MNEMOSYNE: A Plugin-Layered Enhancement Suite for Personal-Agent Memory Systems

Oscar Serra

2 June 2026

Abstract

Personal AI assistants that accumulate months of memory face four gaps that general-purpose retrieval systems do not close: constant-time concept lookup degrades to linear search at scale, retrieval scoring ignores the agent's current task, memory writes can silently introduce contradictions, and context compaction permanently destroys information that was used once but never queried again. We present **MNEMOSYNE**, a plugin-layered enhancement suite that addresses all four gaps without modifying the underlying memory-core codebase. MNEMOSYNE registers four hooks — `retrieval_pre`, `before_message_write`, `before_compaction`, and `after_compaction` — on top of OpenClaw's markdown-first memory system. A concept index and compaction-aware capture are deployed in production; task-conditioned retrieval scoring and the contradiction gate are scaffolded for the next release. We also describe how the contradiction gate becomes a bi-temporal validity layer (§9.5): rather than blocking or warning, a contradicting write *supersedes* the prior fact by closing its validity interval, so the corpus retains full history and every retrieval can ask "what is true now" versus "what was true then." The validity substrate is not paper-only — memory-core already ships the validity columns, the temporal search predicate, and the supersede writer, so MNEMOSYNE has only to route the gate's decision into it. We position the compaction-capture feature explicitly against reversible Compress-Cache-Retrieve as implemented in chopratejas/headroom, which solves the eviction-loss problem from the opposite end (lossless-by-caching rather than selective-by-filtering), and we are candid that MNEMOSYNE's central performance claims remain architectural rather than benchmarked. The plugin packages all four features as a single installable unit for marketplace distribution.

1. Introduction

1.1 The Personal-Assistant vs Enterprise-Wiki Divergence

The memory-core library that MNEMOSYNE builds on was designed as a general-purpose memory system — part personal assistant, part enterprise wiki. It provides hybrid FTS5 full-text search, vector embeddings with cosine similarity, Maximal Marginal Relevance (MMR)

deduplication, and temporal decay scoring. For an enterprise wiki where queries are well-formed and contexts are stable, this suite is sufficient.

A personal assistant diverges in three structural ways. First, queries are ambiguous: "what was that thing about the cron job?" contains no entity that FTS5 can match. Second, the agent context-switches constantly — from debugging cron tasks to drafting emails to reviewing code — and retrieval scoring should reflect this. Third, the agent writes its own memories autonomously through nightly consolidation cycles, and those writes can contradict earlier facts without any human review gate.

These divergences motivate a layered enhancement approach rather than a fork of memory-core.

1.2 Why a Plugin, Not a Fork

OpenClaw's plugin system provides four properties that make it the correct integration point:

1. **Failure isolation.** A crash in MNEMOSYNE does not take down memory-core. The hook runner catches exceptions and logs them; the underlying retrieval continues unimpaired.
2. **Upgrade independence.** The memory-core library ships frequent updates. A plugin does not create merge conflicts.
3. **Distribution.** A single `tinkerclaw-memory-enhancements` package can be installed via ClawHub, OpenClaw's plugin marketplace, without users maintaining a fork.
4. **Composability.** Each of the four features can be independently enabled or disabled via plugin configuration.

1.3 Contributions

1. An O(1) concept index that short-circuits hybrid retrieval for known concepts (Feature 1).
2. A task-conditioned score modifier that biases retrieval toward the agent's current work (Feature 2).
3. A contradiction gate that intercepts memory writes before promotion to durable storage (Feature 3), and its bi-temporal evolution into a supersession layer (§9.5).
4. A compaction-aware capture pass that snapshots context before it is evicted (Feature 4).
5. A unified plugin architecture that delivers all four as a single distributable unit.

2. Background and Related Work

2.1 Underlying Memory-Core Architecture

OpenClaw's memory-core implements a markdown-first memory system with five retrieval stages:

1. **FTS5 full-text search** over a SQLite corpus.
2. **Vector embedding similarity** using cosine distance against pre-computed chunk embeddings.
3. **MMR deduplication** to prevent redundant chunks from consuming retrieval budget.
4. **Temporal decay** that exponentially down-weights older memories (half-life configurable, default 30 days).
5. **Relevance threshold** that filters chunks below a minimum combined score.

The retrieval pipeline is extensible via hooks registered at plugin load time. Memory-core exposes `retrieval_pre` (modify query or inject candidates before search), `before_message_write` (intercept writes to durable storage), `before_compaction` (observe context about to be evicted), and `after_compaction` (react to completed evictions). MNEMOSYNE uses all four.

2.2 Prior Components This Work Composes

MNEMOSYNE assembles four mechanisms that have been described and built separately, and which we summarize here so this paper stands on its own.

An event-sourced memory store with pointer-based compaction. In this model the context window is treated as a cache over a durable event log; when context is evicted, the system inserts time-range pointer markers (e.g., "cron debugging happened between 14:00 and 15:30") rather than discarding the slot silently. MNEMOSYNE's compaction capture (Feature 4) operates on this substrate: it intercepts the eviction to preserve the content that a pointer alone cannot recover.

A pre-computed concept-to-address index for constant-time retrieval. The idea is to map a finite vocabulary of recurring concepts — project names, people, tools, error classes — to their associated memory chunks offline, so that a query containing a known concept can be answered without a full hybrid scan. MNEMOSYNE's Feature 1 is the production plugin implementation of this index, and it also owns the bi-temporal *schema* — the `validity_start / validity_end / ingestion_time` columns and the temporal search predicate — that MNEMOSYNE's validity layer (§9.5) sits on top of.

An identity-and-consistency layer that keeps a persona coherent across sessions. A long-running agent maintains a structured record of what it "believes" to be true about itself; that record must not drift or contradict itself across restarts. MNEMOSYNE's contradiction gate (Feature 3) protects this kind of structure from silent corruption.

Nightly reflection and consolidation ("dreaming"). Between sessions, the agent summarizes the day's events and promotes durable memories autonomously. These autonomous writes — not human edits — are precisely what Feature 3 gates and Feature 4 captures.

2.3 Related Systems

Mem0 [MemoryOS, 2024] builds multi-level memory (short-term, episodic, long-term) but does not address task-conditioned retrieval or write-time contradiction checking. **MemGPT** [Packer et al., 2023] introduces virtual context paging but operates at the context-window level rather than the retrieval-scoring level. **A-MEM** [Xu et al., 2025] gives agents control over memory formation strategies; MNEMOSYNE complements this by adding retrieval-time and write-time gates that A-MEM does not address. None of these three attacks the eviction-loss problem that Feature 4 targets: they manage what is stored and how it is layered, not what is destroyed when the context window overflows.

