Toward a Human-Like Memory System for AI Agents

Finite Storage, Sparse Contexts, and Probabilistic Recall

Nithin Mani (Cosdata)

2025-07-10

Contents

1 Purpose and Scope

This document outlines a comprehensive memory architecture for AI agents, consisting of:

1.1 Long-Term Archival Storage

- Immutable versioned database powered by *Cosdata* (semantic search and key-value store)
- Multi-modal index supporting dense, sparse, and full-text fragments
- Optimized for completeness and long-term preservation
- Stored on slow-retrieval systems (S3, filesystem, object stores)
- Unbounded growth with cross-referential indexing and temporal versioning

1.2 Core Memory (*Cosdata-memory*)

- High-speed in-RAM or disk-based layer
- Dynamic, human-like memory behavior under finite storage constraints
- Fast context-sensitive retrieval and prioritization
- Implements intelligent caching, abstraction, and compression strategies

The focus of this document is on *Cosdata-memory*, the runtime system responsible for real-time memory manipulation, retention policies, and high-relevance recall for autonomous agents.

2 Overview

This document outlines a comprehensive memory architecture for AI agents that addresses the fundamental challenge of "Infinite Memory, Finite Storage" through dynamic, context-aware memory management systems that mirror human cognitive processes.

3 Core Architecture Principles

3.1 Dynamic Memory Architecture

The system implements real-time updates and persistent memory across sessions, moving beyond static parameters or simple vector databases. Memory adapts continuously based on interactions, maintaining consistency while enabling progressive learning without full retraining.

3.2 Intelligent Compression Through Abstraction

Rather than storing raw data, the system captures relationships, patterns, and semantic meaning. This approach enables efficient storage while maintaining rich information density through sophisticated abstraction mechanisms.

3.3 Adaptive Forgetting & Reinforcement

The memory system dynamically strengthens frequently accessed knowledge while allowing less critical data to decay naturally. This prevents information overload while ensuring important memories remain accessible.

3.4 Memory Reconstruction

Instead of exact retrieval, the system reconstructs memories based on context and associations, providing flexibility and efficiency superior to traditional storage approaches.

3.5 Structured, Typed Memory System

Moving beyond unstructured embeddings, the framework includes type inference, temporal dynamics, and contextual markers that enable nuanced and accurate information retrieval.

4 Memory Component Definitions

4.1 Memory

What happened — a full, human-recallable episode or experience

A structured experience composed of fragments, stored to reflect events as humans would recall them. Each memory represents a complete episode with temporal, spatial, and contextual dimensions.

4.1.1 Example Structure:

```
{
  "event": "breakfast",
  "timestamp": "2025-07-10T09:30",
  "location": "home",
  "food": "cereal",
  "companion": "John",
  "mood": "rushed",
  "weather": "sunny",
  "pre_event": "preparing for meeting"
}
```

4.2 Context

How the memory is encoded and retrieved — the surrounding state or cues

A dynamically constructed frame made of select fragments that influences memory storage and recall. Context provides the lens through which memories are interpreted and retrieved.

4.2.1 Why Encoding Context Matters

Encoding context is not about what gets stored — it's about what gets emphasized. It enables:

- More meaningful, efficient, and explainable retrieval
- Learning patterns from what truly mattered
- Structuring long-term behavior around what was salient
- Preventing retrieval drowning in irrelevant matches

Without encoding context, everything is treated as equal importance, losing the **why** behind the memory system.

4.2.2 Multi-Context Encoding Example

The same breakfast event can be encoded with different contextual emphasis:

1. Encoding Context A (Task-Focused):

```
{
 "memory_id": "breakfast_20250710",
 "context_id": "ctx_task_20250710",
 "context_type": "task"
 "weighted_fragments": {
   "mood": {"value": "rushed", "weight": 0.9},
   "goal": {"value": "prepare_for_meeting", "weight": 0.8},
   "time": {"value": "09:30", "weight": 0.7}
 },
}
```

