Memory Infrastructure Layer: Entity-Specific Associative Networks for Context-Aware AI Systems

Fredric Cliver
OpenAIKits
fredriccliver@gmail.com

January 17, 2026

Abstract

Modern AI applications face significant challenges in managing context and memory across interactions. While current AI stacks include application layers, inference infrastructure, and model providers, they lack a dedicated **Autonomous Memory Infrastructure**—a general, application-independent memory layer that can seamlessly integrate with workflows, agents, and diverse AI systems. This forces each application to implement its own context management solutions, resulting in duplicate implementations, inefficiency, and lack of standardization. We present OAK.memory, a unified autonomous memory infrastructure developed as part of the OpenAIKits project that addresses this gap through **entity-specific associative networks**. Our approach combines vector similarity search on memory nodes with graph traversal algorithms, providing a hybrid retrieval mechanism that leverages both semantic similarity and structural relationships. A key design decision is that graph edges store only structural connections (UUID arrays) without static semantic embeddings, enabling dynamic relationship inference from node content at query time. This design, inspired by human brain structure where memory associations differ for each individual, allows each entity to maintain its own unique memory network structure, where the same concept can connect differently for different entities. Unlike existing systems that use static edge embeddings or globally-defined relationships, our approach provides greater flexibility and context-awareness. This design enables personalized, context-aware AI systems without requiring model fine-tuning, providing a general memory solution that can be easily connected to workflows, agents, and various AI applications. We describe the architecture, key design decisions, and demonstrate the feasibility of this approach through system design and validation.

1 Introduction

1.1 Background

The rapid advancement of large language models (LLMs) has enabled sophisticated AI applications across diverse domains. However, the current AI technology stack, while comprehensive in many aspects, exhibits a notable gap. The stack typically consists of three primary layers:

- **Application Layer**: Domain-specific, user-facing applications
- **Inference/Training Infrastructure**: vLLM, TensorRT, Modal, Replicate, and similar platforms
- **Model Providers**: OpenAI, Anthropic, Meta, Mistral, and other LLM providers

A critical **missing layer** is **Autonomous Memory Infrastructure**—a general, application-independent memory system that can autonomously manage context and memory across diverse domains and use cases.

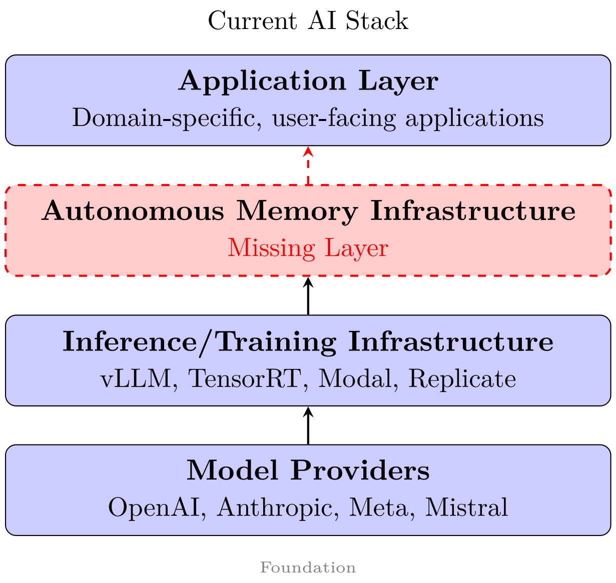

Current AI Stack
Application Layer
Domain-specific, user-facing applications
Autonomous Memory Infrastructure
Missing Layer
Inference/Training Infrastructure
vLLM, TensorRT, Modal, Replicate
Model Providers
OpenAI, Anthropic, Meta, Mistral
Foundation

Figure 1: Current AI technology stack showing the missing Autonomous Memory Infrastructure layer.

Unlike application-specific solutions, such infrastructure should be domain-agnostic and seamlessly integrate with workflows, agents, and various AI systems without requiring custom implementations for each application. In the absence of such standardized memory infrastructure, each application must independently implement its own solutions for:

- Retrieval-Augmented Generation (RAG) implementations, including vector database integration and search logic
- User-specific memory management and personalization
- Conversation history management and session persistence
- Context window optimization and token management

This fragmentation leads to duplicate implementations, operational inefficiency, and a lack of standardization across the AI ecosystem, hindering interoperability and increasing development costs. What is needed is a **general memory infrastructure** that provides autonomous memory management capabilities, independent of specific applications or domains, and can be easily connected to workflows, agents, and diverse AI systems.