Reversible Compress-Cache-Retrieve (headroom). The most directly comparable external system to Feature 4 is chopratejas/headroom (Apache-2.0), an open-source context-compression layer for agents built on *reversible CCR — Compress-Cache-Retrieve*. Rather than dropping evicted content, headroom compresses it, caches the originals, and lets the LLM retrieve the full original on demand. It ships per-content-type compressors — statistical crushing of JSON arrays ("SmartCrusher"), AST-aware code compression via tree-sitter, and dedicated log/diff/text handlers — reports 60–95% token savings at near-zero accuracy delta on public benchmarks (GSM8K, TruthfulQA, SQuAD, BFCL), and, distinctively, ships a *reproducible* evaluation suite (`python -m headroom.evals`) alongside a trained compression model (Kompress-v2-base).

headroom and MNEMOSYNE solve overlapping problems from opposite ends, and the differentiation is worth stating precisely because it is not a matter of one being a subset of the other. headroom is *content-agnostic, reversible, and lossless-by-caching*: every byte survives eviction in compressed form and is recoverable, with type-specific compressors doing the work. It has no concept of task-conditioned retrieval, no write-time contradiction or validity semantics, and no bi-temporal "what was true then" axis. MNEMOSYNE is *selective and semantic*: the `isTrivial`/redundancy filtering of §7.3 deliberately discards greetings and acknowledgments and keeps only the "by the way" details, and its capture feeds the concept index and — via

§9.5 — a validity-aware corpus. The slogans are complementary: headroom maximizes recall-of-everything-per-token; MNEMOSYNE maximizes signal-per-stored-chunk and supplies the retrieval, write-gating, and temporal machinery that headroom does not attempt. The two are composable rather than competing — headroom could compress MNEMOSYNE's `captured/` corpus, and conversely MNEMOSYNE's nearest CCR analogue, the pointer marker, is *lossy* exactly where headroom's cache is reversible. §7.5 returns to this as a concrete design gap.

Ecosystem eval discipline. A broader signal frames the empirical posture of this revision. Both headroom and coreyhaines31/marketingskills — a large, widely adopted skills collection — independently converge on shipping per-system reproducible assertion suites (headroom's `headroom.evals`, marketingskills' per-skill `evals.json`-style checks). When competing and adjacent systems all ship runnable evidence, an unmeasured systems paper reads as under-evidenced. §10.4 confronts this directly.

No existing system combines O(1) concept lookup, task-conditioned scoring, write-time contradiction gating, and compaction-time capture in a single plugin-layered architecture. The novelty is the *composition* and the validity layer, not any single mechanism; headroom in particular is the stronger system on the narrow axis of eviction-loss recovery, and we say so.

3. Architecture

3.1 Plugin-Layered Design

MNEMOSYNE is a single OpenClaw plugin with ID `tinkerclaw-memory-enhancements`. It registers four hooks at load time:

```
/**
 * MNEMOSYNE plugin entry point.
 * Registers four hooks on memory-core's retrieval and write
 * pipelines. No modification of the underlying source required.
 */
export function activate(api: PluginAPI): void {
  api.hooks.register('retrieval_pre', conceptIndexLookup);
  api.hooks.register('retrieval_pre', taskConditionedScoring);
  api.hooks.register('before_message_write', contradictionGate);
  api.hooks.register('before_compaction', compactionCapture);
}
```

Hook execution order is deterministic: hooks registered first run first. The concept index runs before task-conditioned scoring so that O(1) lookups can short-circuit the pipeline entirely — if the concept index returns a high-confidence result, the full hybrid search is skipped.

Figure 1 shows the full data flow: how a query enters through the two `retrieval_pre` hooks, how a write passes the contradiction gate before promotion, and where the supersede path routes the gate's decision into memory-core's bi-temporal writer (§9.5).

3.2 Hook Surface

Hook	Timing	Receives	Returns
`retrieval_pre`	Before hybrid FTS+vector search	Query string, task descriptor, retrieval config	Modified query, injected candidates, or `skip_search` signal
`before_message_write`	Before promotion to MEMORY.md	Candidate chunk, existing facts	`allow`, `warn`, `block`, or (§9.5) `supersede`
`before_compaction`	Before context eviction	Messages about to be dropped	Captured chunks (written to corpus)
`after_compaction`	After eviction completes	Eviction metadata	Void (used for bookkeeping)

3.3 Failure Isolation

Each hook is wrapped in a try-catch at the memory-core level. If MNEMOSYNE throws, the exception is logged to the plugin's own log file (`~/.openclaw/logs/tinkerclaw-memory-enhancements.log`) and the pipeline continues as if the hook were not registered. This means:

- A bug in the concept index does not prevent hybrid search from running.
- A crash in the contradiction gate does not block memory writes.
- A failure in compaction capture does not prevent compaction.

3.4 Storage Isolation

MNEMOSYNE stores its data separately from memory-core:

- **Concept index:** `~/.openclaw/data/mnemosyne/anchors.json`
- **Capture corpus:** `~/.openclaw/data/mnemosyne/captured/` (one `.md` file per capture event)
- **Contradiction log:** `~/.openclaw/data/mnemosyne/contradictions.jsonl`
- **Task descriptor cache:** In-memory only (ephemeral per session)

This separation ensures that uninstalling the plugin leaves memory-core's data untouched. The one exception is the bi-temporal validity layer (§9.5), which writes through memory-core's own `chunks` table rather than a separate store — because superseding a fact must close *that fact's* validity interval, not a shadow copy.

4. Feature 1: Concept Index for O(1) Retrieval

4.1 Design

A personal assistant's memory corpus contains a finite vocabulary of recurring concepts — project names, people, tools, error classes. Mapping these concepts to their chunks offline turns query-time retrieval into a constant-time lookup: when a query contains a recognized concept token (e.g., "cron", "pallet-scan", "WhatsApp"), the index returns the pre-ranked chunk list directly, bypassing the full FTS5 + vector + MMR pipeline.

4.2 Data Structure

```
/**
 * Concept index. Maps concept tokens to their associated memory
```

```
 * chunk IDs with importance scoring for retrieval ranking.
 * Persisted to anchors.json, loaded lazily on first query.
 */
type ConceptEntry = {
  chunkIds: Set<string>;
  importance: number;
  lastSeenAtMs: number;
};

const conceptIndex: Map<string, ConceptEntry> = new Map();
```

Each entry stores: - **chunkIds**: The set of memory chunk identifiers associated with this concept. Chunk IDs are opaque strings assigned by memory-core. - **importance**: A scalar combining frequency, recency, and distinctiveness (§4.5). - **lastSeenAtMs**: Unix timestamp of the most recent ingestion event that referenced this concept, used for staleness pruning.

