Semantic Write Coordination for AI Memory Graphs

Treating duplicates as reinforcement in MCP-era multi-source memory systems

Priyadarshi Papa Gavali
KPZG LLC
conxt.dev

Preprint · June 13, 2026

Initial draft — revised June 18, 2026 to correct the aggregate duplication estimate in §6.6 for testing-induced duplication; see §6.6 and §8.6.

Abstract

AI memory layers — systems that capture decisions, preferences, and context from conversations with AI assistants and make them available to future sessions — are emerging as critical infrastructure for cross-tool developer workflows. As these systems connect to multiple AI surfaces simultaneously through the Model Context Protocol (MCP), a write-coordination problem appears that traditional consensus algorithms do not address: the same logical decision is written multiple times by independent extraction paths, producing near-duplicate but not byte-identical memory records. We call this the three-source write problem and present a deployable semantic coordination design that resolves it. The design layers a normalized content-hash fast path, a vector-similarity slow path, and optimistic locking on top of an existing memory store. It treats repeated writes as reinforcement signals rather than waste, preserving canonical content while raising confidence with each restatement. Existing techniques (hashing, embedding-based deduplication, optimistic locking, semantic deduplication pipelines) are well known individually; the contribution here is the combination of these primitives inside an MCP-era AI memory graph, with reinforcement scoring as the resolution mechanism. We additionally characterize the duplication present in the production graph and validate the design's 0.85 similarity threshold against a 149-pair labeled benchmark (F1 = 0.827), finding that exact-match coordination resolves under 4% of the dominant entity's duplication while semantic matching resolves the bulk. We describe the design, sketch the implementation deployed in the Conxt production memory graph, and discuss how the same approach generalizes to other AI memory products.

1 Introduction

AI memory layers sit between a user's AI tools and a persistent store. They capture conversations (typically through a browser extension), extract structured memories using a language-model extractor, embed them as vectors for semantic retrieval, and expose them to downstream tools through APIs such as the Model Context Protocol (MCP). Products in this category include Conxt, Mem0, Zep, Honcho, and Memdex.

As an individual user connects more AI surfaces to a single memory graph, the system necessarily develops multiple independent write paths. A capture engine writes from the browser extension. An explicit MCP tool such as `add_memory` writes when a developer invokes it from Cursor or Claude Code. A second device writes from a different physical machine. These paths do not coordinate. The result is a recurring failure mode in which the same logical decision appears in the memory graph many times, in many slightly different surface forms, with no awareness from any single writer that the others have happened.

This is not the failure mode that classical distributed consensus is designed to prevent. Raft and Paxos assume that conflicting writes target the same key and differ only in value. Conflict-free Replicated Data Types assume operations are commutative or that conflicts can be resolved by a deterministic merge func-

tion. None of these models map cleanly to AI memory, where the writes target semantically related but syntactically distinct content. A traditional consensus system would treat the ten different rows as ten valid writes. A useful AI memory system needs to recognize them as one reinforced memory.

We do not propose a replacement for classical consensus. We propose an application-layer semantic write-coordination mechanism for memory systems where conflicts are meaning-level duplicates rather than byte-level disagreements.

1.1 Contributions

- A formal description of the three-source write problem in AI memory systems, distinguishing it from the byte-level conflicts that classical consensus algorithms address.
- A deployable semantic write-coordination design that composes three known primitives — normalized content hashing, vector-similarity matching, and optimistic locking — into a single insert path tailored to MCP-era AI memory graphs.
- The framing of duplicate detection as reinforcement rather than waste, including a concrete confidence-update rule that grows quickly for the first few reinforcements and saturates thereafter.
- An empirical characterization of duplication in a live production graph, including a validated similarity threshold and a measured demonstration that exact-match coordination is insufficient.
- A reference implementation in the Conxt production memory graph, with the design released as part of the Conxt engine repository.

1.2 What This Paper Is Not

To position the contribution honestly: this paper does not claim to invent semantic deduplication, hash-based duplicate detection, vector-similarity matching, or optimistic locking. Each is well established in prior literature and industrial practice. Semantic deduplication using embeddings and cosine similarity is described, among other places, in the NVIDIA NeMo Curator documentation. The contribution here is the combination of these primitives inside the specific write-coordination problem that AI memory graphs face after the broad adoption of MCP, and the use of reinforcement scoring as the resolution mechanism in place of duplicate discard.

2 Background

2.1 AI Memory Layers

An AI memory layer is a service that persists a user's intent, decisions, preferences, coding rules, and project context across sessions with different AI assistants. The architecture is consistent across products in the category: a capture interface (typically a browser extension or IDE plugin), an extraction engine (typically driven by an LLM that reads raw turns and emits structured memories), a vector-augmented store (often pgvector on Postgres, sometimes Pinecone or Weaviate), and a retrieval interface that exposes memories to downstream AI tools.

These systems became more practically deployable in 2025 with the broad adoption of the Model Context Protocol, which standardizes how AI tools discover and call external services. Before MCP, exposing memory to Claude, Cursor, and Windsurf required three separate integrations. After MCP, a single Streamable HTTP endpoint can serve all three.