1.2 Problem Statement

Existing approaches to personalization and context management exhibit several fundamental limitations:

- **Fine-tuning**: Requires substantial computational resources, reduces model independence, and cannot practically maintain per-user models at scale
- **LoRA Adapters**: While more efficient than full fine-tuning, still incur training costs and introduce deployment complexity
- **Test-time Learning**: Significantly increases inference latency and operational costs
- **Prompt Engineering**: Relies on manual curation, lacks scalability, and is constrained by fixed context window limitations

Furthermore, existing memory systems for AI agents face architectural and design limitations:

- **Application-level positioning**: Prominent solutions such as Mem0 [4], MemGPT [3], and Zep are designed as applications rather than reusable infrastructure components
- **Static relationship definitions**: Edge semantics are globally defined across all entities, preventing entity-specific associations
- **Architectural complexity**: Systems like OpenMemory [1] employ multi-sector classification schemes that increase system complexity and maintenance burden
- **Insufficient conflict management**: Lack automatic mechanisms for detecting and resolving conflicting or contradictory memories

1.3 Contributions

This work presents OAK.memory, an autonomous memory infrastructure layer developed as part of the OpenAIKits project[1] that addresses the aforementioned limitations. OAK.memory provides a general, application-independent memory system that can autonomously manage context and memory, seamlessly integrating with workflows, agents, and diverse AI applications. The system is implemented as the `@openaikits/memory` package, providing reusable memory infrastructure that is domain-agnostic and can be easily connected to various AI systems. Our primary contributions are:

1. **Entity-Specific Associative Networks**: We introduce a novel memory architecture where each entity maintains its own unique memory network structure. This design, inspired by human brain structure where memory associations differ for each individual, enables personalized memory associations without requiring model modifications.
2. **Autonomous Infrastructure Layer**: Unlike existing application-level solutions, we position OAK.memory as autonomous, general-purpose memory infrastructure. The system is application and domain-independent, enabling seamless integration with workflows, agents, and diverse AI frameworks through a unified connector pattern, promoting standardization and reducing duplication across the AI ecosystem.
3. **Dynamic Relationship Inference**: We propose a design that avoids static edge embeddings. Instead, relationships are inferred dynamically from node content at query time, providing greater flexibility and context-awareness compared to statically-defined edge semantics.
4. **Unified Graph-Vector Architecture**: We present a hybrid approach that combines vector similarity search on memory nodes with graph traversal algorithms, enabling comprehensive context retrieval that leverages both semantic similarity and structural relationships.
5. **Connector Pattern**: We design a unified interface that enables seamless integration with workflows, agents, and various AI frameworks (including LangChain and custom implementations), through an adapter pattern that abstracts framework-specific details, making the memory infrastructure easily connectable to diverse AI systems.

[1]OpenAIKits: `https://openaikits.com`

2 Related Work

2.1 Memory Systems for AI Agents

2.1.1 OpenMemory (Mnemosyne)

OpenMemory [1] provides a multi-sector memory system that categorizes memories into five distinct sectors: Reflective, Semantic, Procedural, Episodic, and Emotional. Each sector exhibits different temporal decay rates and reinforcement mechanisms, modeling aspects of human memory. The system employs a waypoint graph structure for multi-hop context traversal and implements composite retrieval that combines semantic similarity search with graph-based traversal. However, the system has notable limitations: edges function only as structural connections (waypoints) and are not searchable through semantic embeddings, the multi-sector classification introduces architectural complexity, and the system is positioned as an application-level solution rather than reusable infrastructure.

2.1.2 Mem0 and MemGPT

MemGPT [3] introduces a hierarchical memory architecture inspired by operating system memory management, consisting of core memory (maintained in-context), recall memory (searchable conversation history), and archival memory (long-term semantic store). The system implements self-editing capabilities where the LLM directly manages memory through specialized tools, enabling dynamic memory updates during conversation. Mem0 [4] (2025) presents a scalable architecture that dynamically extracts and consolidates salient events from conversation streams, with an optional graph-based variant (Mem0g) that uses Neo4j for structured storage. While both systems demonstrate effective memory management, they are positioned as application-level solutions rather than infrastructure, and edge vector search capabilities remain limited even in the graph variant.

2.2 Vector-Graph Unified Approaches

Recent research has explored unified vector-graph architectures. RGL [5] presents a graph-centric modular framework for RAG on graphs, achieving significant speedup. HM-RAG [6] uses hierarchical multi-agent multimodal RAG with plug-and-play vector, graph, and web retrieval sources. However, these systems still treat edges primarily as structural connections rather than enabling entity-specific associative networks.

2.3 Agentic Memory Construction

A-MEM [7] (2025) uses LLM prompts to decide edge linking, building a graph of memories dynamically. Mem-α [8] (2025) treats memory insertion/update/deletion as actions in a Markov Decision Process, using reinforcement learning. While these approaches share similarities with our agentic construction vision, they do not emphasize entity-specific network structures.