4.3 Ingestion

The index is built during nightly consolidation. For each memory chunk processed during consolidation:

1. **Tokenize** the chunk into candidate concept tokens using whitespace splitting and stop-word removal.
2. **Filter** to tokens that appear in the anchor vocabulary — a curated set of entity names, project identifiers, tool names, and error classes extracted from the agent's operational history.
3. **Update** the concept index: for each surviving token, add the chunk ID to the token's `chunkIds` set and recompute importance.

Incremental updates occur during the session when new memories are written: the `after_compaction` hook triggers a lightweight re-index of newly captured chunks.

4.4 Lookup

At query time, the `retrieval_pre` hook tokenizes the incoming query and checks each token against the concept index:

```
function conceptIndexLookup(ctx: RetrievalContext): RetrievalResult | null {
  const tokens = tokenize(ctx.query);
  const matches: ConceptEntry[] = [];

  for (const token of tokens) {
    const entry = conceptIndex.get(token);
    if (entry && entry.importance >= CONFIDENCE_THRESHOLD) {
      matches.push(entry);
    }
  }

  if (matches.length === 0) return null; // fall through to hybrid search

  const chunkIds = unionChunkIds(matches);
  const ranked = rankByImportance(chunkIds, matches);
  return { candidates: ranked.slice(0, ctx.config.maxResults), skipSearch: true };
}
```

When the index returns a result with `skipSearch: true`, memory-core skips the full hybrid pipeline. If no concept matches above the confidence threshold, the hook returns `null` and hybrid search proceeds normally.

4.5 Importance Scoring

Importance is computed as a weighted sum of three factors:

```
importance(token) = w_f * frequency(token) + w_r * recency(token) + w_d * distinctiveness(to
```

Where: - **frequency** = log(1 + count of chunks containing this token), normalized to [0, 1]. - **recency** = exponential decay from `lastSeenAtMs` with a half-life of 14 days. - **distinctiveness** = inverse document frequency: tokens that appear in fewer chunks score higher. A token appearing in every chunk (e.g., "the") has distinctiveness near zero.

Default weights: `w_f = 0.3`, `w_r = 0.4`, `w_d = 0.3`. Recency is weighted highest because a personal assistant's interests shift over weeks — a project abandoned two months ago should not dominate retrieval.

4.6 Pruning

Concepts whose `lastSeenAtMs` is older than 90 days and whose importance has decayed below `PRUNE_THRESHOLD` (default 0.05) are removed during nightly consolidation. This prevents the index from growing unboundedly as the agent's operational vocabulary evolves.

Pruning is conservative: a concept is removed only if both conditions hold. A rarely-seen but highly distinctive concept (e.g., a unique project name) may persist indefinitely because its distinctiveness score keeps importance above the threshold.

4.7 Complexity Analysis

Operation	Hybrid search (memory-core)	Concept index (MNEMOSYNE)
Query-time retrieval	O(n log n) — FTS5 scan + vector cosine + MMR	O(|Q|) — token lookup in Map, where |Q| is query token count
Index build	N/A	O(n * |V|) — one-time, during nightly consolidation
Incremental update	N/A	O(|V|) per new chunk
Storage	FTS5 index + vector store	anchors.json (~50KB for 2,000 concepts)

The critical property is that query-time retrieval is **O(1) in corpus size n**. As the memory corpus grows from thousands to hundreds of thousands of chunks, hybrid search degrades (FTS5 scan time grows, vector search requires more comparisons); the concept index remains constant. This is a theoretical property of the data structure, not a measured result; §10.4 is explicit that wall-clock confirmation on the production corpus is still pending, and treats the absence of that measurement as a substantive limitation rather than a footnote.

4.8 Persistence

The index is serialized to `anchors.json` using JSON, with `Set` values converted to arrays:

```
{
  "cron": {
    "chunkIds": ["c_4a2f", "c_891b", "c_3e7d"],
    "importance": 0.72,
    "lastSeenAtMs": 1744675200000
  },
  "whatsapp": {
    "chunkIds": ["c_12ab", "c_56cd"],
    "importance": 0.61,
    "lastSeenAtMs": 1744588800000
  }
}
```

Loading is lazy: the file is read and parsed on the first retrieval query of each session. Subsequent queries use the in-memory Map.

5. Feature 2: Task-Conditioned Retrieval Scoring

5.1 Motivation

Standard retrieval scoring is **query-only**: the score of a memory chunk depends solely on its textual similarity to the query. For a personal assistant that context-switches between tasks throughout the day, this is insufficient.

Consider: the agent is debugging a cron failure. The user asks "what did we change last week?" Without task conditioning, the retrieval system returns all changes from last week — code commits, email drafts, configuration edits — ranked by textual similarity to "change" and "last week." With task conditioning, chunks tagged with cron-related metadata are boosted, surfacing the specific cron script modifications that the user implicitly meant.

The phenomenon is well-studied in information retrieval as **query context bias** [Croft et al., 2010], but has not been applied to agent memory systems where the "context" is the agent's current task rather than a user profile.

5.2 Task Descriptor

The task descriptor is a lightweight structure extracted from the agent's current session state:

```
type TaskDescriptor = {
  keywords: string[];       // e.g., ["cron", "merge", "guardian"]
  projectId: string | null; // e.g., "tinkerclaw"
  taskType: string | null;  // e.g., "debugging", "writing", "review"
};
```

The descriptor is populated from two sources: 1. **Explicit task markers** — if the agent's session includes a task header (e.g., `## Task: Debug safe-cron-merge.sh`), the keywords and project are extracted directly. 2. **Implicit inference** — if no explicit marker exists, the descriptor is inferred from the last N messages using keyword extraction (TF-IDF over the session window).

5.3 Score Modifier

The task-conditioned score modifier is registered as a second `retrieval_pre` hook that runs after the concept index lookup (and only if the concept index did not short-circuit):

```
function taskConditionedScoring(ctx: RetrievalContext): void {
  const task = ctx.taskDescriptor;
  if (!task || task.keywords.length === 0) return; // no task context available

  for (const candidate of ctx.candidates) {
    const taskRelevance = computeTaskRelevance(candidate.metadata, task);
    candidate.score += TASK_WEIGHT * taskRelevance;
  }
}
```

The modifier does not replace existing scores — it adds a weighted bonus. Chunks already highly relevant to the query stay highly ranked; the task bias adjusts marginal rankings.