2.2 The Model Context Protocol

The Model Context Protocol defines a transport-agnostic interface between AI tools and external services. The current specification, dated June 18, 2025, defines Streamable HTTP as the primary transport for remote MCP servers, using HTTP POST for tool calls and an optional server-sent events stream for incremental responses. A compliant MCP server exposes tools (named functions with typed parameters), accepts authenticated calls from any compliant client, and returns structured results. The MCP authorization specification describes OAuth-style discovery and Bearer-token authentication for remote protected servers.

The Conxt system exposes tools such as `get_context`, `get_memories`, `add_memory`, `list_my_teams`, `get_team_memories`, and `add_team_memory` through a Streamable HTTP MCP endpoint at `mcp.conxt.dev`.

The relevance of MCP to the write-coordination problem is that it creates a second independent write path. Before MCP, the only way memories entered the graph was through the capture-and-extract pipeline. After MCP, an AI tool can call `add_memory` directly, in parallel with the capture pipeline, on the same logical conversation.

2.3 Classical Consensus and Why It Is Insufficient Here

Raft [3] and Paxos [2] provide strong consistency through leader election and quorum agreement. They assume that conflicting writes target the same key and differ only in value, and that one of the values must be chosen as canonical. Conflict-free Replicated Data Types [4] relax this by requiring operations to be commutative, so concurrent updates can merge without coordination. Vector clocks [1] order events causally rather than by wall-clock time, useful for detecting concurrent updates.

These primitives are useful and we build on them — the optimistic-locking layer of our design uses a version counter that is fundamentally a vector-clock idea. But none of them address the dominant pattern in AI memory: writes that target the same logical idea, differ in surface form, and are all individually correct. The right resolution is not to choose one over the other but to recognize that they are the same memory and treat the repetition as signal.

2.4 Prior Work on Semantic Deduplication

Semantic deduplication — detecting and removing near-duplicate documents based on embedding similarity rather than exact string match — is well established. Industrial pipelines such as NVIDIA NeMo Curator describe this approach for large language-model training corpora: embed each document, cluster by cosine similarity, keep one representative per cluster, discard the rest. Entity resolution and record linkage literature in databases has studied analogous problems in structured data for decades.

Our work differs from these in two ways. First, the goal is not corpus pruning but live write-time coordination across multiple ongoing extraction paths. Second, the resolution policy is reinforcement rather than discard — the duplicates carry information about how often a decision was restated, and we keep that information.

3 The Three-Source Write Problem

In a production AI memory system connected to multiple AI tools, the same logical decision can arrive through three distinct write paths within minutes.

3.1 Browser-Extension Capture

A browser extension captures the user's session with claude.ai, chatgpt.com, or gemini.google.com and forwards it to an extraction engine. The engine, driven by an LLM, reads the full conversation and emits structured memories: decisions, preferences, entities, open questions, coding rules, tool choices, and workflows. The user does not directly control which memories are produced; the extractor decides. A single conversation that touches a technology choice five times may yield five separate decision rows, each in slightly different prose.

3.2 Direct MCP Write

The user, working in an MCP-connected tool such as Cursor or Claude Code, explicitly invokes an MCP tool such as `add_memory` or `add_team_memory` to record a specific decision. This write is intentional, scoped, and high-confidence. But the capture extension may also be running, and the same conversation that prompted the explicit MCP write is also being captured for passive extraction. The system now has two writes describing the same decision, originating from two paths, neither of which knows about the other.

3.3 Multi-Device Sync

The same user, signed in on a work laptop and a home machine, captures overlapping conversations on each device. Each device writes to the central graph independently. Without device-aware coordination, neither write knows the other has happened until both are committed.

3.4 Empirical Pattern

These paths produce a specific failure mode that we observed directly on the Conxt production graph. The single decision *Use Tailwind for the marketing site* appeared, within approximately one hour, as ten distinct rows in the `memory_records` table. Variations included *Using Tailwind for the KPZG marketing site*, *We're using Tailwind for the marketing site*, *Use Tailwind for the marketing site instead of CSS modules*, and seven additional restatements differing in punctuation, word order, or framing. Each row was a valid extraction by itself. Together they were noise.

The traditional consensus question — which of these ten writes is the canonical one? — is the wrong question. All ten are correct. The right question is how to recognize that they describe one decision, preserve a single canonical form, and treat the other nine as evidence that the decision matters more than a memory mentioned only once.

4 Semantic Write Coordination: Three Layers

We propose a three-layer design. Layers can be deployed independently. Layer 1 alone resolves the dominant duplicate case from same-source captures and multi-path writes by a single user; Layers 2 and 3 are needed only when multi-device or multi-user concurrency becomes operationally significant.

4.1 Layer 1: Content-Hash and Vector-Similarity Coordination

Before inserting a memory, the engine performs two checks in sequence. First, it computes a normalized SHA-256 content hash and looks for an exact match in the user's existing memories. If an exact match exists, the engine reinforces the existing row rather than inserting a new one. If no exact match exists, the engine computes a vector embedding of the new content and performs a cosine-similarity search against the user's recent memories. If any existing memory exceeds a similarity threshold of 0.85, the engine reinforces that memory instead of inserting a new row.