2.4 Context Engineering

Recent research has highlighted the importance of context engineering as a separate concern. MemTrust [9] (2026) proposes a five-layer abstraction (Storage, Extraction, Learning, Retrieval, Governance) with zero-trust architecture. The Model Context Protocol (MCP) by Anthropic represents an industry effort to standardize context management interfaces. Our work aligns with this trend by positioning memory management as autonomous, general-purpose infrastructure that can operate independently of specific applications or domains.

3 Architecture

3.1 Design Philosophy: Entity-Specific Associative Networks

Our architectural design is grounded in a key insight from cognitive science: **memory associations differ for each individual entity**. In human cognition, the same concept or stimulus evokes different associations for different individuals based on their unique experiences, preferences, emotional contexts, and learned patterns. This phenomenon, well-documented in cognitive psychology [2], suggests that effective memory systems for AI should similarly allow each entity to maintain its own unique associative network structure rather than imposing a global, one-size-fits-all relationship model.

3.1.1 Motivation

Consider the concept "pizza":

- For entity "Fredric": Connected as "likes" (personal preference)
- For entity "Alice": Connected as "allergy" (personal constraint)
- For entity "Restaurant": Connected as "menu item" (business context)

The same node, but **different associations** for each entity—reflecting how human memory works. This enables natural expression of individuality and personalized context-aware memory.

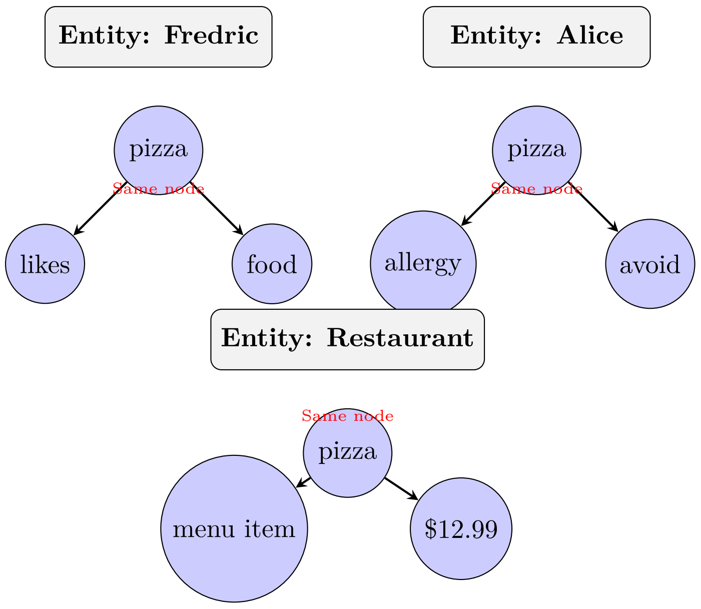


Figure 2: Entity-specific associative networks: the same concept ("pizza") connects differently for different entities, reflecting personalized memory associations.

3.1.2 Design Decision: No Static Edge Embeddings

During the design phase, we carefully evaluated the trade-offs between static edge embeddings and dynamic relationship inference. After extensive consideration, we decided **not** to store semantic meaning in edges statically. This decision is motivated by three key observations:

1. **Semantic richness limitation**: Edges, when considered in isolation, lack sufficient semantic richness to serve as meaningful search targets. Edge embeddings would capture only limited information compared to the rich semantic content available in connected nodes.

2. **Conceptual boundary preservation**: Adding substantial semantic meaning to edges would blur the distinction between structural connections and semantic entities, potentially making edges indistinguishable from nodes and undermining the graph's structural clarity.

3. **Contextual flexibility**: Dynamic relationship inference from node content provides superior flexibility and context-awareness. This approach allows relationships to be interpreted differently based on query context, entity-specific associations, and temporal factors, rather than being constrained by static embeddings determined at storage time.

Instead, we use:

- Simple structural connections (`outgoingEdges` array of UUIDs)
- Dynamic relationship inference from node content at query time
- Entity-specific network structures that naturally express individuality

3.2 System Overview

OAK.memory provides a unified memory infrastructure layer with the following core components:

- **Memory Storage**: PostgreSQL database with pgvector extension, providing efficient vector similarity search and graph traversal capabilities through recursive CTEs
- **Embedding Service**: Generates embeddings for memory nodes using configurable embedding models (OpenAI, Cohere, HuggingFace, etc.), abstracted through an adapter pattern for easy provider switching
- **Memory Connector**: Provides a unified interface for memory operations, supporting integration with LangChain and other AI frameworks through automatic adapter detection
- **Graph Structure**: Node-edge-node representation where nodes contain semantic content with embeddings, and edges are simple structural connections stored as UUID arrays

The system is distributed as the `@openaikits/memory` package, enabling easy integration into existing AI applications while maintaining framework-agnostic design principles.

