Exploration of Agentic RAG-based Memory Systems for LLMs

Name: Jessy Tseng
UCID: 30065611
Email: jessy.tseng@ucalgary.ca

1 - INTRODUCTION

In recent years, the field of natural language processing (NLP) has significantly advanced with the development of transformer-based large language models (LLMs) capable of understanding and generating natural language [1,2]. However, LLMs have inherent limitations due to their stateless architecture and fixed context length [3,4]. Although recent advancements have increased the context length of LLMs (e.g. 128k tokens in GPT-4o), persistent memory across multiple sessions still remains a challenge [3,4]. This constraint makes it challenging to develop applications that require long-term memory, such as personalized AI assistants that learn about their users and multi-session workflows that build on their previous interactions [3,4]. Persistent memory is important for developing AI systems that can retain historical context, such as user preferences and prior interactions [3]. Such systems could enable personalized experiences and facilitate complex workflows that require continuity [3]. Researchers are exploring the use of retrieval-augmented generation (RAG) in tandem with agentic systems to create long-term memory for LLMs: **MemGPT** (Packer et al., 2024), **MemoryBank** (Zong et al., 2024), **ReAct** (Yao et al., 2022), and many more [2,3]. This work aims to explore existing RAG-based memory systems to understand how they work. Using the insights gained, a proof-of-concept system will be developed to demonstrate a simplified approach to implementing persistent memory for LLMs.

1.1 - Definitions

Since the field agentic systems and retrieval-augmented generation is still evolving and terms are constantly changing, the following definition will be used for this work:

- **Retrieval-augmented generation** is a technique used to improve the responses generated by LLMs by supplementing contextual information from external data sources into the prompt. Typically in **Naïve RAG** we see documents being split into smaller chunks and embedded in a vector database using an embedding model like BERT during the indexing phase. In the retrieval phase, a similarity search (usually cosine similarity) is performed to retrieve the top k relevant chunks which are then incorporated into the prompt of the LLM.
- **AI agents** are able reason and act to achieve a specified goal. There are different levels of agency, but most commonly these days, we see **tool calling agents** that use an LLM to reason and tools to act. There are also **router agents** that direct user input to the most appropriate function.
- The term **tool calling** can be a bit misleading as LLMs do not actually call tools, they generate a structured response with the function to call and the appropriate arguments. The response is programmatically parsed then executed by an external system.
- **Agentic systems** consist of multiple AI agents working together to achieve a specific goal.

2 - MATERIALS AND METHODS

2.1 - Analysis of Existing Agentic RAG-based Memory Systems

The analysis of Letta's MemGPT and LangGraph's ReAct memory agents, revealed several core insights for developing RAG-based memory systems [4,5,6]. Both frameworks use a single tool calling agent to perform the memory operations (Letta's *core_memory_append() and core_memory_replace()*; LangGraph's *upsert_memory()*) and to converse with the user [5,6]. This design choice prioritizes explainability over low chat latency as the user can see what is happening at each step as the agent is performing memory operations in real time in the hot-path as opposed to in the background. These tool calling agents rely on the LLM of the agent to infer when and what to invoke memory operations on. This delegates the memory filtering to the LLM and relies on its ability to interpret the tool/argument descriptions. Both frameworks use a vector database to store memory embeddings for context retrieval. Letta uses a dynamic context retrieval approach that uses an agent triggered tool to retrieve similar memories, while LangGraph uses a fixed approach by always retrieving the top 10 similar memories [5,6]. At its core, both RAG-based memory systems store information from messages in a vector database to be later retrieved for context. Since storing every message is not practical a tool calling agent is used to filter out what to store.