2. Encoding Context B (Social-Focused):

```
{
 "memory_id": "breakfast_20250710",
 "context_id": "ctx_social_20250710",
 "context_type": "social"
 "weighted_fragments": {
   "companion": {"value": "John", "weight": 0.9},
   "location": {"value": "home", "weight": 0.8},
   "interaction_quality": {"value": "comfortable", "weight": 0.6}
 },
}
```

4.2.3 Retrieval Context Matching Strategy

A retrieval context must match at least 2+ key-value pairs from a stored encoding context to trigger meaningful recall. This strategy provides:

- Increased retrieval specificity
- Reduced noise from coincidental matches
- Simulation of deeper, more intentional recall

1. Example Retrieval Scenarios:

   (a) Query: "What happened when I was rushed before a meeting?" Retrieval Context:

   ```
   {
    "mood": "rushed",
    "goal": "prepare_for_meeting",
    "context_type": "task_oriented"
   }
   ```

   Matches: Context A (3/3 matches) → High relevance retrieval
   Result: Strong recall of task-focused breakfast memory

   (b) Query: "What did I do with John at home?" Retrieval Context:

   ```
   {
    "companion": "John",
    "location": "home",
    "context_type": "social_oriented"
   }
   ```

   Matches: Context B (2/2 matches) → High relevance retrieval
   Result: Strong recall of social-focused breakfast memory

4.3 Fragments

Atomic building blocks — the key-value pairs that compose a memory

The smallest meaningful units of experience representing who, what, where, when, and how elements. Fragments are reusable across multiple memories and contexts.

4.3.1 Fragment Types:

- **Temporal:** `timestamp`, `duration`, `sequence_order`
- **Spatial:** `location`, `environment`, `proximity`
- **Social:** `companion`, `interaction_type`, `relationship`
- **Affective:** `mood`, `emotion`, `satisfaction`
- **Semantic:** `activity`, `object`, `concept`

4.4 Semantic Abstraction

Meta-schema — the structural or indexing features used across memories

Generalized templates or schemas used for organizing, querying, and summarizing memories. These provide structural patterns that enable efficient indexing and retrieval. While raw memories capture what happened, semantic abstraction provides the structure needed to derive patterns, insights, and generalizations across events.

4.4.1 Key Components:

- `event_type`: The category of memory this abstraction applies to (e.g., "breakfast").
- `core_fragments`: The fundamental key-value fields typically found in this event type.
- `semantic_views`: Computable views over the memory base, such as aggregation, co-occurrence analysis, or temporal clustering. Each view represents a meaningful perspective for indexing or summarizing knowledge.

4.4.2 Schema Example:

```
{
  "schema_id": "breakfast_schema_v1",
  "event_type": "breakfast",
  "core_fragments": ["timestamp", "location", "food", "companion", "mood", "goal"],

  "semantic_views": [
    {
```

```
  "view_id": "breakfast_companion_freq",
  "description": "Companion frequency in breakfast memories",
  "view_type": "aggregation",
  "relation": "breakfast → companion",
  "aggregation": {
    "operation": "count",
    "group_by": ["companion"]
  }
},
{
  "view_id": "breakfast_mood_goal",
  "description": "Co-occurrence frequency of mood and goal during breakfast",
  "view_type": "aggregation",
  "relation": "breakfast → mood+goal",
  "aggregation": {
    "operation": "count",
    "group_by": ["mood", "goal"]
  }
},
{
  "view_id": "breakfast_time_clusters",
  "description": "Clusters of breakfast times based on timestamp distribution",
  "view_type": "temporal_clustering",
  "relation": "breakfast → timestamp",
  "clustering": {
    "method": "kmeans",
    "field": "timestamp",
    "parameters": {
      "k": 3,
      "time_granularity": "300"  // 5 minute
    }
  }
},
{
  "view_id": "breakfast_recency_weighted",
  "description": "Recent vivid breakfast memories",
  "view_type": "recency_prioritization",
  "relation": "breakfast → timestamp",
  "prioritization": {
  "field": "timestamp",
```

```
      "max_retention": 3, //top 3 instances
      }
    },
  ]
}
```

4.5 Semantic Memory

Derived facts — generalizations or inferences extracted from episodic memories

Facts or knowledge derived from repeated experiences, statistics, or inference over episodic memories. These represent learned patterns rather than specific recallable episodes.