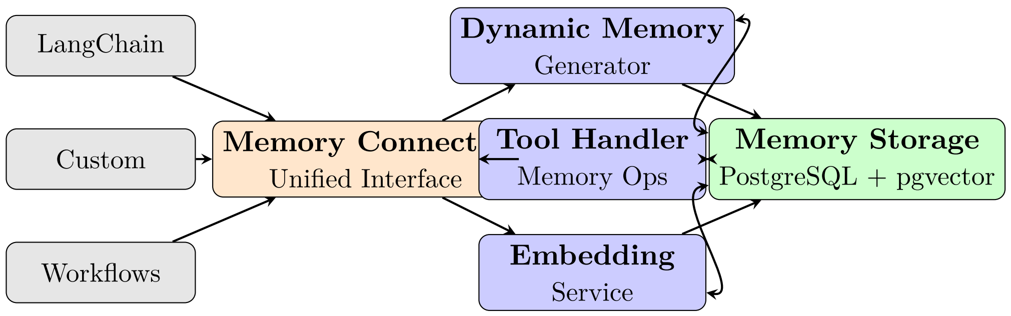


Figure 3: OAK.memory system architecture showing the connector pattern enabling framework-agnostic integration with diverse AI systems.

3.3 Memory Structure

Memories are represented as a graph where:

- **Nodes**: Core entities (users, documents, concepts, events) with vector embeddings for semantic search
- **Edges**: Structural connections stored as `outgoingEdges` arrays (UUIDs), without static semantic embeddings
- **Entity-specific networks**: Each entity maintains its own graph structure, enabling personalized associations

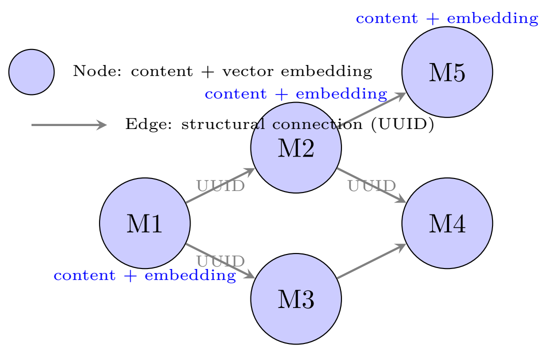


Figure 4: Memory graph structure: nodes contain semantic content with vector embeddings, while edges are simple structural connections (UUIDs) without static semantic meaning.

3.4 Key Components

3.4.1 Memory Storage

PostgreSQL-based storage with pgvector extension. Supports:

- Vector similarity search on nodes using cosine similarity
- Graph traversal using recursive CTEs following `outgoingEdges`
- Transactional operations for consistency
- Entity-based filtering for entity-specific networks

3.4.2 Embedding Service

Generates embeddings for nodes using configurable embedding models. The service is abstracted through an adapter pattern, allowing easy switching between providers (OpenAI, Cohere, HuggingFace, etc.).

3.4.3 Connector Pattern

The connector pattern provides a unified interface that enables seamless integration with diverse AI systems:

- **Workflow Integration**: Can be easily connected to workflow systems and orchestration frameworks
- **Agent Integration**: Supports integration with AI agents through standardized interfaces

- **Framework Support**: LangChain chains (automatic detection and adapter creation), custom chatting managers (via ChattingManager interface), and direct API calls

For example, integrating with a LangChain chain is straightforward:

```
import { MemoryConnector } from '@openaikits/memory';

const connector = new MemoryConnector({
  entityId: 'user_123',
  storageAdapter: postgresAdapter,
  embeddingAdapter: openAIAdapter
});

// Automatic adapter detection for LangChain
const chain = new ConversationChain({
  memory: connector.getLangChainMemory()
});
```

Listing 1: LangChain integration example (TypeScript/JavaScript)

This design makes OAK.memory application and domain-independent, allowing it to serve as a general memory infrastructure that can be connected to any AI system requiring memory capabilities.

3.4.4 Dynamic Memory Generator

The Dynamic Memory Generator orchestrates context retrieval by combining multiple search strategies:

- **Vector Search**: Performs semantic similarity search to identify memory nodes relevant to the query based on embedding similarity
- **Graph Traversal**: Follows structural connections (`outgoingEdges`) from initial search results to discover contextually related memories through graph relationships
- **Hybrid Aggregation**: Combines results from both approaches, deduplicates memories, and ranks them to provide comprehensive context for memory generation

The retrieval process can be invoked programmatically:

```
const memories = await memoryGenerator.retrieve({
  entityId: 'user_123',
  query: 'What are my preferences?',
  maxResults: 10,
  graphDepth: 2
});
// Returns: vector search results +
//          graph-connected memories
```

Listing 2: Hybrid memory retrieval (TypeScript/JavaScript)

This hybrid approach ensures that both semantically similar memories and structurally connected memories are considered, providing richer context than either approach alone.