2.2 - Proposed System Architecture for Simplified Implementation

The proposed system architecture splits the conversational and memory management tasks into two separate components to optimize performance: the **chatbot agent** converses with the user while the **memory agent** selectively stores messages from the conversation in the background (Figure 1). This decoupling is meant to address the perceived latency issue with the single agent design (Letta/LangGraph), where the real time memory updates would delay responses. This separation would also allow for different LLMs to be used for the chatbot and memory agents, for more flexibility. The chatbot prioritizes low latency, while the memory agent prioritizes reasoning capability to trigger the proper memory operation and to perform the memory insertion or update. To simplify the implementation, it was assumed that only user messages could introduce new information and that all AI messages would reiterate information from user messages. The chatbot agent maintains the conversation with the user using a sliding window context maintaining the latest 8192 tokens using message trimming, while the memory agent filters user messages before summarizing, embedding, and storing the information in the vector database. Memories are retrieved using a cosine similarity search using the user message as the query.

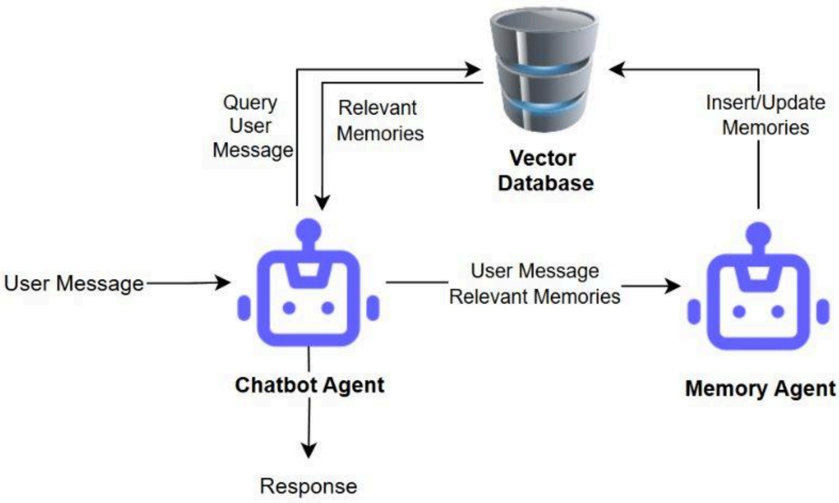

Figure 1: System architecture of the simplified implementation of an agentic RAG-based memory system.

2.3 - Implementation Details
The system was implemented using:

- **Python 3.11:** This specific version of Python was used as the **LangGraph** library package requires Python 3.11 or higher.
- **Ollama:** Host open-source local LLMs and embedding models (**llama3.2:1b-instruct-fp16** used for both the chatbot and memory agent; **nomic-text-embed** used as text embedding model).
- **LangGraph Platforms:** The **LangGraph** library was used to orchestrate multi-agent AI workflows as it had built-in vector store and Ollama integrations. **LangGraph Studio** was used for testing and **LangSmith** was used for tracing agent workflows.
- **Hardware:** The system was developed using a NVIDIA T600 (4 GB) GPU.

2.4 - System Evaluation Methodology
A multi-session chat (MSC) framework with empirical tests was used to assess the functionality of the memory system. The decision to use MSC to evaluate the system over classification metrics (like average, precision, recall, F1 score) was made due to the dynamic state of the memories stored in the database as the update memory operation can modify stored memories. If the memory system only had the insert memory operation, then the correct storage decisions and proper memory insertions could be tallied up and used to calculate classification metrics. However, with the presence of the update memory operation the classification metrics would fail to capture the complexities that come from modifying stored content. Also, MSC evaluation is more realistic as it aligns with real world use cases where memories undergo iterative refinement.

3 - RESULTS
3.1 - Evolution of the Memory Agent System Architecture
The initial architecture (Version 1) of the **memory agent** was implemented with a single tool calling agent with two memory operations: *insert_memory()* and *update_memory()*, which required the agent to infer which tool to use (Figure 2). Version 2 introduced a two-tier routing system, replacing the single tool calling agent with two specialized router agents: a **message router agent** to decide whether to store or discard an incoming user message, and an upsert **router agent** to decide whether to insert or update a memory (Figure 2). Version 3, further refined this architecture by replacing the upsert router agent with a **conflict router agent** that checks if there is conflicting information between an incoming user message and each similar memory individually: if there

is a conflict it routes it to update that memory, or if there are no conflicting memories after iterating through the similar memories it routes it to insert a memory (Figure 2). Note that in all the iterations of the memory agents, the insert and update functions use LLMs to create/update memory content.