5.4 Weight Calibration

The `TASK_WEIGHT` parameter (default 0.3) controls how aggressively task context biases retrieval. At 0.0, the modifier has no effect. At 1.0, task relevance dominates textual similarity.

The default of 0.3 was chosen empirically: high enough to promote task-relevant chunks from page 2 to page 1 of results, low enough that a query about an unrelated topic still returns correct results. Formal calibration against an annotated retrieval-failure benchmark is planned for the next release.

5.5 Interaction with Concept Index

The two `retrieval_pre` hooks interact as follows:

1. If the concept index fires and returns `skipSearch: true`, the task-conditioned modifier never runs. This is correct: O(1) concept lookup already returns the right chunks for well-known concepts, and adding task bias would only degrade a precise result.
2. If the concept index does not fire (no recognized concepts in the query), the task-conditioned modifier runs on the hybrid search candidates. This is where it adds the most value — ambiguous queries where hybrid search returns too many marginally relevant results.

6. Feature 3: Contradiction-Gate on Writes

6.1 Threat Model: Agent Self-Corruption

Autonomous agents that write their own memories face a risk that human-curated knowledge bases do not: **self-corruption through hallucinated or outdated facts**. The threat manifests in three scenarios:

1. **Hallucinated consolidation.** During nightly dreaming, the LLM summarizes the day's events. If the summary includes a hallucinated detail — a wrong date, an incorrect attribution, a confused sequence of events — that hallucination is promoted to durable storage and becomes a "fact" the agent retrieves in future sessions.

2. **Stale override.** The agent learns a new fact ("the meeting was moved to Thursday") but a later consolidation cycle processes an older memory that still says "the meeting is on Wednesday." If the older memory is promoted after the newer one, the durable store now holds a contradiction, and temporal ordering may not resolve it.

3. **Semantic drift.** Over months, repeated summarization introduces gradual semantic shifts. A project described as "experimental" becomes "production" through successive paraphrases, not through any actual status change.

These threats are not hypothetical. A self-contradiction failure was observed in production when the agent flagged a calendar conflict for a meeting it had already noted as postponed.

6.2 Gate Mechanism

The contradiction gate registers on the `before_message_write` hook. When memory-core's dreaming phase prepares to promote a short-term recall to MEMORY.md, the gate intercepts the candidate chunk and performs a targeted contradiction check:

```
async function contradictionGate(ctx: WriteContext): Promise<WriteDecision> {
  const candidate = ctx.chunk;
  const existingFacts = await queryRelatedFacts(candidate);

  if (existingFacts.length === 0) {
    return { action: 'allow' }; // no existing facts to contradict
  }

  const contradictions = detectContradictions(candidate, existingFacts);

  if (contradictions.length === 0) {
    return { action: 'allow' };
  }

  logContradiction(candidate, contradictions);

  return {
    action: config.contradictionMode, // 'warn', 'block', or 'supersede' (§9.5)
    reason: formatContradictionReport(contradictions),
  };
}
```

The `detectContradictions` function uses entity-level comparison: it extracts named entities (dates, names, project identifiers, status labels) from both the candidate and existing facts, then checks for direct conflicts (same entity, different value). This is a conservative approach — it catches explicit contradictions ("meeting on Wednesday" vs "meeting on Thursday") but not subtle semantic drift. Semantic drift detection remains future work.

6.3 Modes: Warn, Block, Supersede

The gate operates in one of three modes, configured per-plugin:

Warn mode: The contradiction is logged to `contradictions.jsonl` with the candidate chunk, the conflicting existing facts, and a timestamp. The write proceeds. The agent (or its operator) can review the log to identify patterns of self-corruption.

Block mode: The write is refused. The candidate chunk is not promoted to MEMORY.md. The blocked write is logged with the same detail as warn mode, plus a `blocked: true` flag. This is safer than warn but risks losing legitimate updates when the detector produces false positives.

Supersede mode (§9.5): The contradiction is resolved rather than logged or blocked. The prior fact's validity interval is closed and the new fact opens its own — both rows are retained,

so nothing is lost. This is a non-lossy resolution; §9.5 describes the mechanism and its grounding in memory-core's bi-temporal substrate.

Warn is the conservative default because block is lossy and supersede changes live behavior; §9.5.4 examines when the default should change.

6.4 Relationship to the Write Model and Identity Layer

The contradiction gate sits at the intersection of two architectural concerns.

The event-sourced **write model** treats memory writes as cache promotions — content moves from the ephemeral context window to durable storage. The contradiction gate adds a validation step to this promotion, analogous to a database constraint check before an INSERT.

The **identity-and-consistency layer** maintains agent identity across sessions by keeping its persona record coherent. The contradiction gate protects not just the persona but the entire fact base from inconsistency. Where the identity layer guards the "who am I?" question, the contradiction gate guards the "what do I know?" question.

7. Feature 4: Compaction-Aware Capture

7.1 The Gap: Messages Used Once and Never Queried Are Lost

Pointer-based compaction replaces evicted messages with time-range markers containing topic hints. The pointers are enough for the agent to know that "something about cron debugging happened between 14:00 and 15:30" — but the actual content (the exact error message, the specific fix attempted, the user's aside about a related problem) is gone from the context window.

Nightly dreaming can recover some of this content by promoting memories that were recently searched for. But dreaming promotes only what has been queried. Content used once — read, acted upon, and never referenced again — is permanently lost after compaction.

This creates a specific class of information loss:

1. User mentions a detail in passing ("by the way, the SSL cert expires next month").
2. The agent acknowledges but does not create a task or note.
3. Compaction evicts the message.
4. The detail was never queried, so dreaming never promotes it.
5. The agent has no record of the SSL certificate expiry.

For a personal assistant, these "by the way" details are often the most valuable — precisely because no one explicitly searched for them later.