3.4.5 Tool Handler

Processes AI tool calling requests for memory operations through a unified interface:

- `createMemory`: Creates new memories with automatic embedding generation and entity validation

- `updateMemory`: Updates existing memories with embedding regeneration and consistency checks
- `updateMemoryLink`: Manages connections between memories, enabling dynamic graph construction
- `deleteMemory`: Removes memories and automatically cleans up associated connections

The tool handler architecture enables extension to advanced memory management operations, which are also implemented as tools:

- `resolveMemoryConflict`: Automatically detects and resolves conflicting or contradictory memories through semantic analysis and temporal reasoning
- `compressMemory`: Performs memory compression by consolidating redundant or similar memories, optimizing storage while preserving essential information
- `debugReasoning`: Provides reasoning debugging capabilities by tracing memory retrieval paths and analyzing context construction for transparency and troubleshooting

The tool handler supports OpenAI function calling format. For example, creating a memory is done through a tool call:

```
{
  "name": "createMemory",
  "arguments": {
    "entity_id": "user_123",
    "content": "User prefers dark mode UI"
  }
}
```

Listing 3: Memory creation tool call

The system automatically generates embeddings, validates the entity, and stores the memory in the graph structure. All memory operations, including conflict resolution, compression, and reasoning debugging, are implemented through the unified tool interface, allowing AI systems to autonomously manage memory lifecycle. The tool handler can be extended to support other tool-calling protocols, enabling seamless integration with various AI frameworks.

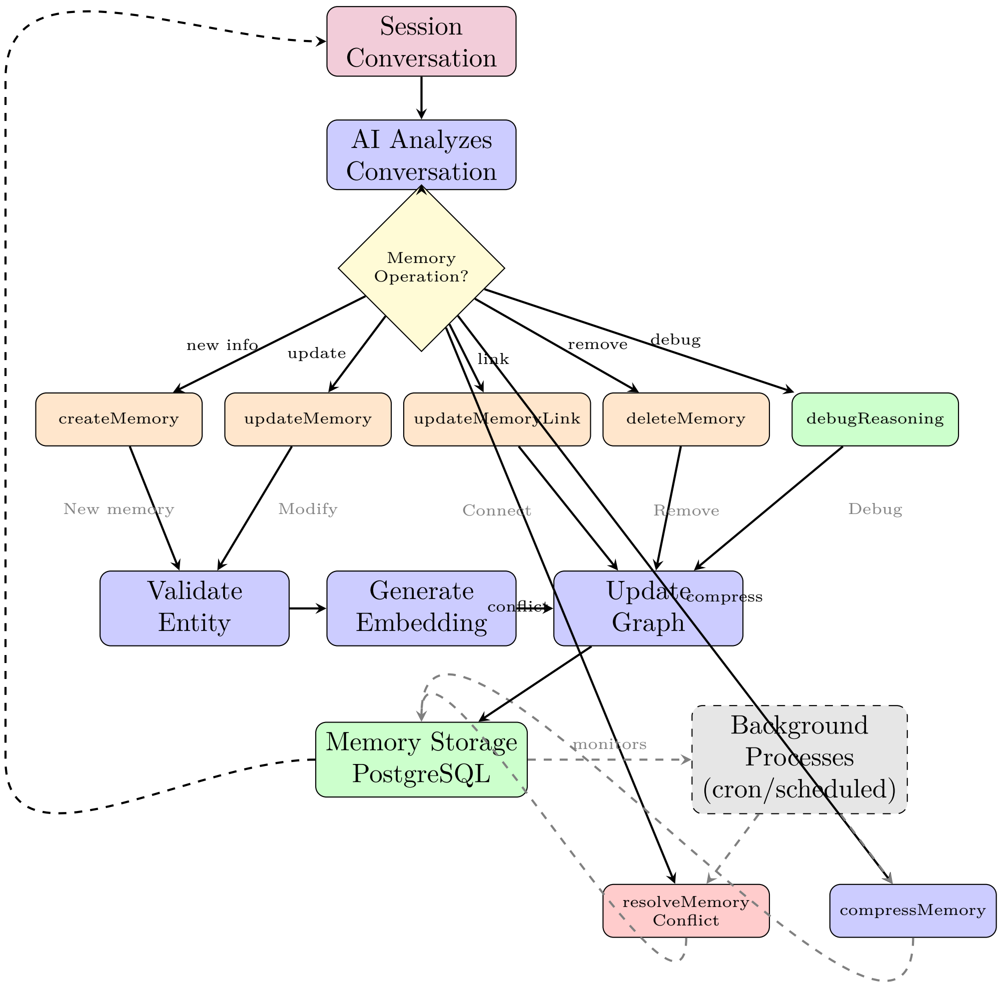


Figure 5: Memory management workflow: AI analyzes session conversations and performs interactive memory operations (create, update, link, delete, reasoning debugging) through the tool handler. Background processes (cron/scheduled) autonomously perform conflict resolution and memory compression.