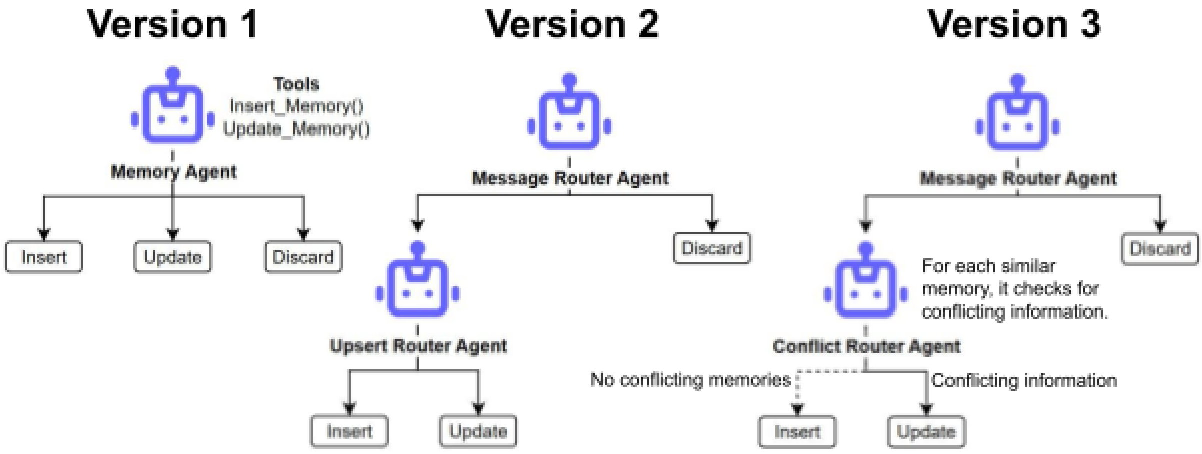

Figure 2: Evolution of the memory agent system architecture. The discard outcome was explicitly drawn to show the possible decisions the agent can make.

3.2 - Self Implemented Agentic RAG-based Memory System (Version 3) Evaluation

The self-implemented agentic RAG-based memory system demonstrated its capabilities through sequential tests involving memory insertions, updates, and cross-session recall. It initially inserted the memory that “Max, a golden retriever, loves playing fetch” into the empty database, then subsequently updated the memory, changing the breed from “golden retriever” to “Labrador mix” while retaining the original context (Table 1). The system was also able to handle non-empty database inserts, adding entries like "Max gets anxious during thunderstorms" and "Max lives with his sister Emily in Portland" (Table 1). In Test 5, the system was also able to perform cross-session memory recall, retrieving memories stored in Session 1 from Session 2 while selectively choosing what user messages to store, in this case it decided to discard the message (Table 1). However, in Test 6 the system made a mistake when appending Max’s age, the system replaced the entire memory instead of appending the new information (Table 1). Overall, the system demonstrated basic functionality in memory insertion, updates and cross-session recall.

Table 1: Multi-session chat (MSC) evaluation of the simplified implementation of agentic RAG-based memory system (Version 3).