Semantic memory is the materialized result of applying semantic abstraction schemas across episodic memory data.

- Aggregated counts
- Time habits
- Conceptual associations
- Learned rules or statistics

4.5.1 Derived Knowledge Examples:

- "John was present in 5 of the last 7 breakfasts"
- "When rushed, user tends to eat cereal"
- "Most breakfasts happen between 9–10 AM"
- "Sunny weather correlates with positive mood during meals"

```
{
  "semantic_memory_id": "semem_user01_breakfast_v1",
  "event_type": "breakfast",
  "generated_from": "breakfast_schema_v1",
  "generated_on": "2025-07-10T20:34:00Z",

  "memory_fragments": {
    "companion_frequency": {
```

```
      "John": 9,
      "Sara": 3,
      "None": 2
    },
    "mood_goal_pairs": {
      "rushed:prepare_for_meeting": 6,
      "relaxed:catch_up_with_family": 4,
      "rushed:none": 2
    },
    "breakfast_time_clusters": [
      { "center": "08:15", "count": 11 },
      { "center": "09:30", "count": 17 },
      { "center": "11:00", "count": 4 }
    ]
  }
}
```

5 Memory Graph Architecture

5.1 Cognitive Graph Structure

Each memory functions as a node in a graph where connections represent relationships via shared fragments, semantic clusters, temporal sequences, or contextual overlaps.

5.1.1 Graph Relationship Types:

- **Temporal:** `sequential`, `concurrent`, `periodic`
- **Semantic:** `similar_activity`, `shared_concept`, `causal_relationship`
- **Social:** `same_companion`, `similar_interaction`, `relationship_evolution`
- **Spatial:** `same_location`, `nearby_location`, `location_transition`
- **Affective:** `mood_match`, `emotional_progression`, `satisfaction_correlation`

5.1.2 Example Memory Graph:

```
[Breakfast w/ John, Jul 10]
    == temporal_next → [Morning Meeting, Jul 10]
```

```
== shares_companion → [Dinner w/ John, Jul 5]
== same_location → [Lunch Alone, Jul 6]
== mood_match → [Late Night Debug, Jul 9]
== causal_relationship → [Rushed Preparation, Jul 10]
```

6 Storage Architecture

6.1 Overview

The memory system implements a proprietary, high-performance storage architecture designed for bounded memory operation with human-like memory characteristics. The system combines a custom bounded hash-partitioned index with a complementary key-value store to achieve efficient context-based retrieval while maintaining strict memory constraints.

6.2 Core Components

6.2.1 Bounded Hash-Partitioned Index

The system utilizes a proprietary high-performance bounded hash-partitioned index implementing a quotient-remainder architecture using 32-bit hashes of terms. The indexing strategy employs a fixed divisor of say 2^{16} to partition each term hash into distinct components that provide explicit bounds on the index structure.

The hash partitioning mechanism divides each `u32` term hash into a quotient and remainder using the 2^{16} divisor. The quotient, derived from the upper 16 bits, serves as the identifier for the dictionary or term entry, while the remainder, taken from the lower 16 bits, is used to locate or index into the corresponding posting list. This partitioning strategy creates a hard limit on the total number of unique terms in the bounded hash-partitioned index, explicitly capping the vocabulary at $2^{16} = 65{,}536$ distinct term hashes.

To further constrain the size of each posting list, the remainder undergoes additional reduction through a modulo $2^{10} = 1024$ operation. This effectively shrinks the addressable space within each partition, ensuring that within each of the 65,536 term entries, there can be at most 1024 distinguishable postings such as context identifiers. This dual-level bounding mechanism tightly controls both vocabulary size and maximum postings per term, creating a compact and performant indexing strategy optimized for speed, cache locality, and memory efficiency.