7.2 Capture Protocol

The compaction-aware capture pass runs on the `before_compaction` hook, which fires immediately before memory-core evicts messages from the context window:

```
async function compactionCapture(ctx: CompactionContext): Promise<void> {
  const messagesToEvict = ctx.messages;
  const captured: CapturedChunk[] = [];

  for (const msg of messagesToEvict) {
    if (isAlreadyInCorpus(msg)) continue;     // already persisted
    if (isTrivial(msg)) continue;              // greetings, acks, etc.

    const chunk = extractChunk(msg);
```

```
    chunk.metadata.capturedAt = Date.now();
    chunk.metadata.captureReason = 'compaction';
    captured.push(chunk);
  }

  if (captured.length > 0) {
    await writeToCapturePath(captured);
    updateConceptIndex(captured); // incremental index update
  }
}
```

7.3 Triviality Filter

Not every evicted message deserves preservation. The `isTrivial` function applies three heuristics:

1. **Length threshold**: Messages under 20 tokens are likely greetings or acknowledgments.
2. **Template match**: Messages matching common conversational patterns ("ok", "got it", "thanks") are skipped.
3. **Redundancy check**: If the message content is already present in a previously captured chunk (checked via a lightweight hash), it is skipped.

The filter is deliberately permissive — it is better to capture a marginally useful message than to lose a detail that matters. Storage is cheap; lost information is expensive. The filter is nonetheless *lossy by construction*: anything it drops is gone, with no recovery path. §7.5 contrasts this with a reversible-cache design that would retain even sub-threshold content.

7.4 Integration with the before_compaction Hook

The `before_compaction` hook receives the full list of messages that memory-core is about to evict. MNEMOSYNE's capture pass processes them synchronously before returning control to memory-core, which then proceeds with eviction. The captured chunks are written to `~/.openclaw/data/mnemosyne/captured/` as individual markdown files, one per capture event.

These captured files are indexed by memory-core's FTS5 crawler on its next pass, making them available for future hybrid search and concept index ingestion. The capture thus feeds back into the retrieval pipeline: content captured during compaction becomes searchable in subsequent sessions.

7.5 Relationship to Pointer-Based Compaction and Reversible Caching

Pointer-based compaction and MNEMOSYNE's capture are complementary, not competing:

- **Pointers** tell the agent *what happened* (time range, topic hints) so it can decide whether to search for details.
- **MNEMOSYNE capture** ensures the *details exist to be found* when the agent does search.

Without capture, the pointers point to content that may no longer exist in any searchable form. Without pointers, capture would be insufficient because the agent would not know to search for captured content. The two mechanisms close each other's gaps — but only for content that survives the triviality filter.

There is an honest gap here, and the comparison to chopratejas/headroom (§2.3) names it cleanly. MNEMOSYNE's pointer-plus-capture pair is, in CCR terms, a *lossy* compress step: a pointer marker is an extreme compression of an eviction window, and any message the §7.3 filter

classifies as trivial is discarded with no cache holding the original. headroom's reversible CCR makes the opposite tradeoff — it never decides a byte is unworthy; it compresses everything and keeps the originals recoverable. The cost of MNEMOSYNE's selectivity is precisely the case where a "by the way" detail of §7.1 is shorter than the length threshold, or is paraphrased closely enough to trip the redundancy check, and is therefore filtered out and lost. A reversible-cache tier for sub-threshold content — compressing rather than discarding what the triviality filter rejects, so it remains recoverable on demand even though it is not promoted to the searchable corpus — is a candidate mechanism that would let MNEMOSYNE keep its signal-per-chunk selectivity for retrieval while closing the residual loss path. §10.5 treats it as future work alongside the capture-volume retention policy, since the two interact: a reversible tier changes what "stored" volume means.

8. Implementation

8.1 Plugin Manifest

```
{
  "id": "tinkerclaw-memory-enhancements",
  "name": "MNEMOSYNE - Memory Enhancement Suite",
  "version": "0.1.0",
  "description": "Concept index, task-conditioned scoring, contradiction gate, and compactio
  "entryPoint": "dist/index.js",
  "configSchema": {
    "conceptIndex": {
      "enabled": { "type": "boolean", "default": true },
      "confidenceThreshold": { "type": "number", "default": 0.4 },
      "pruneThresholdDays": { "type": "number", "default": 90 }
    },
    "taskScoring": {
      "enabled": { "type": "boolean", "default": false },
      "weight": { "type": "number", "default": 0.3 }
    },
    "contradictionGate": {
      "enabled": { "type": "boolean", "default": false },
      "mode": { "type": "string", "enum": ["warn", "block", "supersede"], "default": "warn"
    },
    "compactionCapture": {
      "enabled": { "type": "boolean", "default": true },
      "trivialityThresholdTokens": { "type": "number", "default": 20 }
    }
  }
}
```

The `configSchema` field is mandatory in current memory-core: a plugin that omits it fails to load entirely. This is a load-time contract, not a runtime check, so a missing schema blocks the plugin before any hook can register.

8.2 Hook Registrations

All four hooks are registered in `activate()` and individually gated by their `enabled` config flag. Hook registration order:

1. `conceptIndexLookup` on `retrieval_pre` — runs first, may short-circuit.
2. `taskConditionedScoring` on `retrieval_pre` — runs second, only if concept index did not fire.
3. `contradictionGate` on `before_message_write` — synchronous, returns allow/warn/block/supersede.
4. `compactionCapture` on `before_compaction` — synchronous, must complete before eviction proceeds.

8.3 Deployment Status

Feature	Status	Notes
Concept index	**Live**	anchors.json persisted, lazy loading, nightly rebuild
Compaction-aware capture	**Live**	Capturing to `~/.openclaw/data/mnemosyne/captured/` FTS5 indexing confirmed
Task-conditioned scoring	Scaffolded	Hook registered but returns early (no task descriptor extraction yet)
Contradiction gate	Scaffolded	Hook registered; today it injects passive contradiction warnings into the retrieval pack rather than blocking, and entity-level conflict detection is not yet wired
Bi-temporal supersede (§9.5)	Core-ready, plugin-pending	memory-core ships the validity columns, search predicate, and supersede writer; MNEMOSYNE must route the gate's decision into them
Reproducible eval harness (§10.4)	Not built	No latency or accuracy benchmark exists; the gap is now a first-class limitation, not a footnote

8.4 Lines of Code

File	LOC	Purpose
`index.ts`	85	Plugin entry, hook registration, config loading
`concept-index.ts`	210	Map data structure, ingestion, lookup, pruning
`task-scoring.ts`	65	Score modifier, task descriptor interface
`contradiction-gate.ts`	120	Write interception, contradiction detection stub

File	LOC	Purpose
`compaction-capture.ts`	145	Before-compaction capture, triviality filter
`utils.ts`	55	Tokenizer, hash, file I/O helpers
Total	**680**	

8.5 Deployment

```
# Build the plugin
cd ~/.openclaw/workspace/extensions/tinkerclaw-memory-enhancements
pnpm build

# Enable in openclaw.json
# plugins.allow: ["tinkerclaw-memory-enhancements"]
# plugins.entries: [{ "id": "tinkerclaw-memory-enhancements", "enabled": true }]

# Restart gateway (SIGUSR1 does not reload ES modules)
openclaw-restart --full
```

9. Discussion

9.1 Why a Unified Plugin

The four features could have been implemented as four separate plugins. A unified plugin was chosen for three reasons:

1. **Shared data structures.** The concept index is updated by both the ingestion pipeline (Feature 1) and the compaction capture pass (Feature 4). Splitting them into separate plugins would require inter-plugin communication or duplicated state.
2. **Consistent configuration.** A single `configSchema` with four feature flags is simpler to manage than four separate plugin configurations.
3. **Marketplace distribution.** ClawHub distributes plugins as atomic units. A user searching for "memory enhancements" finds one package with four toggles rather than four packages that must be installed in the right combination.