4 Implementation

4.1 Database Schema

The system uses PostgreSQL with the following schema:

```sql
CREATE TABLE memories (
  id UUID PRIMARY KEY,
  entity_id TEXT NOT NULL,
  content TEXT NOT NULL,
  embedding VECTOR(1536),
  outgoing_edges UUID[] DEFAULT '{}',
  created_at TIMESTAMPTZ,
  updated_at TIMESTAMPTZ
);
```

Listing 4: Memory storage schema

Key design decisions:

- `entity_id` is TEXT (not UUID) to support various entity types (user, persona, workspace, agent)
- `outgoing_edges` is a UUID array for simple structural connections
- Vector index using IVFFlat for efficient similarity search
- GIN index on `outgoing_edges` for fast graph traversal

4.2 Retrieval Algorithms

4.2.1 Node Vector Search

We perform semantic similarity search on memory nodes using cosine similarity. Given a query embedding $\mathbf{q} \in \mathbb{R}^d$ and a memory node embedding $\mathbf{m} \in \mathbb{R}^d$, the similarity score is computed as:

$$\text{similarity}(\mathbf{q}, \mathbf{m}) = 1 - \frac{\mathbf{q} \cdot \mathbf{m}}{||\mathbf{q}|| \cdot ||\mathbf{m}||} = 1 - \cos(\theta) \tag{1}$$

where θ is the angle between vectors $\mathbf{q}$ and $\mathbf{m}$. This metric ranges from 0 (orthogonal, no similarity) to 1 (identical vectors). We filter results by a configurable similarity threshold τ (typically $\tau = 0.7$) and rank results by similarity score in descending order.

4.2.2 Graph Traversal

We implement breadth-first traversal from starting nodes following the `outgoingEdges` array. The algorithm uses PostgreSQL recursive Common Table Expressions (CTEs) for efficient traversal:

```sql
WITH RECURSIVE memory_tree AS (
  -- Base case: starting memories
  SELECT id, outgoing_edges, 0 AS depth, ARRAY[id] AS visited_path
  FROM memories WHERE id = ANY(start_ids)

  UNION ALL

  -- Recursive case: follow outgoing_edges
  SELECT m.id, m.outgoing_edges, mt.depth + 1,
         mt.visited_path || m.id
  FROM memories m
  INNER JOIN memory_tree mt ON m.id = ANY(mt.outgoing_edges)
  WHERE mt.depth < max_depth
    AND NOT (m.id = ANY(mt.visited_path))
)
SELECT * FROM memory_tree WHERE depth > 0;
```

Listing 5: Graph traversal using recursive CTE

Key features:

- Efficient database-level traversal using recursive CTEs
- Configurable depth limits to control exploration scope
- Cycle detection through visited path tracking

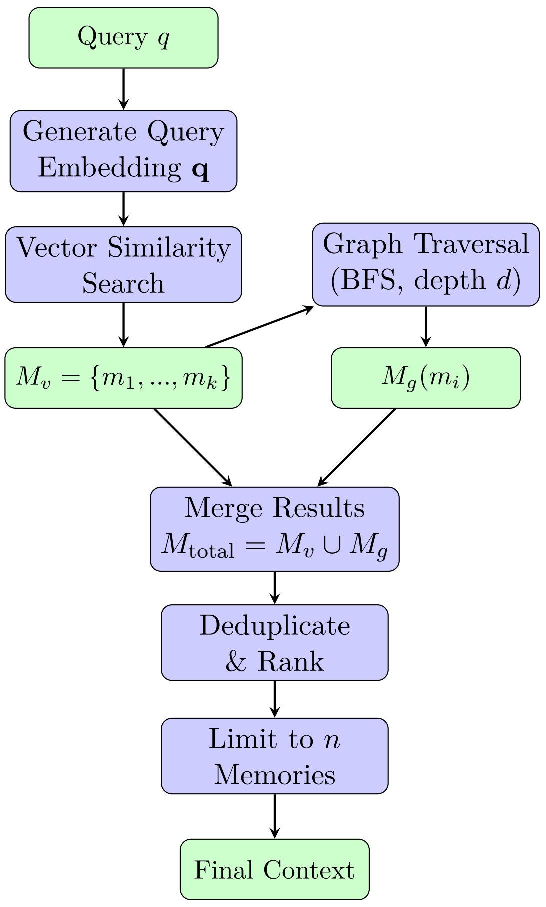
- Support for multi-source traversal (starting from multiple nodes simultaneously)
- Entity-scoped traversal ensuring network isolation

4.2.3 Hybrid Search

Our hybrid search algorithm combines vector similarity search with graph traversal to provide comprehensive context retrieval. The algorithm proceeds as follows:

1. **Initial vector search**: Given a query q, we generate a query embedding $\mathbf{q}$ and perform vector similarity search to identify the top-k most relevant memory nodes $M_v = \{m_1, m_2, \ldots, m_k\}$ where similarity$(\mathbf{q}, \mathbf{m}_i) \geq \tau$.
2. **Graph expansion**: For each memory $m_i \in M_v$, we perform graph traversal following `outgoingEdges` up to depth d, collecting connected memories $M_g(m_i)$.
3. **Result aggregation**: We merge all discovered memories: $M_{\text{total}} = M_v \cup \bigcup_{i=1}^{k} M_g(m_i)$.
4. **Deduplication and ranking**: We remove duplicate memories (by UUID) and rank the remaining results, prioritizing vector search results over graph traversal results.
5. **Result limiting**: We limit the final result set to a maximum of n memories, ensuring manageable context size.