4.1.1 Normalization

Normalization converts memory content to a canonical form before hashing so that surface variations do not produce different hashes. The normalization function lowercases the text, collapses internal whitespace to single spaces, strips leading and trailing whitespace, removes trailing punctuation, and for structured JSON content sorts keys alphabetically before serialization. *We're using Tailwind for the marketing site.* and *we're using tailwind for the marketing site* produce the same normalized hash.

4.1.2 Threshold Selection

We use 0.85 cosine similarity as the default threshold. Unlike earlier drafts, this value is no longer chosen by inspection alone: it is the F1-optimal threshold on a 149-pair labeled benchmark drawn from the Conxt graph (§6.7). At 0.85, semantically equivalent restatements ("use Tailwind for marketing" vs. "use Tailwind on the marketing site") match, while semantically related but distinct decisions ("use Tailwind for the marketing site" vs. "use Tailwind for the dashboard") do not. The threshold remains configurable per deployment and should be re-tuned to the host system's embedding model and memory-type mix; §6.7 reports the full precision/recall/F1 curve that justifies 0.85 for our embedding model, and §8.5 discusses transfer to other domains.

4.1.3 Reinforcement

On a hash or vector match, the engine performs three updates atomically in a single transaction: increment `reinforcement_count`, update `last_seen` to the current timestamp, and increment `version`. The original content is preserved; subsequent restatements do not overwrite the canonical form. The choice to preserve the canonical content is deliberate. Recent work on LLM-driven memory consolidation has shown that continuous rewriting of memories by language models can introduce drift and factual errors over time. Keeping the canonical text immutable, while letting `reinforcement_count`, `last_seen`, and confidence evolve, separates the durable evidence from the evolving signal.

4.2 Layer 2: Optimistic Locking

For cases where two writers genuinely intend to update the same memory — typically multi-device scenarios where both devices have read the same row and now both want to update it — we use optimistic locking via a version column on `memory_records`. Every update succeeds only if the version supplied by the writer matches the current version in the database. Otherwise the update affects zero rows, and the writer must re-read the current state and decide whether to retry, merge, or surface a conflict.

```
UPDATE memory_records
SET content = $1, version = version + 1
WHERE id = $2 AND version = $3;
-- if rowcount = 0, a conflict has occurred
```

The choice between retry and surface-conflict depends on the source of the update. Routine reinforcement updates from Layer 1 retry automatically. Updates that change the memory's canonical content surface as conflicts and route to Layer 3.

4.3 Layer 3: Conflict Resolution Policies

For genuinely conflicting writes — two team members making opposite decisions about the same topic in independent AI sessions — we offer three configurable resolution policies. The policy is selected per memory type or per team.

- **Last-write-wins.** The most recent timestamp wins. Adequate for single-user, multi-device deployments where divergence is rare and recovery is cheap.
- **Merge and review.** Both memories are preserved, both flagged with `conflict_state` set to `merged`, and a `merged_from` array on each row references the other. The conflict is surfaced for human review through the dashboard. Appropriate for team-tier deployments where genuine disagreement should not be silently resolved.
- **Source priority.** Explicit MCP writes from `add_memory` or `add_team_memory` take precedence over passive extraction-pipeline writes. Explicit MCP writes provide a stronger intent signal than passive extraction; this policy treats that intent signal as authoritative.

5 Reinforcement as a Signal

Treating duplicates as reinforcement rather than waste reframes the problem. A memory that has been captured five times across three sessions is not a redundant artifact; it is a stronger signal of the user's actual preferences than a memory captured once. Traditional semantic deduplication pipelines discard the duplicates; we keep the information they carry.

5.1 Confidence Curves

Each memory in the Conxt graph carries a confidence value in the closed interval from zero to one. We update confidence on reinforcement using a logarithmic curve that grows quickly for the first few reinforcements and slowly thereafter, capped at 0.95 to leave room for explicit human verification to reach 1.0.

$$\text{confidence}_{\text{new}} = \min\left(0.95,\ \text{confidence}_{\text{old}} + 0.05 \cdot \log(1 + \text{reinforcement\_count})\right)$$

The first reinforcement raises confidence noticeably; the tenth raises it almost not at all. This matches the intuition that the first restatement of a decision is strong evidence, while the tenth adds little new information.

5.2 Decay

Reinforcement raises the question of whether memories should decay if they go unreinforced. Three positions are defensible. The first is no decay: memories persist indefinitely until explicitly archived. The second is time-based decay: confidence gradually drops as memories age. The third is retrieval-based decay: memories that are never returned by retrieval queries lose confidence faster than memories that are frequently surfaced. We currently take the first position in Conxt. Whether decay improves retrieval quality is an empirical question, and is left for future work.