9.2 Relationship to Upstream Memory-Wiki

OpenClaw's upstream is developing a "Memory-Wiki" feature that combines memory-core with a wiki-style editor for structured knowledge management. Memory-Wiki targets enterprise teams managing shared knowledge bases.

MNEMOSYNE targets a different use case: a single personal assistant whose memory grows organically through conversation, nightly consolidation, and imported archives. The two systems are complementary:

- Memory-Wiki assumes well-structured, human-curated content. MNEMOSYNE handles messy, auto-generated content.
- Memory-Wiki optimizes for collaborative editing and version history. MNEMOSYNE optimizes for single-agent consistency and retrieval speed.
- Memory-Wiki provides a UI for manual fact management. MNEMOSYNE provides automated gates that reduce the need for manual review.

9.3 The Complementarity Argument

Enterprise-wiki and personal-assistant are not just different products — they produce different *failure modes*. An enterprise wiki fails when it returns outdated information (solved by version history and review processes). A personal assistant fails when it contradicts itself (solved by contradiction gating and supersession) or loses context (solved by compaction capture). Designing for one set of failures does not address the other.

This divergence justifies maintaining MNEMOSYNE as a separate enhancement layer rather than lobbying for its features to be merged into the memory-core mainline. The mainline maintainers optimize for the general case; MNEMOSYNE optimizes for the personal-assistant edge.

9.4 Potential ClawHub Distribution

The plugin architecture and configSchema compliance make MNEMOSYNE a candidate for ClawHub distribution once the next release reaches stability. Distribution requirements:

- All four features must be individually toggleable (already implemented via config flags).
- The plugin must not crash when memory-core hooks are unavailable (already handled by failure isolation).
- Documentation must include configuration examples for common scenarios.

The wider ClawHub-adjacent ecosystem now sets an evidence bar as well as an interface bar: distributed agent systems such as chopratejas/headroom and coreyhaines31/marketingskills ship per-system reproducible assertion suites alongside their code, and a plugin that claims a constant-time retrieval property without a runnable benchmark will read as weaker than its measured neighbors. §10.4 treats closing that gap as a distribution prerequisite, not a nicety.

9.5 Bi-Temporal Validity: From Contradiction-Logger to Contradiction-Resolver

The contradiction gate (§6) is a *single-valued-present* operation: it decides allow / warn / block over a corpus implicitly assumed to be a flat snapshot of "what is true now." That assumption is exactly what the §6.1 *stale-override* and *semantic-drift* threats violate. Bi-temporal validity replaces *overwrite/block* with *supersede*: a contradicting write neither destroys the prior fact (lossy — the reason warn is the conservative default) nor silently keeps both (the calendar-conflict failure of §6.1). Instead it **closes the prior fact's validity interval and opens the new one**, so the corpus carries the full history and every retrieval can ask "valid-at-now" versus "valid-at-time-T." This upgrades the gate from a *logger of conflicts* into a *resolver of conflicts*, and it makes task-conditioned scoring (§5) honest: a debugging query asking "what's the cron schedule" gets the currently-valid value, not the superseded one written three weeks ago.

Figure 2 illustrates a single fact's life across a supersede event: fact A is written with an open interval, a contradicting fact B closes A's interval and opens its own, and a `current` query returns only B while a `valid-at` query before B's write still returns A.

9.5.1 Division of Labor: The Index Owns the Schema, MNEMOSYNE Owns the Behavior

The schema work belongs to the concept-index subsystem; the write/retrieve *semantics* belong to MNEMOSYNE. The substrate has landed in memory-core and can be verified directly in the codebase:

- The `chunks` table carries `validity_start`, `validity_end`, `ingestion_time`, and `superseded_by` columns, added idempotently via `ensureColumn()` in `memory-schema.ts`, with a covering

index `idx_chunks_validity` on `validity_end` so the hot-path "current" filter does not table-scan. (`validity_end IS NULL` means currently-valid / unbounded; back-filled legacy rows are NULL and so remain retrievable in the default mode.)
- `StoredChunk` (`storage/types.ts`) carries the matching optional fields `validityStart`, `validityEnd`, `ingestionTime`, and `supersededBy`, and `SearchParams` gains `temporalMode: "current" | "valid-at" | "all"` plus `asOfTime`.
- The read path is already temporal-aware: `manager-search.ts` exports `temporalPredicate()`, threaded into the vector, FTS, and hybrid SELECTs — `current` appends `validity_end IS NULL OR validity_end > now`, `valid-at` binds `validity_start <= asOf AND (validity_end IS NULL OR validity_end > asOf)`, and `all` drops the predicate entirely.

So MNEMOSYNE's claim is not gated on a future migration. The substrate exists. What MNEMOSYNE owns is routing the contradiction gate's decision into it and keeping task scoring validity-aware. The unmeasured part is the *cost* of the always-on `validity_end` filter under production load; §10.4 lists it.

