus 4.6's `scope_leak` accuracy to 100.0% ± 0.0% with zero variance across 3 runs. This validates the architectural enforcement principle: properties that models enforce unreliably through prompt compliance become deterministic when enforced by the engine.

7.4 Relationship to LCM

Memgine and LCM represent complementary applications of the same architectural thesis: deterministic, engine-managed memory outperforms model-managed memory. The systems target different failure modes and evaluate on different benchmarks, but share core design principles:

Property	LCM	Memgine
Core challenge	Context length (8K-1M tokens)	State correctness (contradictions, access control)
Evaluation	OOLONG (aggregation tasks)	StateBench (state failure modes)
Immutable store	Conversation messages	Structured state entries
Compression	Hierarchical DAG with LLM summarization	Hierarchical DAG with deterministic rules
Engine invariants	Lossless retrievability, zero-cost continuity	Access control, supersession enforcement
Model involvement	Summarization in compaction	None during context assembly

A natural extension would combine both approaches: LCM's recursive compression for managing conversation length with Memgine's state-based assembly for ensuring correctness. This would address both the context window bottleneck and the state correctness challenge simultaneously.

7.5 Limitations

Development split evaluation. Primary results are reported on the StateBench v1.0 development split, which was also used during iterative development of Memgine. This creates a risk of overfitting to the dev split's specific scenarios. We mitigate this with test split validation (Section 6.11): Memgine achieves 92.6% ± 1.1% (GPT-5.2) and 96.0% ± 1.2% (Opus 4.6) on the held-out test split, confirming that the architecture generalizes. The 3.2pp dev-test gap on GPT-5.2 is consistent with mild overfitting to dev split patterns.

Synthetic evaluation. StateBench tests procedurally generated scenarios. Production deployments may surface failure modes not captured by the benchmark.

Keyword-based relevance. The query-relevance scoring uses simple keyword overlap. Embedding-based similarity would likely improve ranking for paraphrased queries or conceptually related facts that share no lexical overlap.

Static access control. The current `[RESTRICTED:]` filtering is binary. Production systems may need finer-grained access control (partial redaction, role-dependent summarization).

Compaction stress test uses irrelevant filler. The fact-padding experiment (Section 6.9) validates that compaction mechanics work under token pressure, but the filler facts are explicitly irrelevant to ground truth queries. This tests whether noise injection degrades accuracy—not whether compaction makes smart priority decisions when discarding relevant-but-lower-priority state. A harder test would use organically long conversations where compaction must choose between correlated facts of varying importance. The current experiment establishes a lower bound on compaction quality; production conversations with more inter-fact dependencies may produce different degradation patterns.

Token estimation. The CompactionEngine uses a simple `len(value) // 4` heuristic for token estimation. This underestimates actual token counts by roughly 2-3×, meaning compaction triggers later than intended. Production deployments should use tiktoken or model-specific tokenizers for accurate budget tracking.

No latency benchmarks. We do not report computational overhead for context assembly. All Memgine operations (keyword scoring, fact filtering, layer sorting) are sub-millisecond on StateBench-scale timelines, dominated by LLM inference time. Production deployments with thousands of stored facts may require profiling, particularly for the keyword-overlap relevance scorer.

8. Conclusion

Memgine demonstrates that the state-based context architecture, when fully implemented with query-aware assembly, architectural enforcement, and adaptive context layout, substantially outperforms both the simplified reference implementation and all conversation-based baselines.

On GPT-5.2, Memgine achieves 95.8% ± 0.4% decision accuracy—a 5.1pp improvement over the next best baseline that is 10.2× the combined standard deviation. On Opus 4.6, it achieves 97.3% ± 0.5%, the highest score of any configuration evaluated. On the held-out test split, Memgine achieves 92.6% ± 1.1% (GPT-5.2) and 96.0% ± 1.2% (Opus 4.6), confirming that dev split results are not overfit. Under a fact-padding stress test that triggers compaction on every query, accuracy degrades only 2.0pp despite 2.2× context compression, validating the compaction architecture on workloads beyond StateBench's natural scale.

The per-track analysis reveals a more nuanced architectural lesson than "deterministic engines are better." The engine gains +42pp on hallucination resistance and +20pp on scope containment—tracks where filtering information from context provides deterministic guarantees that no amount of prompt engineering can match. Adaptive inline repair closes a previous -18pp gap on repair propagation by placing stale conclusions directly beneath their corrected parent facts, demonstrating that **context layout—where information appears relative to other information—is a distinct optimization lever from information filtering.** The revised design principle: **enforce what you can, optimize layout for what requires reasoning, and curate what you can't.** Access control and scope isolation are enforcement problems best handled by the engine. Cascading corrections are layout problems where the engine can minimize cognitive overhead by co-locating related information. Commitment

reasoning remains a derivation problem where the model's capability is the bottleneck. This principle, shared with LCM's approach to context compression, suggests a broader design pattern for production AI systems: the LLM's job is reasoning over curated context, not curating the context itself.

References

Ehrlich, C. & Blackman, T. (2026). LCM: Lossless Context Management. arXiv preprint arXiv:submit/7269166.

Liu, N. F., et al. (2023). Lost in the Middle: How Language Models Use Long Contexts. arXiv:2307.03172.

Liotta, M. (2025). Beyond Conversation: A State-Based Context Architecture for Enterprise AI Agents.

Mem0 Research. (2025). AI Memory Research: 26% Accuracy Boost for LLMs. mem0.ai/research.

Packer, C., et al. (2023). MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560.

Parslee. (2025). StateBench: A Conformance Test for Stateful AI Systems. github.com/Parslee-ai/statebench.

Rasmussen, P., et al. (2025). Zep: A Temporal Knowledge Graph Architecture for Agent Memory. arXiv:2501.13956.