6 Implementation in Conxt

We have implemented the design described above in the Conxt production memory graph, which is built on Supabase Postgres with pgvector for vector storage and a FastAPI engine on Railway for the extraction and MCP services. The reference implementation is part of the Conxt engine. At the time of the measurements reported in this section, the one-time consolidation migration had been staged but not yet executed against the production graph; every record therefore still carried `reinforcement_count = 1`, giving the clean pre-intervention baseline analyzed below.

6.1 Schema Changes

The following columns are added to `memory_records`. All changes are guarded with `IF NOT EXISTS` and are reversible.

Column	Type	Purpose
`content_hash`	TEXT	Normalized SHA-256 for exact-match fast path
`reinforcement_count`	INTEGER	Number of times re-observed
`last_seen`	TIMESTAMPTZ	Most recent reinforcement timestamp
`version`	INTEGER	Optimistic-locking version counter
`device_id`	TEXT	Origin device, for multi-device tracking
`conflict_state`	TEXT	One of `clean`, `conflict`, or `merged`
`merged_from`	UUID[]	IDs of memories merged into this one

A composite index on `(user_id, content_hash)` backs the exact-match fast path. Existing duplicates can be collapsed during migration using row identifiers (`ctid` in Postgres) rather than `min(id)`, since the primary key is a UUID and `min()` does not order UUIDs meaningfully.

6.2 The dedup_or_insert Function

The core entry point, `dedup_or_insert`, is intentionally decoupled from existing match-and-insert paths so that it can be unit-tested in isolation and rolled out incrementally.

```
async def dedup_or_insert(user_id, content, memory_type, ...):
    h = normalized_hash(content)
    if existing := await find_by_hash(user_id, h):
        await reinforce(existing.id)
        return existing.id, 'reinforced_hash'
    emb = await embed(content)
    if existing := await match_recent(user_id, emb, 0.85):
        await reinforce(existing.id)
        return existing.id, 'reinforced_vector'
    new_id = await insert(user_id, content, h, emb, ...)
    return new_id, 'inserted'
```

The function returns both the resulting memory ID and a label describing what happened, allowing the caller to log the outcome and downstream consumers to distinguish first writes from reinforcements.

6.3 Migration Strategy

Migration proceeds in three phases. First, the new columns are added with defaults that preserve existing behavior (`content_hash` is NULL, `reinforcement_count` starts at 1, `version` starts at 1). Second, a background job backfills `content_hash` for existing rows by computing the normalized hash of each row's content. Third, a one-time consolidation query collapses existing duplicates by grouping rows on `(user_id, content_hash)` and keeping the row with the earliest `created_at` as canonical. Reinforcement counts on the canonical row are set to the size of the duplicate group.

6.4 Baseline Measurements

We measured the Conxt production memory graph immediately before deploying the design described above. At the snapshot used throughout this section, the graph contained 4,188 active memory records (`deleted_at IS NULL AND is_archived = false`) belonging to the lead author, distributed across seven memory types and four write paths. Because the graph receives continuous writes, all counts in this section reflect a single snapshot; counts taken minutes apart differ by a few rows.

The four write paths confirm the multi-source pattern of §3 on live traffic, heavily weighted toward a single capture surface:

Source	Records	Share
Claude (extension)	3,594	85.8%
Gemini (extension)	334	8.0%
ChatGPT (extension)	242	5.8%
MCP (`add_memory`)	18	0.4%

By type, the distribution is heavily skewed toward entities, which is the type the LLM extractor produces most aggressively:

Type	Records
entity	1,684
tool_choice	566
decision	526
open_question	470
preference	398
workflow	370
coding_rule	174

This skew is relevant because entities are also the type most prone to duplication: the same project, tool, or organization is referenced repeatedly across sessions. The graph has grown roughly 24% since an earlier internal audit (then 3,373 records) without any deduplication pass having run — every record still carries `reinforcement_count = 1` — so the snapshot is a clean pre-intervention baseline.

6.5 Embedding Integrity

Before any similarity analysis is meaningful, the embedding column must be trusted, and it cannot be assumed clean. An integrity audit of the snapshot classified 371 of the 4,188 records (8.9%) as carrying an unusable embedding: 18 with a NULL vector and 353 with a degenerate vector (all-zero or containing NaN components). These records are silently unretrievable — any cosine-similarity query against them returns NaN — and they actively corrupt duplicate counts. Because PostgreSQL orders NaN as greater than every real number, an unguarded similarity filter (`sim >=` τ) admits every corrupt record at every threshold. An initial naive sweep over the raw column reported 135,949 candidate duplicate pairs; after excluding the 371 corrupt rows the true count fell to 28,995, an inflation of roughly $4.7\times$ attributable entirely to NaN ordering.

We report this as a first-class finding rather than a preprocessing footnote. A deduplication or write-coordination pipeline operating on a real production graph must be robust to embedding corruption on two counts: corrupt vectors poison similarity-based duplicate detection, and they represent memories that are invisible to semantic retrieval and therefore lost to the user. The remaining analysis is performed over the 3,856 records with valid embeddings present in the snapshot export (the small difference from $4,188 - 371$ reflects writes between the audit query and the export).