The architecture enforces bounded memory allocation with a small epsilon tolerance expressed as a configurable percentage, ensuring that hard

memory bounds are never exceeded under any operational conditions. This hash-partitioned approach provides predictable memory usage patterns while maintaining high-performance access characteristics through the mathematical constraints of the quotient-remainder division.

Context storage and encoding within the bounded hash-partitioned index follows a sophisticated approach where contexts are encoded as sparse vectors with uniform distribution characteristics. These sparse vectors are intentionally scattered across different postings within the hash-partitioned structure to optimize retrieval patterns and prevent clustering that could degrade performance. The hash-partitioned design ensures that context-fragment mappings are distributed efficiently across the bounded posting space.

The system implements a comprehensive priority and persistence management framework where each entry maintains a priority/persistence score that determines its eligibility for overwriting. These scores undergo periodic and probabilistic updates based on frequency of access through reinforcement learning mechanisms, temporal decay functions that simulate human forgetting, and context relevance weighting. Entries with the lowest persistence scores become candidates for space reclamation when the system approaches capacity constraints within each hash partition.

6.2.2 Complementary Key-Value Store

The bounded hash-partitioned index operates in conjunction with a proprietary high-performance key-value store that provides direct access to complete memory objects. This key-value store maintains a limited key capacity and employs probabilistic space management strategies. Complete memory objects are stored as values within this system, and like the bounded hash-partitioned index, the key-value store is subject to overwrite and space reuse policies when operating under capacity pressure.

The integration between the bounded hash-partitioned index and key-value store creates a cohesive storage ecosystem. Memory objects stored in the key-value store are referenced by context entries distributed across the hash-partitioned posting structure, establishing linking between context entries and memory storage. Both storage systems execute their own eviction/overwrite policies yet the system operates coherently and optimize memory utilization while respecting the bounded nature of the hash-partitioned index.

6.3 Retrieval Architecture

6.3.1 Context-Based Retrieval

Retrieval from the bounded hash-partitioned index operates through sophisticated context-based mechanisms that leverage the hash partitioning structure for efficient access. The system utilizes fragment weightings and context-fragment aggregations for scoring, with sparse vector operations optimized for the hash-partitioned architecture.

Top-k inner product calculations operate within the constraints of the hash-partitioned structure, utilizing the 1024-entry posting limit per term to maintain efficient processing. This approach enables effective identification of relevant memories based on contextual similarity while maintaining the bounded memory constraints.

The retrieval process begins with hash computation of query terms, followed by quotient-remainder decomposition to identify target partitions. Within each partition, the system performs efficient searches across the bounded posting lists, utilizing the persistence scores and context weightings to rank results. The mathematical bounds of the hash partitioning ensure that retrieval operations complete within predictable time and space constraints.

6.3.2 Memory Object Retrieval Sequence

Following successful context-based retrieval, the system attempts to fetch the complete memory object from the key-value store using the context-provided references. When the complete memory object is unavailable due to eviction or capacity constraints, the system initiates probabilistic reconstruction based on the checksum values embedded within the context entries.

The reconstruction process operates with a confidence scoring mechanism that evaluates the likelihood of successful memory restoration based on available fragments and their integrity checksums. This reconstruction attempt is subject to a configurable compute threshold that prevents excessive processing overhead. If the reconstruction process exceeds this threshold or the confidence score falls below acceptable levels, the reconstruction attempt is terminated and the system acknowledges that the context exists but provides only the available contextual information without further enrichment.

This tiered approach ensures responsive operation while providing maximum possible information retrieval within computational constraints. The bounded hash-partitioned index provides the contextual foundation, while

the key-value store and reconstruction mechanisms offer progressively more detailed memory access based on availability and computational feasibility.

6.4 Data Integrity and Reconstruction

6.4.1 Checksum-Based Integrity

The system incorporates `u8` checksums embedded within individual posting entries to provide integrity verification capabilities within the hash-partitioned structure. Each posting maintains its own checksum that represents the entire memory object while being encoded within that specific posting. Since contexts can be split across different terms and their corresponding postings, the checksums from multiple postings can collectively contribute to reconstruction efforts, with each checksum providing independent verification of the complete memory object.