9.5.2 The Supersede Write Decision

MNEMOSYNE extends the gate's `WriteDecision` with a `supersede` variant alongside `warn`/`block`. memory-core already ships the executor: `engram/supersede-writer.ts` exposes `decideSupersede(db, candidate, mode = "supersede")`, which finds the currently-valid (`validity_end IS NULL`) prior chunk(s) at the candidate's identity and returns `{ action, supersedes: ChunkId[], reason }`; and `applySupersede(db, decision, candidate)`, which — only in `supersede` mode — calls `invalidate()` once per superseded chunk. `invalidate()` (in `temporal-invalidation.ts`) stamps `validity_end` and `superseded_by` on rows that are *still open* (`WHERE ... AND validity_end IS NULL`), never deleting and never resurrecting a closed row. The behavior is non-lossy: the prior row stays queryable in `all` / `valid-at` mode.

```
type WriteDecision =
  | { action: "allow" }
  | { action: "warn",   supersedes, reason }
  | { action: "block",  supersedes, reason }
  | { action: "supersede", supersedes: ChunkId[], reason }   // resolves the conflict

function contradictionGate(ctx):
    candidate = ctx.chunk
    decision  = decideSupersede(db, candidate, config.contradictionMode)  // memory-core
    if decision.action == "allow": return decision
    logContradiction(candidate, decision)                  // still append to contradictions
    return decision                                        // 'warn' | 'block' | 'supersede

# On apply (supersede mode only): close each prior interval as of the candidate's validity_s
applySupersede(db, decision, { id: candidate.id, validityStart: now })
```

`decideSupersede` defaults its `mode` argument to `"supersede"`. `applySupersede` exposes an `onClosed(closedIds, reason)` callback that fires only when an interval is actually closed; the production caller (`manager-embedding-ops.ts`) uses that seam to emit a `fork.prefrontal.trailEvent` of kind `recipe-supersede`, so the UI renders a supersede icon. The MNEMOSYNE side is the configuration default it advertises — see §9.5.4.

9.5.3 Validity-Aware Task Scoring

Because the read-path predicate (§9.5.1) already excludes superseded chunks in `current` mode, the scoring change is small. In `valid-at` / `all` modes, where historical chunks *are* returned,

`taskConditionedScoring` (§5.3) must not let a stale-but-textually-similar fact out-rank the current one on task bias alone. The guard down-weights — not zeros — a candidate whose validity interval does not contain the query's `asOf`:

```
function taskConditionedScoring(ctx):
    if not ctx.taskDescriptor: return
    asOf = ctx.asOfTime ?? now
    for cand in ctx.candidates:
        rel = computeTaskRelevance(cand.metadata, ctx.taskDescriptor)
        if cand.validityEnd != null and cand.validityEnd <= asOf:
            rel = rel * HISTORICAL_TASK_DAMP    # down-weight, don't zero
        cand.score += TASK_WEIGHT * rel
```

The scorer already carries the right primitive: its event-level scoring applies a fixed `SUPERSESSION_DISCOUNT` (0.1) to any event a newer version has replaced, so a superseded fact is multiplicatively penalized rather than dropped. The validity-aware guard generalizes that constant into a mode-sensitive damp keyed on the query's `asOf`, so the penalty applies only in `valid-at` / `all` retrieval, not unconditionally.

A related correctness note: recency-based importance should derive its recency tier from `validity_start`, not the row's last-updated timestamp — a fact reinstated after being superseded became *true again* at its new `validity_start`, and that is the date a recency bonus should reflect. `ingestion_time` is stamped once at first insert and never on re-touch, so it stays the honest "when did the system first learn this" clock.

9.5.4 The One Genuine Product Decision

memory-core defaults `decideSupersede` to `supersede` mode, but MNEMOSYNE's published plugin defaults `contradictionGate.mode` to `warn` for backward compatibility (§8.1). Whether the *plugin* default should flip to `supersede` is a product call, not a code-derivable one. The argument for flipping is that supersede is strictly safer than block — nothing is destroyed — and resolves rather than merely logs the §6.1 failure. The argument against is that it changes live behavior for every existing installation, and a false-positive contradiction would close the interval of a fact that was actually still valid (recoverable via `all`-mode query, but no longer surfaced by default). The conservative path — ship `supersede` opt-in, observe the `contradictions.jsonl` false-positive rate, then consider flipping the default later — is the one this paper recommends. Note that "observe the false-positive rate" is itself a measurement the paper does not yet report, which is the same evidence gap §10.4 names for the retrieval claims.

9.5.5 What Is Not Yet Built

Two pieces of the design remain unimplemented and are stated honestly as future work:

1. **Cascade retraction.** `temporal-invalidation.ts` is explicit that there is *no cascade in the current build*: invalidating a fact does not invalidate facts that depended on it. A `depends_on` edge plus a topological, visited-set-guarded cascade (to terminate on cyclic dependencies) is a future item, and it overlaps the link-graph work — the `backlinks` table is already in the schema.
2. **Fact-level (triple) validity.** The landed schema is *chunk-level*: one validity interval per chunk row. A true fact-level bi-temporal graph — `(subject, predicate, object, valid_from, valid_to)` — is richer but a larger build, and is left as an open granularity question rather than claimed.

A third design question — whether the supersede boundary uses *ingestion time* (simple, what the current executor assumes via the candidate's `validity_start`) or an *extracted event time* from the content (correct for "the meeting moved to Thursday," but requires reliable temporal entity extraction) — is unresolved. Using ingestion time can mis-order the §6.1 stale-override scenario when an old memory is consolidated after a newer one; event-time extraction is the principled fix and a candidate for a later release.

9.5.6 Test Plan for the Validity Layer

The substrate ships with tests; the plugin-side behavior needs its own, matching the subsystem's vitest convention (`pnpm vitest src/memory/...`). The cases that pin the supersede semantics:

- **Supersede closes the prior interval.** Insert fact A (`validity_end = NULL`); write a contradicting fact B in `supersede` mode. Assert A's `validity_end` equals B's `ingestion_time` and B's `validity_end` is NULL, and that both rows are still present — no `DELETE`.
- **Current-mode filter.** `search(temporalMode: 'current')` returns B, not A.
- **Valid-at query.** `search(temporalMode: 'valid-at', asOfTime: T)` with `A.validity_start < T < B.validity_start` returns A — the "what was true then" path.
- **All-mode.** `search(temporalMode: 'all')` returns both A and B.
- **Scoring honesty.** In `valid-at` mode a superseded chunk that matches the task descriptor scores below the current chunk despite higher textual similarity (`HISTORICAL_TASK_DAMP` applied).
- **Index guard.** `PRAGMA index_list(chunks)` includes `idx_chunks_validity`, since every `current`-mode search filters on `validity_end`.

The end-to-end target is the §6.1 calendar-conflict reproduction: a recipe-conditioned query at `asOf = now` never surfaces the superseded "Wednesday" fact once the "Thursday" write has passed through the gate in `supersede` mode. These are correctness tests; they pin behavior but do not measure latency, which §10.4 calls out separately.

10. Limitations and Future Work

10.1 Hook Dependency on Memory-Core

MNEMOSYNE's four hooks (`retrieval_pre`, `before_message_write`, `before_compaction`, `after_compaction`) are not part of memory-core's public API. They exist as extension points but are not contractually stable. A refactor could rename, remove, or change the signature of any hook.