6.6 Cluster Analysis: Exact-Match Is Insufficient

A grouping query over exact content descriptions — the method available to the content-hash fast path of Layer 1a — identifies a long tail of duplicate clusters, the largest containing 27 rows. Taken alone, this number understates the problem, and it understates it using precisely the exact-match method this paper argues against.

Embedding-based clustering at the $\tau = 0.85$ threshold tells a different and far larger story. Consider the single most duplicated entity, the Conxt product itself. Counting by entity name, 734 records describe the entity "Conxt," across 585 distinct natural-language descriptions — from "Cross-AI memory layer SaaS that works across ChatGPT, Claude, Gemini, and Copilot" to hundreds of near-equivalent rephrasings differing in detail, word order, or framing. The consequence is stark:

- Exact-content grouping (Layer 1a, the hash path) would emit up to 585 separate records for this one entity, its largest exact group capturing only 27. It resolves under 4% of the entity's duplication.
- Embedding similarity at $\tau = 0.85$ (Layer 1b, the vector path) collapses 604 of these records into a single cluster, resolving the bulk.

This 27-versus-604 gap on the same entity is the central empirical justification for the design. The content-hash fast path is cheap and catches the truly identical restatements, but the overwhelming majority of real duplication in an AI memory graph is paraphrastic and invisible to it. The vector path is not an optimization; it is doing the load-bearing work.

Aggregated across the graph at $\tau = 0.85$, the 28,995 valid candidate pairs resolve, under single-linkage clustering, into 409 clusters absorbing 2,798 records, of which 2,389 are redundant (cluster members beyond the canonical first). Because precision at $\tau = 0.85$ is 0.765 (§6.7), the precision-adjusted estimate of truly redundant records is roughly 1,830 — about 48% of the valid-embedding graph; at the more conservative $\tau = 0.92$ (precision 0.867) the adjusted estimate is about 36%. Between a third and a half of the graph is recoverable as reinforcement signal rather than stored as distinct rows.

Redundancy concentrates in entities, consistent with §6.4:

Dominant type	Clusters	Redundant records
entity	93	1,390
tool_choice	43	249
decision	88	235
preference	66	182
open_question	60	167
workflow	40	94
coding_rule	19	72

Entities contribute 58% of all redundancy. This motivates a refinement to Layer 1: entity duplicates share a normalized name field and can be collapsed by a cheap name-match key before any embedding is computed, recovering the single largest source of duplication at near-zero cost, while free-text types (decisions, preferences, rules) fall back to the vector path.

A robustness check on testing-induced duplication. The lead author's account was also used during May and June 2026 to test the capture pipeline itself, including deliberately sending identical prompts to Claude, ChatGPT, and Gemini in rapid succession to validate cross-platform extraction. This activity necessarily produces near-duplicate writes by construction, independent of any organic restatement, and is a distinct phenomenon from the multi-source write problem described in §3. To quantify its effect on the aggregate redundancy estimate above, we identified all capture sessions on the lead author's account that were followed or preceded by a session from a different source within a ten-minute window, and recomputed the candidate-pair counts at $\tau = 0.85$ and $\tau = 0.92$ with the memories from these sessions excluded. The pair count at $\tau = 0.85$ fell from 34,983 to 29,368 (a 16.0% reduction); at $\tau = 0.92$ it fell from 6,229 to 5,160 (a 17.2% reduction). These counts were recomputed against the current production graph rather than the original snapshot export, and are reported here as a same-population sensitivity check rather than a direct replacement for the baseline counts above, which reflect a frozen, earlier snapshot. We treat the 48% and 36% redundancy estimates reported above as upper bounds that include a non-trivial contribution from this testing activity; a corrected estimate that excludes it, holding the clustering and precision-adjustment

methodology fixed, is reported in a forthcoming revision once the full cluster analysis has been rerun against a contamination-excluded snapshot. The entity-level finding above — that "Conxt" appears 734 times across 585 distinct wordings, with 604 resolved by semantic clustering — is unaffected by this check, as the testing sessions in question concern an unrelated test scenario (a personal finance tracker application) and do not contribute to the Conxt entity cluster.

A caution on merging. The $\tau = 0.85$ clustering above is correct for flagging candidates but unsafe to auto-merge transitively. The largest single-linkage component at $\tau = 0.85$ contains 786 records — a 604-record core of genuine "Conxt" duplicates plus roughly 180 distinct sub-entities (`conxt-dashboard`, Conxt Chrome Extension, `conxt-engine`, Conxt MCP server) chained in through name similarity. That component is highly threshold-sensitive: its size collapses from 786 to 23 (786, 497, 371, 85, 41, 23 at $\tau = 0.85, 0.90, 0.92, 0.95, 0.97, 0.99$) while the total cluster count stays stable (409 to 208), as shown in Figure 1. A true cluster of genuine duplicates would not shrink 34×; the collapse is the signature of single-linkage over-merging, where bridge records fuse otherwise-distinct topics. An auto-merge step that trusted the $\tau = 0.85$ clustering would silently collapse distinct sub-entities into the project entity. We therefore separate detection from merging in §6.7.