6.4.2 Probabilistic Reconstruction

When complete memory objects are not directly available, the system employs probabilistic reconstruction using checksums distributed across multiple postings. The hash-partitioned structure supports efficient reconstruction by limiting the search space to bounded posting lists. Client-provided memory checksums can optimize reconstruction efficiency by providing additional verification points that map to specific partitions within the hash partitioning scheme.

The system maintains fault tolerance through graceful degradation with confidence-based reconstruction that leverages the distributed checksum approach, where multiple independent checksums representing the same memory object can improve reconstruction success rates and reduce processing requirements.

6.5 Graph Connectivity

6.5.1 Memory and Context Linking

Special fields within postings enable graph-based memory connections, allowing individual memories and contexts to be linked to related memories and contexts across the hash-partitioned structure. The hash partitioning provides efficient pathways for traversing related memories that may be distributed across different quotient partitions, while the bounded posting size ensures predictable traversal performance.

Graph connectivity operates through explicit link fields embedded within posting entries, creating a distributed graph structure that spans the hash-partitioned space. Links between memories and contexts are maintained through hash-based addressing, enabling efficient traversal while respecting the bounded nature of the posting lists. The quotient-remainder structure provides natural clustering for related memories, improving graph traversal locality.

Relationship management operates through efficient traversal algorithms designed for memory association discovery within the hash-partitioned architecture. The graph connectivity enables complex retrieval patterns where memories can be accessed not only through direct context matching within partitions but also through associative pathways that connect related experiences and knowledge across the bounded vocabulary space.

The distributed graph structure supports various relationship types including temporal sequences, semantic associations, and contextual similarities. Graph maintenance operations respect the bounded nature of the hash-partitioned index, with link management algorithms designed to operate efficiently within the posting list constraints.

6.6 Passive Maintenance Model

Unlike traditional memory systems that rely on background threads or scheduled jobs to maintain priority or persistence scores, the memory system is designed to operate without any such continuous processes. All updates to persistence or reinforcement scores are performed **lazily** and **opportunistically**—only during critical points such as when new memory is being indexed or when eviction decisions must be made. This approach ensures that memory upkeep does not consume additional runtime resources, keeping the system lightweight and efficient under real-time constraints.

By embedding score updates directly into the natural lifecycle of memory operations, the system avoids unnecessary churn, CPU load, or synchronization overhead. This design choice preserves the integrity of memory prioritization while enabling the architecture to scale predictably, even under constrained hardware environments or edge deployment scenarios.

7 Implementation Guidelines

7.1 Memory Encoding Process

1. **Fragment Extraction:** Identify atomic components from experiences

2. **Context Construction:** Build encoding framework based on current state
3. **Schema Application:** Apply relevant semantic abstractions
4. **Graph Integration:** Establish connections with existing memories
5. **Storage Allocation:** Determine core vs. archival storage placement

7.2 Memory Retrieval Process

1. **Context Analysis:** Interpret current retrieval context
2. **Graph Traversal:** Navigate memory relationships
3. **Relevance Scoring:** Evaluate memory importance and similarity
4. **Reconstruction:** Rebuild memory based on fragments and context
5. **Reinforcement:** Update access patterns and importance scores

8 Integration Considerations

8.1 API Design

The memory system should provide clean interfaces for memory creation, retrieval, update, and analysis operations while maintaining abstraction over underlying storage complexity.

8.2 Scalability

Architecture must accommodate growth in memory volume, relationship complexity, and retrieval sophistication while maintaining response time requirements.

8.3 Consistency

Ensure memory coherence across concurrent operations and maintain referential integrity within the memory graph structure.

9 Conclusion

This novel memory architecture represents a fundamental shift toward AI systems that better mirror human cognitive processes, where memory is dynamic, reconstructive, and context-aware rather than static and retrieval-based. The *Cosdata dynamic memory system* serves as the primary interface for real-time memory operations, while the archival storage provides unlimited capacity for long-term memory preservation and semantic search capabilities.