This hybrid approach leverages both semantic similarity (for initial discovery) and structural relationships (for context expansion), providing more comprehensive context than either approach alone.



Query q
Generate Query Embedding $\mathbf{q}$
Vector Similarity Search
Graph Traversal (BFS, depth d)
$M_v = \{m_1, ..., m_k\}$
$M_g(m_i)$
Merge Results $M_{\text{total}} = M_v \cup M_g$
Deduplicate & Rank
Limit to n Memories
Final Context

Figure 6: Hybrid search algorithm flow: combining vector similarity search with graph traversal for comprehensive context retrieval.

4.3 Entity-Specific Network Isolation

Each entity maintains its own memory network through:

- `entity_id` filtering in all queries
- Entity-scoped graph traversal (only following edges within the same entity)
- Automatic entity validation in tool operations

This ensures that memory associations remain entity-specific, enabling personalized networks.

5 Preliminary Results

5.1 System Capabilities

OAK.memory provides a comprehensive memory infrastructure with the following capabilities:

- **Memory Storage**: PostgreSQL database with pgvector extension, supporting efficient vector similarity search and graph traversal operations
- **Node Vector Search**: Semantic similarity search using cosine similarity with configurable similarity thresholds
- **Graph Traversal**: Breadth-first traversal implemented using PostgreSQL recursive CTEs, supporting configurable depth limits and cycle detection
- **Memory Connector**: Unified interface supporting LangChain integration through automatic adapter detection and creation, with extensibility for other AI frameworks
- **Dynamic Memory Generator**: Component that collects augmentation data by combining vector search and graph traversal results for comprehensive context retrieval
- **Tool Handler**: Comprehensive implementation of memory operations (create, update, delete, link) with automatic validation, embedding generation, and connection management
- **Entity-Specific Network Isolation**: Each entity maintains its own independent memory network, enabling personalized associations without cross-entity interference

5.2 Architecture Validation

The architecture has been validated through practical implementation and testing:

- **Framework Integration**: Successful integration with LangChain chains, demonstrating the connector pattern's effectiveness for framework-agnostic design
- **Production-like Scenarios**: Tool-based memory management tested in scenarios simulating real-world usage patterns
- **Entity Isolation**: Entity-specific network isolation confirmed through testing, ensuring that memory associations remain scoped to individual entities
- **Hybrid Search**: Hybrid search combining vector and graph approaches validated to work as designed, providing comprehensive context retrieval

5.3 Qualitative Observations

Observations from implementation and testing suggest:

- Entity-specific networks naturally express personalized associations without explicit relationship type definitions
- Dynamic relationship inference provides flexibility in interpreting connections based on query context
- The simplified architecture (compared to multi-sector approaches) reduces complexity while maintaining expressiveness
- Infrastructure-layer positioning enables reuse across different projects and frameworks

5.4 Extended Capabilities

The architecture supports advanced memory management capabilities:

- **Conflict Detection and Resolution**: Automatic detection and resolution of conflicting or contradictory memories through semantic analysis and temporal reasoning, implemented as `resolveMemoryConflict` tool
- **Memory Compression**: Consolidation of redundant or similar memories to optimize storage while preserving essential information, implemented as `compressMemory` tool
- **Reasoning Debugging**: Tracing memory retrieval paths and analyzing context construction for transparency and troubleshooting, implemented as `debugReasoning` tool
- **Agentic Graph Construction**: AI-driven automatic edge construction and memory organization, enabling the system to autonomously build and refine memory networks
- **Temporal Reasoning**: Integration of temporal decay and reinforcement mechanisms to model memory strength over time
- **Multi-modal Support**: Support for images, audio, and other modalities beyond text
- **Comprehensive Evaluation**: Benchmark evaluation on established datasets (LoCoMo, LongMemEval) and quantitative comparison with existing systems

6 Discussion

6.1 Key Design Decisions

Our architectural decision to avoid static edge embeddings and instead employ entity-specific networks with dynamic relationship inference represents a significant departure from existing approaches in the literature. This design choice offers several advantages:

- **Personalization**: Each entity's unique associations are naturally expressed through its distinct network structure, enabling personalized memory without requiring model fine-tuning or per-entity model instances.
- **Complexity reduction**: By avoiding static edge embeddings, we eliminate the need to manage edge embedding generation, storage, and retrieval models, reducing system complexity and operational overhead.