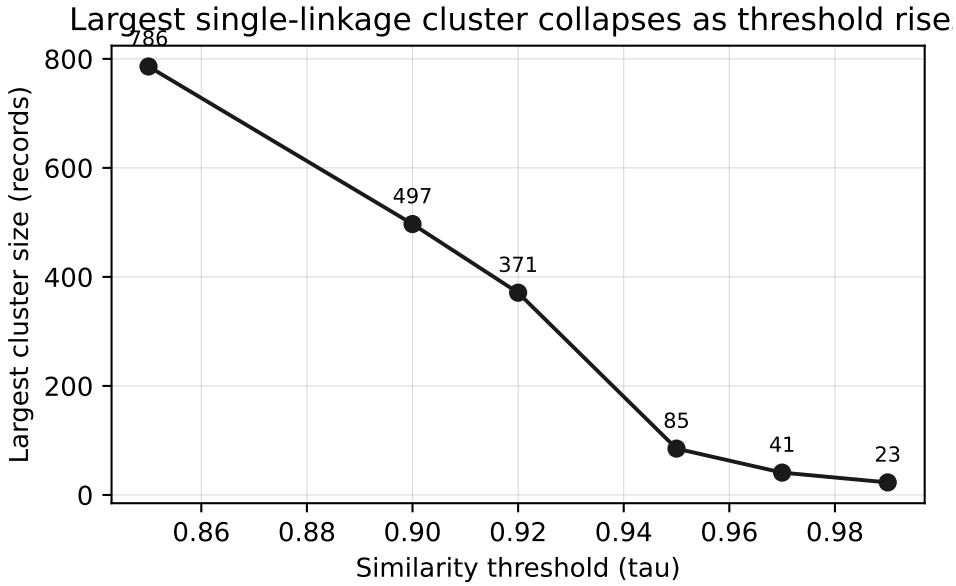

Figure 1: Single-linkage chaining. As the similarity threshold rises, the largest cluster collapses 34× (786 to 23) while the total number of clusters stays stable, the signature of transitive over-merging at low thresholds.

6.7 Threshold Validation

To select the similarity threshold defensibly rather than by assertion, we drew a stratified sample of 150 candidate pairs — 30 from each of five similarity bands spanning 0.80–1.00 — and a single expert annotator (the lead author, who owns the graph and therefore knows the ground-truth identity of every entity and decision in it) labeled each pair as duplicate or distinct. The labeling standard was *same fact, not same topic*: two records about hosting are duplicates only if they assert the same hosting choice; different tools, rules, or sub-entities are distinct. 149 of 150 pairs were labeled (two malformed rows excluded).

Band-level duplicate rates rise monotonically with similarity — 31%, 55%, 76%, 77%, 100% across the

five bands — independently confirming that the embedding model ranks true duplicates above false ones. Treating each threshold τ as a classifier (predict duplicate iff sim $\geq \tau$) against the labels yields the curve in Figure 2:

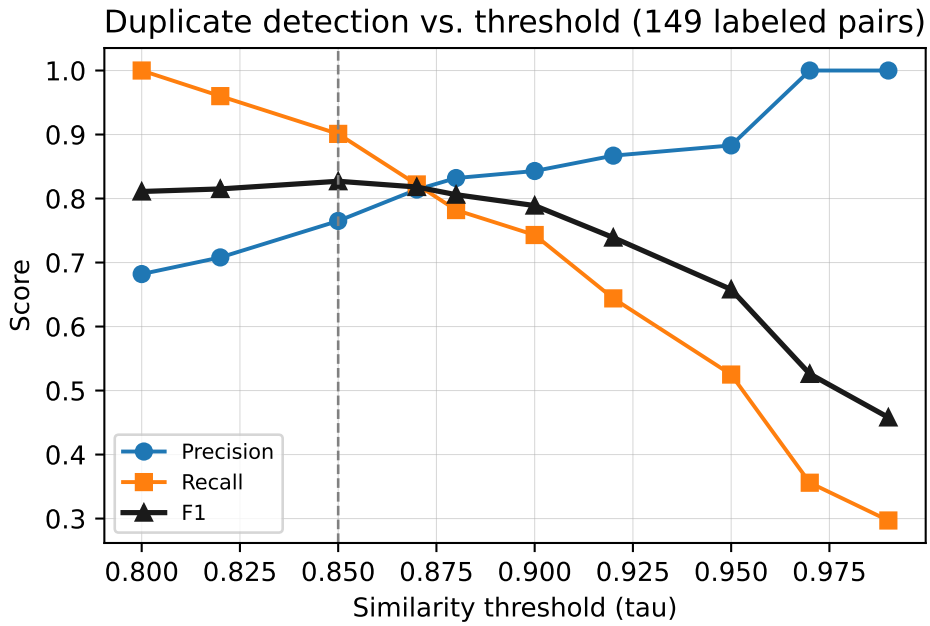

Figure 2: Duplicate detection vs. cosine-similarity threshold over the 149 manually labeled pairs. F1 is maximized at $\tau = 0.85$ (0.827); precision and recall trade off cleanly and monotonically with no inversions.

