Documentation: Advanced Memory Agent Implementation

Markdown Knowledge Agent Team

December 19, 2025

Contents

1 Introduction

This document details the implementation of the **Memory Agent** within the Markdown Knowledge Agent system. The memory module is designed to be an autonomous, active component that intelligently manages conversation history, extracts knowledge, and optimizes context injection for the Large Language Model (LLM).

Unlike simple sliding-window memory systems, this implementation distinguishes between *Passive Memories* (storage mechanisms) and the *Active Agent* (logic for retrieval and management).

2 Architecture Overview

The architecture follows a modular design where the `MemoryModule` acts as the central coordinator.

- **Passive Memories**: Distinct storage backends optimized for different types of data (Semantic, Conceptual, Structural, User Context).
- **Active Agent**: A decision-making layer that determines *what* to recall, *where* to look, and *how* to format the information based on the user's query.

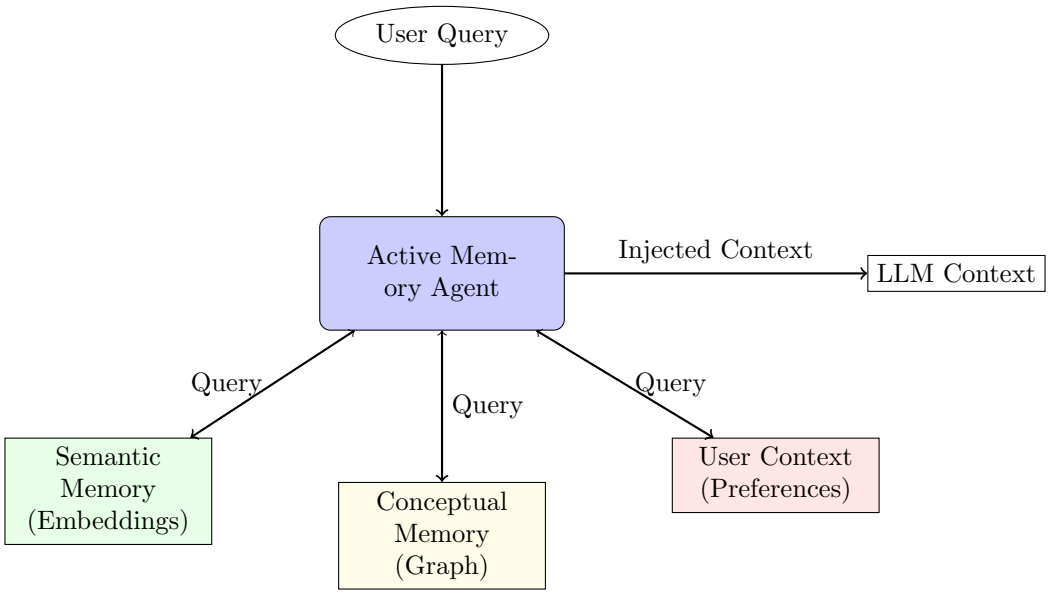

Figure 1: Memory Agent Architecture

3 Memory Components (Passive Stores)

3.1 1. Semantic Memory (Vector Store)

- **Purpose**: Stores the "meaning" of conversations to allow fuzzy matching and thematic recall.
- **Implementation**: Uses `sentence-transformers` (model: *all-MiniLM-L6-v2*) to generate 384-dimensional embeddings of user/assistant messages.
- **Storage**: `faiss` (Facebook AI Similarity Search) index for efficient similarity search ($L2$ distance).
- **Usage**: Retrieved when the user asks "Who", "What", "When", or asks to "remember" something.

3.2 2. Conceptual Memory (Knowledge Graph)

- **Purpose**: Tracks specific entities, concepts, and keywords mentioned in the conversation.
- **Implementation**:

  - **Concepts**: Dictionary tracking frequency (`count`) and first appearance.
  - **Index**: Inverted index mapping `Concept` $\rightarrow$ `[Message Indices]`.
- **Optimization**: Provides $O(1)$ lookup complexity for keyword-based recall, avoiding full history scans.

3.3 3. User Context Memory

- **Purpose**: Maintains a persistent profile of the user's interests and preferences.
- **Implementation**: Dictionary tracking domain interest counters (e.g., {`"ai": 5, "cooking": 2`}).
- **Trigger**: Updated when the user expresses preference ("I like...", "I prefer...") or engages deeply with a specific domain.

3.4 4. Short-Term Memory (Buffer)

- **Purpose**: Maintains immediate conversational continuity.
- **Implementation**: A fixed-size buffer holding the last 4 messages (2 turns).
- **Policy**: Always injected into the context regardless of relevance score.

4 The Active Agent Workflow

The `process()` method orchestrates the memory lifecycle for each request:

4.1 Step 1: Strategy Selection

The agent analyzes the user's query to decide which memory stores to consult. This prevents unnecessary computation (e.g., not searching vector DB for simple greetings).

```python
def _decide_consultation_strategy(self, query: str) -> List[str]:
    strategies = ["short_term"] # Always active

    if any(w in query for w in ["who", "what", "remember"]):
        strategies.append("semantic")

    if any(w in query for w in ["concept", "summary"]):
        strategies.append("conceptual")

    return strategies
```

4.2 Step 2: Consultation & Retrieval

The agent executes queries against the selected stores in parallel.

- **Semantic**: `vector_index.search(query_vec, k=3)`
- **Conceptual**: `concept_index.get(keyword)`

4.3 Step 3: Ranking and Filtering

Raw results from different stores are merged. The agent:

1. Deduplicates messages (a message might be found by both semantic and conceptual search).
2. Sorts them chronologically to preserve conversation flow.
3. Prioritizes Short-Term memory over Long-Term recall.

5 Optimization Strategies

5.1 Token Efficiency

Instead of sending the entire conversation history (which grows linearly), the agent selects only the most relevant pieces of information.

- **Selective Recall**: Only messages relevant to the current query are retrieved.
- **Result Limit**: Semantic search is limited to top-k results.

5.2 Computational Efficiency

- **Inverted Indexing**: Conceptual memory uses a hash map for instant lookups, avoiding $O(N)$ string matching across the entire history.
- **FAISS**: Uses optimized C++ bindings for vector similarity search, scalable to thousands of vectors.

6 Visualization

The system includes a frontend visualization of the **Knowledge Graph**.

- **Nodes**: Represent concepts (size = frequency).
- **Edges**: Represent co-occurrence of concepts in the same message turn.