- **Contextual flexibility**: Relationships can be interpreted differently based on query context, entity-specific associations, and temporal factors, providing greater adaptability than statically-defined edge semantics.
- **Cognitive plausibility**: The design reflects how memory associations differ for each individual in human cognition, providing a more biologically-inspired approach to AI memory systems.

6.2 Comparison with Existing Systems

Our approach differs from existing systems in several key aspects:

- **vs. OpenMemory**: Unlike OpenMemory's multi-sector classification scheme, our simplified architecture reduces complexity while maintaining expressiveness through entity-specific networks. We avoid the overhead of sector classification while still enabling personalized memory associations.
- **vs. Mem0/MemGPT**: Unlike these application-level solutions, our infrastructure-layer positioning enables reuse across multiple projects and frameworks. Our connector pattern provides framework-agnostic integration, whereas Mem0/MemGPT are designed as complete applications.
- **vs. Systems with Static Edge Semantics**: Unlike systems that define edge semantics globally (e.g., labeled graph databases), our entity-specific networks enable personalized associations where the same concept can connect differently for different entities, more closely modeling human memory structure.
- **vs. Vector-Only Systems**: Unlike pure vector-based systems (e.g., Pinecone, Chroma), we combine vector search with graph traversal, enabling both semantic similarity discovery and structural relationship exploration.

6.3 Implications

This work demonstrates that:

- Memory infrastructure can be positioned as a reusable layer
- Entity-specific networks enable personalization without model fine-tuning
- Dynamic relationship inference provides flexibility over static embeddings
- Simplified architectures can be effective when designed thoughtfully

7 Conclusion

We have presented OAK.memory, an autonomous memory infrastructure layer developed as part of the OpenAIKits project that introduces entity-specific associative networks inspired by human brain structure. Our key contribution is demonstrating that allowing each entity to maintain its own unique memory network structure, combined with dynamic relationship inference from node content, enables personalized, context-aware AI systems without requiring model fine-tuning. By positioning memory management as autonomous, general-purpose infrastructure that is application and domain-independent, we provide a solution that can seamlessly integrate with workflows, agents, and diverse AI systems, addressing a critical gap in current AI technology stacks.

The architecture provides a foundation for advanced memory capabilities, including agentic graph construction, automatic conflict detection and resolution, temporal reasoning, and multi-modal support. As an autonomous, general-purpose memory infrastructure, OAK.memory can operate independently of specific applications or domains, enabling seamless integration with workflows, agents, and diverse AI systems. We believe this approach represents a significant step toward making memory management a first-class, autonomous infrastructure component in AI systems, similar to how inference infrastructure and model providers have become standardized layers.

The system supports comprehensive evaluation on established benchmarks (LoCoMo [13], LongMemEval [10]), quantitative comparison with existing systems, and extension to temporal reasoning and multi-modal memory. The implementation is available as part of the OpenAIKits project at `https://openaikits.com`.

Acknowledgments

We thank the open-source community for tools and frameworks that made this work possible, including PostgreSQL, pgvector, and the LangChain framework. This work is part of the OpenAIKits project (`https://openaikits.com`), which aims to provide autonomous, general-purpose AI infrastructure components that can seamlessly integrate with workflows, agents, and diverse AI systems.

Project Information:
OAK.memory is available as part of the OpenAIKits project. For more information, visit `https://openaikits.com` or contact `fredriccliver@gmail.com`.

References

[1] OpenMemory Architecture Documentation. `https://openmemory.cavira.app`

[2] Tulving, E. (1972). Episodic and semantic memory. In E. Tulving & W. Donaldson (Eds.), *Organization of memory* (pp. 381-403). Academic Press.

[3] MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560

[4] Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv:2504.19413

[5] RGL: Graph-Centric Modular Framework for RAG on Graphs. arXiv:2503.19314

[6] HM-RAG: Hierarchical Multi-Agent Multimodal RAG. arXiv:2504.12330

[7] A-MEM: Agentic Memory for LLM Agents. arXiv:2502.12110

[8] Mem-α: Learning Memory Construction via Reinforcement Learning. arXiv:2508.19828

[9] MemTrust: Zero-Trust Memory Framework. arXiv:2601.07004

[10] LongMemEval: Benchmarking Long-term Interactive Memory in Chat Assistants. arXiv:2410.10813

[11] EvolMem: Benchmark for Multi-session Dialogue Memory. arXiv:2601.03543

[12] HaluMem: Evaluating Hallucinations in Memory Systems. arXiv:2511.03506

[13] LoCoMo: Very Long-term Conversational Memory Evaluation. arXiv:2402.17753