Threshold τ	Precision	Recall	F1
0.80	0.682	1.000	0.811
0.82	0.708	0.960	0.815
0.85	0.765	0.901	0.827
0.87	0.814	0.822	0.818
0.88	0.832	0.782	0.806
0.90	0.843	0.743	0.789
0.92	0.867	0.644	0.739
0.95	0.883	0.525	0.658
0.97	1.000	0.356	0.526
0.99	1.000	0.297	0.458

The curve is clean and monotonic (Figure 2): precision rises from 0.68 to 1.00 as recall falls from 1.00 to 0.30, crossing near $\tau = 0.87$. F1 is maximized at $\tau = 0.85$ (0.827), precision 0.765, recall 0.901, which is the value adopted in §4.1.2.

This result also dictates the merge policy. At $\tau = 0.85$, recall is high (0.90) but roughly one in four flagged pairs is a false positive (precision 0.765), and — per §6.6 — transitive closure at that threshold over-merges. We therefore detect at $\tau = 0.85$ but reserve automatic merging for $\tau \geq 0.92$, where precision reaches 0.867, routing the 0.85–0.92 band to confirmation: either the reinforcement mechanism of §5, which lets repeated independent captures raise confidence over time, or a lightweight human/assistant check. This flag-cheaply / merge-carefully separation is the practical policy the production data forces.

6.8 Forward-Looking Evaluation

This section reports the pre-intervention state of the graph and a validated detection threshold; it does not yet report post-migration row counts, because the one-time consolidation migration had not been executed against the production graph at the time of measurement (every record carried `reinforcement_count = 1`). The remaining evaluation — post-migration row counts, the resulting reinforcement-count distribution, insert-path latency overhead, and a false-merge audit at the $\tau \geq 0.92$ merge gate — is the subject of a follow-up technical report once the design has run in production long enough to produce stable numbers. The baseline-plus-validation result stands independently: it establishes that the problem is real and severe (§6.4, §6.6), that exact-match coordination is insufficient and semantic matching is necessary (§6.6), and that the chosen threshold is empirically optimal on the host graph (§6.7).

7 Generalization to Other AI Memory Systems

The design is not specific to Conxt. Any AI memory product with the following properties can implement an equivalent system.

- **Multiple write paths.** Some combination of a passive capture pipeline, an explicit memory-creation API, and cross-device or cross-team writers.
- **Vector embeddings of memory content.** The exact embedding model is not relevant; what matters is that cosine similarity over the embedding space reflects semantic similarity over the content.
- **A primary store that supports atomic updates with version checks.** Postgres supports this natively; MongoDB supports it through `findOneAndUpdate` with a version predicate; DynamoDB supports it through conditional writes with version expressions.

Stack-specific notes follow. On Pinecone or Weaviate, the `content_hash` and reinforcement state live in an external metadata store (often Postgres), and the vector store is consulted only for the Layer 1 similarity check. On graph-native memory systems such as those built on temporal knowledge graphs, the Layer 1 vector check is supplemented by graph-traversal checks for entity equivalence; the reinforcement signal can be applied to graph edges in addition to nodes.

8 Limitations and Future Work

8.1 Cross-User Team Consensus

The current design treats team memories as a single shared graph with last-write-wins as the default conflict policy. True team consensus, in which two members disagree about a decision and the system must surface the disagreement rather than silently choose one, requires deeper integration with the team's review workflow. The merge-and-review policy in Layer 3 is the first step toward this but does not yet propose or evaluate a UI surface for resolving the conflict.

8.2 Multi-Region Deployment

At scale, a single-region primary store becomes both a consistency bottleneck and a single point of failure. A multi-region deployment of the design requires vector clocks or a similar causal-ordering mechanism to detect concurrent writes from different regions, and a region-aware merge function. We treat this as future work; the current production deployment at Conxt is single-region.

8.3 Offline-First Capture

An offline-first browser extension that captures locally and reconciles on reconnect introduces additional conflict scenarios. Writes from an offline device may arrive out of order with respect to writes from an online device. The current design assumes online writes; offline-first capture is future work and likely requires Layer 2's optimistic-locking primitive to be supplemented by a per-device write log.

8.4 Embedding Drift

Layer 1's vector-similarity check depends on the embedding model. If the embedding model is changed, historical `content_hash` values remain valid but historical embeddings do not. A re-embedding migration is required, and the similarity threshold may need retuning. This is a general property of any system that uses embeddings as a long-lived identifier. The integrity audit of §6.5 is relevant here: a re-embedding migration is also the natural point to repair the 8.9% of records whose stored vectors are currently corrupt.

8.5 Threshold and Domain Sensitivity

The 0.85 threshold is validated on the Conxt graph (F1 = 0.827 against a 149-pair labeled benchmark, §6.7), but that validation is single-annotator and single-domain. Whether the same threshold transfers to memory graphs in legal, medical, or financial domains is an open question; in those domains false merges carry higher cost, and the threshold likely needs to be tuned upward, traded against more false splits. The validation methodology of §6.7 — stratified sampling across similarity bands with expert labeling — is the transferable artifact; the specific value is not.