Test No.	Test	Sess No.	Msg No.	Msg Type	Message	Mem Op	Database	Trace Link
1	Empty Insertion	1	1	Human	My dog Max is a golden retriever. He loves playing fetch.	Insert	Max, a golden retriever, loves playing fetch.	https://smith.langchain.com/public/7fabfcde-a166-480d-b460-044c2b26ea83/r
		1	2	AI	*Details Omitted*	---	---	
2	Update 1	1	3	Human	Actually, Max is a Labrador mix, not a purebred.	Update	Max, a **golden retriever**, loves playing fetch. => Max, a **Labrador mix**, loves playing fetch.	https://smith.langchain.com/public/d0953955-719b-494c-abff-d18a035deac1/r
		1	4	AI	*Details Omitted*	---	---	
3	Non-empty Insertion 1	1	5	Human	Max gets anxious during thunderstorms.	Insert	Max gets anxious during thunderstorms.	https://smith.langchain.com/public/84584a2e-145e-4589-841f-4ad47b1684f4/r
		1	6	AI	*Details Omitted*	---	---	
4	Non-empty Insertion 2	1	7	Human	Max lives with my sister Emily in Portland.	Insert	Max lives with his sister Emily in Portland.	https://smith.langchain.com/public/529f182b-721e-448e-a411-81b4aa27e6d8/r
		1	8	AI	*Details Omitted*	---	---	
5	Cross-session Memory Recall	2	9	Human	What do we know about Max?	Discard	---	https://smith.langchain.com/public/7e602288-8ce9-4280-bd14-29051016371e/r
		2	10	AI	Based on the memories I have associated with the user's query, here are some things that we know about Max: 1. **Max is a Labrador mix**: This information suggests that Max is a breed of dog, and he may be a mixed-breed or a purebred Labrador. 2. **Max lives with his sister Emily in Portland**: This indicates that Max has a sibling, which could provide insight into his personality or behavior.	---	---	

					3. **Max gets anxious during thunderstorms**: This memory suggests that Max experiences anxiety or stress in response to certain situations, such as thunderstorms. It's also worth noting that the user hasn't provided any additional information about Max, so we can only make assumptions based on the memories I have associated with this query.			
6	Update 2	1	11	Human	Max is now 5 years old.	Update	Max, a Labrador mix, loves playing fetch. => **Max is now 5 years old.**	https://smith.langchain.com/public/7639e1c0-036b-450c-9283-dbc4b7a5dcc9/r
		1	12	AI	*Details Omitted*	---	---	

4 - DISCUSSION AND CONCLUSION

4.1 - Evolution of Memory Agent System Architecture for Optimized Memory Operation Triggers

The evolution of the **memory agent** system architecture was driven by the need to improve the reliability of the memory operation triggers, when working with smaller LLMs like llama3.2:1b-instruct-fp16. In Version 1 the memory agent was implemented with a single tool calling agent with two tools: one for inserting new memories and another for updating existing ones (Figure 2). While this design was relatively straightforward, it relied on the agent's ability to infer when to use a memory operation to insert, update, or discard (do nothing) a user message as a memory. However, this strategy proved to be unreliable when working with smaller models like the 1 billion parameter 16-bit floating-point precision instruct version of Llama 3.2. The agent consistently defaulted to inserting every user message as a new memory, failing to ignore irrelevant user messages and trigger memory updates when appropriate. This behavior stems from the model's reduced reasoning capabilities as compared to the flagship, 7 billion parameter version of Llama 3.2. This made it difficult for the agent to correctly determine whether a user message contained information that needed to be inserted as a memory or update an existing memory.

To address this issue, Version 2 of the memory agent system architecture introduced a two-tier routing system, splitting the single tool calling agent into two specialized router agents (Figure 2). The rationale behind this was that since smaller LLMs had diminished reasoning capabilities, by breaking down the decision-making process into smaller, more manageable steps, smaller LLMs would be able to make the right decision improving the reliability of memory operation triggers. The **message router agent** decides whether to store or discard incoming user messages, while the **upsert router agent** decides whether to insert a new memory or update an existing one. This separation allowed for finer control over the decision-making process on when a memory operation is triggered through prompt engineering allowing the message router agent to more effectively filter out irrelevant messages. This improved the performance of the store/discard decision at the message router agent. However, the upsert router agent was struggling to distinguish between insert and update operations, and defaulted to inserting a memory each time due to the same underlying problem in that the decision being made by the agent was still too large as it was considering multiple similar memories at a time when making this decision.

With this in mind, Version 3 of the memory agent architecture replaced the upsert router agent with a **conflict router agent** (Figure 2). This new agent functions at a more granular level, programmatically comparing the incoming user message with each existing memory from the similarity search checking for conflicting information. If a conflict is detected, the system would update the relevant memory, otherwise if no conflicts were found after iterating through all memories from the similarity search, the user message would be inserted as a new memory. This approach effectively offloaded the complex reasoning from the LLM to a more deterministic, rule-based process, resulting in more reliable memory operations. Overall, these architectural changes were motivated by the need to compensate for the smaller model's diminished reasoning capabilities.

