How Should I Manage Memory for my LLM Chatbot?

When building a chatbot using a Large Language Model (LLM), app developers often want to draw upon a user's previous conversations to inform newly generated content.

In order to create a highly personalized experience for each user, it's critical to have the ability to reference earlier messages and prior conversations.

For example, one of our customers is building a chatbot that acts as a medical provider/physician. Since the provider-patient relationship is long term in nature and tailored to each patient's needs, effective memory management is a crucial part of their product. Managing memory incorrectly results in awkward, generic experiences, but if you get it right, you open up exciting new ways of building products.

LLMs are inherently stateless, meaning they don't have a built-in mechanism to remember or store information from one interaction to the next. Each request to the model is processed independently, without any knowledge of previous requests or responses. This stateless nature is a fundamental characteristic of LLMs, which can pose challenges when developing applications that require context or memory.

If your users are engaged, they'll eventually build up long conversation histories, and given limited context windows & cost/latency criteria, it's vital to carefully consider how you're going to give the model the right context to respond to each message.

LLM Memory Management Options

Using LLMs with Bigger Context Window

Each response generated by an LLM is based on the context provided at runtime.
Context windows of popular models range from 4k tokens (or 3,000 words) to 128k tokens (or 96,000 words). The LLM providers that offer the biggest context windows are Anthropic and OpenAI.

As the developer of an LLM powered chatbot, you need to determine how to best leverage the context you're passing in to implement memory. Adding every conversation without modifications into the context window can quickly result in high cost, high latency and context window limitations.

Using Vector Databases to store your data

When there are a lot of prior conversations that may need to be referenced in the current user interaction, you also have the option to store this prior information in a vector database like Pinecone or Weaviate.

As the developer of this chatbot, correctly retrieving the information from a vector database and adding to the context window is another aspect to consider when referencing information from outside the current conversation.

Note: if a given conversation gets too long you could store that in a vector DB too and reference just the relevant material using metadata filtering.

Memory Management Strategies

Active Conversational Memory

While the easiest option to manage memory in a conversation is passing the full chat history in the prompt, its not the most efficient option due to cost and latency concerns.
As the conversation history grows, the model will take longer to process the input and generate a response, leading to increased latency. Additionally, the cost of running the model increases with the length of the text, making this approach potentially expensive for long conversations.

Summarization & buffering are two techniques that can help and you can use them in isolation or additively. While choosing an option here, you need to determine if you want the context from the whole conversation and/or only the most recent messages.

Here are the options available to you:

- **Summarization — Recursively summarize the conversation every time the conversation exceeds some threshold of conversations/messages:** Retains context from the whole conversation but you may lose some details;
- **Buffering — Pass in the the last N messages:** The chatbot doesn’t have memory of the older messages but has complete recent context;
- **Summarize conversation up to Nth message and pass in last N messages:** This gives the best of both worlds but has a larger number of tokens in the request;
- **Specific context retrieval:** Long conversations can be stored in a vector DB and the most relevant pieces can be added based on information in last N messages. This saves tokens but the model may entirely miss context adjacent to relevant messages in prior interactions.

Retaining Past Conversational Memory

If you're building a chatbot that needs to remember prior interactions with a user, old conversations can be stored in a vector database and referenced at runtime.

Long-Term Memory Management Strategies

Long term memory management boils down to using two strategies in tandem:

1. **Relevance:** Identifying the parts of the past conversations that are likely to be useful in generating the next message;
2. **Compression:** Reducing the number of tokens needed to represent the potentially relevant messages you'd like to pass in.

Relevance

The trick to pulling in the most relevant messages is defining what relevant means in your use case. Some common signals that a message is likely to be relevant:

- **Recency of the message** - the last few messages are more likely to be relevant than a message from last month
- **Similarity** - messages with similar embeddings or using the same rare keywords as the last few messages are more likely to be relevant
- **Session conte**xt - the user's recent actions in your product likely provide information on which messages may be more relevant

Compression

Compression, also known as Summarization, involves sending prior Chat History to a prompt with a large context window to summarize the conversation or extract critical details/keywords, and then send those as an input to another prompt that then operates on that compressed representation of chat history. You can adopt various prompt engineering techniques to compress prior conversations (e.g., simple summary, extract major themes, provide a timeline etc.), and the correct choice depends on the user experience you're trying to enable.

Keep in mind you can mix Relevance w/ Compression to compress only some subset of past chat messages.

Putting it all together for your application

A mix and match strategy between memory in conversation and from prior conversations seems most promising. Use a Prompt that takes two input variables – one is the memory from current conversation, and the other is a compressed representation of all relevant prior interactions.

As you can see, the approach to manage memory for your LLM chatbot depends a lot on the user experience you're trying to create. If you'd like tailored advice on your use case and want to build these approaches in our application without building much custom code, request a demo here. We're excited to see what you end up building with LLMs!

FAQs

What is the role of Pinecone?

Pinecone is a vector database that enables fast storage and information retrieval for your AI apps.

Why is adding memory to LLM complex?

The process is very complex because it demands rigorous testing and optimization. Also, storing a lot of new data can cause some of that information to overlap or "interfere" with other pieces of information that the model already has.