8.6 Testing-Induced Duplication

The measurements in §6.4–§6.6 are drawn from a production account that, during the period the system was being built, was also used to test the capture pipeline directly: the same prompt was deliberately issued to multiple AI platforms in a single sitting to confirm that each platform's capture path worked correctly. This is a necessary part of building a multi-source capture system, but it introduces near-duplicate writes that are categorically different from the organic multi-session restatement the paper is centrally about — they are duplicates by test design, not duplicates produced by a user naturally re-explaining a decision across unrelated sessions. We identified and quantified this effect in §6.6 and found it material: candidate duplicate pairs fall by roughly 16–17% at the paper's validated thresholds once these sessions are excluded. We report the original figures alongside this correction rather than silently revising them, and recommend that any reader citing the redundancy percentages in §6.6 treat them as upper-bound estimates pending the contamination-excluded recomputation. More broadly, this suggests a methodological note for future

measurement of production AI memory graphs: any account used both for genuine work and for testing the capture system itself should have its testing sessions flagged or excluded at measurement time, ideally by tagging sessions at capture time rather than reconstructing them after the fact from timing heuristics, as we were required to do here.

9 Related Work

9.1 Classical Distributed Consensus

Lamport's work on logical clocks [1] and Paxos [2], and the Raft consensus algorithm of Ongaro and Ousterhout [3], provide the underlying primitives for ordering and atomic update. The conflict-free replicated data types of Shapiro et al. [4] inspire the merge-and-review policy in Layer 3. As discussed in §2.3, none of these algorithms address semantic duplication directly; we use their primitives but solve a different problem.

9.2 Semantic Deduplication

Industrial pipelines such as NVIDIA NeMo Curator describe semantic deduplication of large training corpora using embedding-based clustering and cosine similarity, with one representative kept per cluster. Our use of the same primitives at write time in an AI memory graph, with reinforcement rather than discard as the resolution, is the application-layer contribution of this paper.

9.3 AI Memory Systems

Several recent and contemporary systems address AI memory. Mem0 positions itself as a scalable memory layer that extracts, consolidates, retrieves, and evaluates memories over long conversations, with reported benchmarks on the LoCoMo dataset and latency and token-cost analyses. Zep and its open-source successor Graphiti use a temporal knowledge-graph approach that tracks facts over time, preserves provenance, and supports hybrid retrieval across semantic, keyword, and graph traversal. Honcho is described as an open-source memory library and managed service for stateful agents, modeling users, agents, groups, ideas, and changing entities. Memora addresses the consolidation of related updates into unified memory entries as memory grows. To our knowledge, none of these systems publish a detailed design for the specific multi-source write-coordination problem this paper addresses; the contribution here is complementary rather than competitive.

9.4 Risks of LLM-Driven Memory Consolidation

Recent work on LLM-driven memory consolidation has shown that continuously rewriting memories using a language model can introduce drift, factual errors, and loss of canonical information over time. This supports the design choice in §4.1.3 to keep canonical memory content immutable, with `reinforcement_count` and `last_seen` as the evolving signal. Letting the model rewrite memories on each reinforcement is, by this evidence, the wrong move.

10 Conclusion

AI memory layers are becoming critical infrastructure. As they connect to more AI surfaces and more devices through MCP, they accumulate near-duplicate writes from multiple independent extraction paths. Classical distributed consensus algorithms address byte-level conflicts; they do not address the meaning-level duplicates that dominate this setting.

We described the three-source write problem, presented a deployable semantic write-coordination design that composes known primitives (normalized hashing, vector similarity, optimistic locking) into a single insert path tailored to MCP-era memory graphs, and proposed reinforcement scoring as the resolution mechanism in place of duplicate discard. We measured the problem on a live production graph, showed that exact-match coordination resolves under 4% of the dominant entity's duplication while semantic matching resolves the bulk, and validated the 0.85 threshold against a labeled benchmark (F1 = 0.827). The cost of adopting this design is low: a few new columns, a single function in the write path, and a one-time migration. The cost of not adopting it grows with the graph.

AI memory systems should design for semantic write coordination and reinforcement from the first version, not retrofit them after the duplicates pile up.

References

[1] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[2] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[3] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS 2011*, volume 6976 of *Lecture Notes in Computer Science*, 2011.

[5] Anthropic and contributors. Model Context Protocol Specification, version dated June 18, 2025. `https://modelcontextprotocol.io/specification`.

[6] NVIDIA. NeMo Curator Documentation: Semantic Deduplication, 2024.

Code and Data Availability

The semantic write-coordination design is described in full in §4 and §6, including the `dedup_or_insert` insert-path pseudocode (§6.2), the schema changes (§6.1), and the migration strategy (§6.3). The reference implementation (`dedup.py`) and the schema-migration script (`dedup_migration.sql`) are part of the Conxt engine and are available from the author on request. The analyzed memory graph belongs to a single user and contains personal project data; the raw records are therefore not publicly released, but the aggregate statistics, the threshold-validation methodology, and the labeling protocol (§6.7) are reported in full to support independent replication on other graphs.