Mitigation: A merge-guardian automation checks for hook availability after every memory-core merge. If a hook disappears, the guardian logs a warning and MNEMOSYNE degrades gracefully — the missing feature is disabled, the others continue.

10.2 Ingestion Wiring

The concept index is populated during nightly consolidation. Real-time ingestion — indexing new memories as they arrive during the session — is implemented as an incremental update in `compactionCapture` but is not yet triggered from all memory write paths. Full real-time ingestion is planned for the next release.

10.3 Contradiction Detection Depth

The current contradiction detector uses entity-level comparison: same entity, different value. This catches explicit contradictions but misses:

- **Semantic contradictions**: "The project is going well" vs "The project has critical blockers."
- **Implication contradictions**: "All tests pass" followed by "Feature X is broken" (the second implies at least one test should fail).

Temporal contradictions — facts that were true at time T but not at T+1 — are no longer a detection gap so much as a *resolution* one: the bi-temporal validity layer (§9.5) is the mechanism that resolves them once the gate routes through `supersede`. Deeper semantic and implication detection using LLM-assisted comparison is planned for a later release.

10.4 Benchmarks: The Central Evidence Gap

This is the paper's most consequential limitation, and the surrounding ecosystem makes it sharper than it was at v1.0. MNEMOSYNE's headline claim — O(1) concept-index retrieval that beats degrading hybrid search at scale (§4.7) — has never been benchmarked. The complexity analysis predicts constant-time retrieval, but no wall-clock measurement on the production corpus (thousands of concepts) exists to quantify the actual speedup, and the same is true of the bi-temporal read path: §9.5.1 notes a covering index that should keep the always-on `validity_end` filter off the table-scan path, but the per-query overhead has not been measured under production load. The supersede false-positive rate (§9.5.4) and the capture's effect on answer accuracy are likewise unmeasured.

The comparison to chopratejas/headroom (§2.3) is what makes this gap a liability rather than a routine "future work" line. headroom ships a runnable benchmark suite (`python -m headroom.evals`) that reports token savings and an accuracy delta against named public datasets (GSM8K, TruthfulQA, SQuAD, BFCL). coreyhaines31/marketingskills similarly ships per-skill reproducible assertion checks. When competing and adjacent systems ship runnable evidence, an unmeasured systems paper reads as under-evidenced regardless of how sound its architecture is.

The next revision has two honest paths and should commit to one:

1. **Adopt a comparable reproducible eval harness.** A fixed query set with relevance labels, measuring concept-index versus hybrid-search latency at growing corpus sizes (thousands → hundreds of thousands of chunks), the bi-temporal filter's per-query overhead, and capture's effect on end-to-end answer accuracy — the analogue of headroom's eval module for a retrieval-and-validity system. This is the path that lets the O(1) and supersede claims be stated as results.
2. **Scope the paper as architecture-only.** If the harness is out of scope for now, the speedup and overhead claims should be reframed explicitly as design predictions, not empirical findings, and the abstract and §4.7 should say so without hedging. The current text leans on "O(1)" language that reads as a measured property.

Path (1) is preferred, because a personal-agent corpus that already exists in production is exactly the realistic workload a benchmark wants, and because eval reproducibility is becoming a distribution prerequisite (§9.4), not a nicety.

10.5 Capture Volume and a Reversible Sub-Threshold Tier

Compaction-aware capture increases the total volume of stored content. Over time, the captured corpus may grow to rival the original memory corpus. A retention policy (e.g., captured chunks older than 180 days with zero retrieval hits are pruned) is planned for the next release.

A second, orthogonal direction comes directly from the headroom comparison (§7.5). MNEMOSYNE's triviality filter is lossy: sub-threshold content is discarded with no recovery path, which is precisely the "by the way" loss case Feature 4 set out to fix when the detail happens to be short or near-redundant. A *reversible sub-threshold tier* — compressing what the triviality filter rejects (in the spirit of headroom's reversible CCR) and caching it as recoverable-on-demand rather than promoting it to the searchable corpus — would close that residual loss path while preserving MNEMOSYNE's signal-per-chunk selectivity for normal retrieval. This interacts with the retention policy above: a reversible tier changes what "stored volume" means, since compressed-and-cached content costs far less than promoted-and-indexed content. Evaluating whether the recovery rate of that tier justifies its storage cost is a benchmark question, and folds into the §10.4 harness.

11. Conclusion

MNEMOSYNE demonstrates that four significant improvements to a personal-agent memory system — constant-time concept retrieval, task-conditioned scoring, write-time contradiction gating, and compaction-aware capture — can be delivered as a single plugin layered on top of an unmodified memory-core. The plugin architecture provides failure isolation, upgrade independence, and marketplace distributability. Two features are live in production; two are scaffolded and await entity extraction and task descriptor wiring. The contradiction gate's most consequential evolution — bi-temporal supersession (§9.5) — rests on a substrate that already exists: memory-core ships the validity schema, the temporal search predicate, and the supersede writer, leaving MNEMOSYNE to route the gate's decision into them. The approach validates a design principle: that personal-assistant memory diverges enough from enterprise-wiki memory to justify a separate enhancement layer rather than modification of the shared core. The work that remains is not only the two scaffolded features but the evidence itself: against ecosystem systems such as chopratejas/headroom that ship reproducible benchmarks for the eviction-loss problem from the lossless-by-caching direction, MNEMOSYNE's selective-capture-plus-validity design must earn its O(1) and supersession claims with measurement, and a reversible sub-threshold tier is the most promising way to absorb headroom's lossless property without surrendering MNEMOSYNE's signal-per-chunk selectivity.

References

- Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020.*
- Packer, C., et al. (2023). MemGPT: Towards LLMs as Operating Systems. *arXiv:2310.08560.*
- Xu, Z., et al. (2025). A-MEM: Agentic Memory for LLM Agents. *arXiv preprint.*
- Croft, W.B., Metzler, D., & Strohman, T. (2010). *Search Engines: Information Retrieval in Practice.* Addison-Wesley.
- MemoryOS. (2024). Mem0: Multi-Level Memory Architecture. *mem0.ai.*
- Carbonell, J. & Goldstein, J. (1998). The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries. *SIGIR 1998.*
- Robertson, S. & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in IR.*
- Chopra, T. (2025). headroom: Reversible Compress-Cache-Retrieve for Agent Context. Open-source project (chopratejas/headroom), Apache-2.0; reproducible eval suite and Kompress-v2-base compression model.

- Haines, C. (2025). marketingskills: A Reproducible Skills Collection for Marketing Agents. Open-source project (coreyhaines31/marketingskills).

References