4.2 - Discussion of Self Implemented Agentic RAG-based Memory System (Version 3) Evaluation

The self-implemented agentic RAG-based memory system demonstrated promising capabilities MSC evaluation with tests involving memory insertion, updates, and cross-session recall. It was able to correctly insert memories in Test 1, 3, and 4. The system was also able to preserve the context during updates, as seen when it modified the breed of Max from "golden retriever" to "Labrador mix" while retaining the original context of playing fetch in Test 2. Additionally, the system was able to successfully perform a cross-session memory recall, retrieving memories stored in Session 1 during Session 2, which demonstrated its ability to maintain persistent memory. The system's selective storage of user messages in Test 5 further indicates the system's ability to understand relevance when filtering out irrelevant messages.

However, there were a few notable issues with the system that were revealed through this evaluation. Although the system was able to correctly discard irrelevant user messages in Test 5, the LangSmith trace revealed that the message router agent did not generate the proper output JSON, it returned {"route": "","rationale": ""} instead of {"route": "store","rationale": "..."}. This suggests that the prompt used could be further refined to ensure consistency. However, the system's data validation mechanism handled the improper JSON response and discarded the message by default. In Test 6, the system made a critical error by replacing the existing memory "Max, a Labrador mix, loves playing fetch" instead of appending the new information about Max's age in the update function. This indicates a need for further refinement in the prompt of the update function. Furthermore, there is also pronoun ambiguity in the inserting "...his sister Emily.." suggesting the need to improve the coreference resolution aspect of the prompt of the insert function. Overall, the system demonstrated basic functionality in memory insertion, updates and cross-session recall.

For future work, there are several changes that can be implemented to address the limitations of the system and to improve performance. The prompt for the message router agent should be refined to ensure it consistently outputs the proper JSON format with “store” or “discard” as the route. The update function prompt needs more explicit instructions for what to do when appending new information vs updating existing information to prevent errors like those observed in Test 6. Similarly, the insert function prompt needs additional instruction to handle coreference resolution to mitigate issues with pronoun ambiguity. The memory representation structure could be extended to explicitly capture entities, relationships and contextual information to improve memory retrieval. Different types of memory inspired by the human cognitive architecture should be explored such as episodic and semantic memory to see if it enhances the system’s flexibility. The memory agent can be extended to not only consider the current user message but also the preceding messages to improve the quality of the stored memories. From a systems architecture perspective, the memory agent could be further simplified by splitting it into two specialized agents: one to consistently insert memories in the background and another to perform batch memory updates. Finally, lightweight natural language models like spaCy’s named entity recognition (NER) model could be incorporated into the workflow to extract the entities from user messages to reduce the burden on the underlying LLM in the agent.

In conclusion, this work demonstrated a valid approach for creating an agentic RAG-based memory system with small LLMs by compensating for the reduced reasoning capabilities with thought architectural changes to break down complex memory operations into smaller manageable steps. These findings are relevant for developing applications with persistent memory and limited hardware like on-device AI assistants

REFERENCES

[1] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. arXiv:2402.06196. Retrieved from https://doi.org/10.48550/arXiv.2402.06196

[2] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. 2025. Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG. arXiv:2501.09136. Retrieved from https://arxiv.org/abs/2501.09136

[3] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A Survey on the Memory Mechanism of Large Language Model based Agents. arXiv:2404.13501. Retrieved from https://doi.org/10.48550/arXiv.2404.13501

[4] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560. Retrieved from https://doi.org/10.48550/arXiv.2310.08560

[5] Letta AI. 2025. Letta. Github. https://github.com/letta-ai/letta

[6] William Hinthorn and Harrison Chase. 2024. LangGraph ReAct Memory Agent. Github. https://github.com/langchain-ai/memory-agent