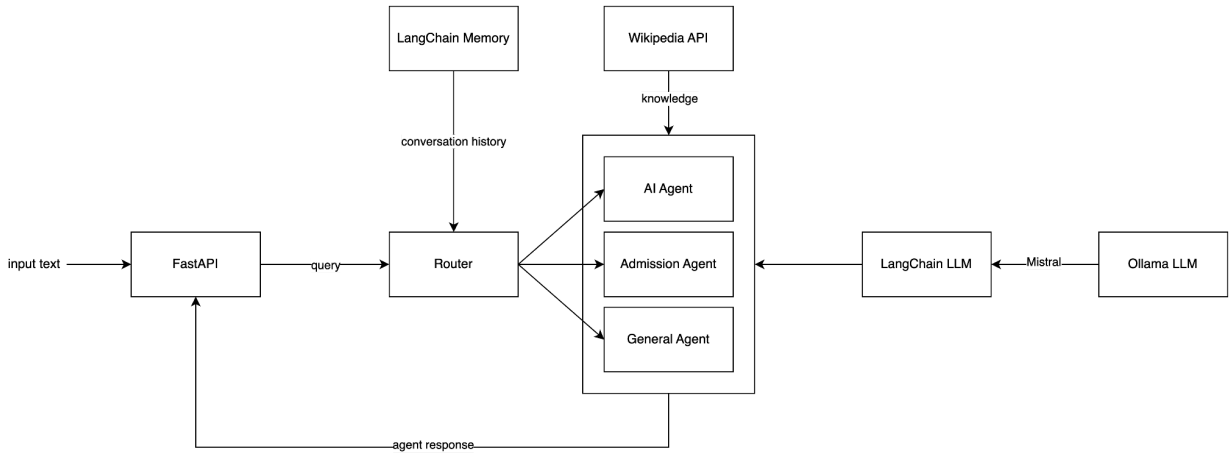
Adaptive Multi-Agent Chatbot System
Technical Report

Architecture



Agents

- AI Agent: Handles queries about artificial intelligence.
- Admission Agent: Handles queries about Concordia University's Computer Science admissions process.
- General Agent: Handles general queries

Components

- FastAPI: provides SwaggerUI interface for inputting text to query the chatbot system and displaying chatbot responses to users.
- Router: routes query to relevant chatbot for response.
- Ollama: provides the LLM model used for all agents.
- LangChain: creates an instance of an agent for each relevant task and combines the model with context, memory, maintaining chat history, and prompt engineering. Further, invokes a response from a model.
- Wikipedia API: Knowledge base to enhance agent responses.

Tech Stack

- FastAPI: For deploying the multi-agent chatbot system.
- LLM: Ollama Mistral model.
- LangChain: Memory, prompt and context engineering.
- Wikipedia API: knowledge base.

Design Decisions

Multi-Agent Architecture

LangChain is a software framework which facilitates the integration of LLMs into applications. Ollama was used to download the base model that all agents would use, Mistral, and LangChain would be used to integrate Mistral into the FastAPI framework for deploying the chatbots, using SwaggerUI on the user's local IP address to query the multi-agent chatbot system.

LangChain provides the necessary tools for setting up an agent in our Project's codebase. The LLMs are imported via Ollama and each agent requires a chat prompt template, each of which is relevant to the chatbot type. For example, the AI agent has as a prompt template "You are an expert in Artificial Intelligence." Further context and memory is added so that the model may know chat history and properly respond to the user's query. Each agent is then created using LangChain Expression Language style chaining. Each agent is invoked using the query posed by the user and is fed context and a knowledge base.

Given the three agents, AI, Concordia Admissions and General, the codebase was divided among three separate files for each. The code for constructing each chatbot, providing memory, context and a knowledge base remained the same for each, however they differed in their chat prompt template. Each had a function for handling their relevant queries and invoking the model. The query posed by the user would be routed to the relevant agent using the router.py file. It calls the relevant handle_agent() function depending on words found in the query text.

Context Awareness

To improve user experience, the chatbot tries to remember the relevant parts of your past conversions, so as to stay within context. This is achieved through keeping a record of past conversations in the form of vector embeddings. This is useful during search since it's looking for what's similar to your current input, not only matching words. That's the vector database handled by ChromaDB. Additionally, it keeps a short term memory of the most recent messages in a buffer, retaining the flow of conversation and elaborating on it. These two come together to form a relevant and contextualized chat experience.

Knowledge Integration

To enhance the chatbot's accuracy and relevance in certain topics, we integrated a dynamic knowledge base from the Wikipedia API. By doing so, we ensured that our chatbot fetches up-to-date and relevant information based on the user's query at run time. This external knowledge is added into the prompt as a separate field *knowledge_base* which is distinguished from the conversation history. To add, separating the knowledge base and the context improves the prompt clarity and the chatbot also can differentiate prior interactions which aids to the multi-turn conversation.

Multi-turn Conversation

Our chatbot saves all user inputs and the corresponding responses as conversation history. This conversation history is then passed back to the chatbot agents when making a query and will be accounted for when generating a response for a query. This allows the user to ask follow up questions without having to restate anything they have already communicated to the chatbot. The chatbot agents use OllamaLLM models. Inputs are passed through these models and generate the responses. The OllamaLLM models allow for an input parameter related to context, which can receive the saved chat history. This allows the chat history to be seamlessly integrated into models and generate cohesive conversations with the user over multiple turns/exchanges.

Challenges

The main challenge of the project was researching and understanding retrieval-augmented generation and context awareness chatbot systems. The multi-agent coordination system was difficult to understand as well but became clear with more research and by following the tutorials and reading the documentation from LangChain. However, the aspects of context-awareness and properly integrating a knowledge base were less clear and oftentimes confusing. For example, it was unclear whether chatbot history should be shared amongst all agents, how the agent can read the chatbot history as there is a special LangChain format for saved responses, as well as for prompt templates. Another issue was the integration of the knowledge base would enhance the chatbot's answers but would add too much context to the chatbot and it lost its capacity for multi-turn conversations. Whenever it was queried about a previous exchange, it would generalize the query and look up an answer using the wikipedia knowledge base. It was difficult to understand why, there should have been seemingly no issues. However, after doing some research, we realized that it was best to add the knowledge base as a separate field in the prompt template instead of combining it with the context. Overall, the project was difficult in that it required a bit of

research and was a completely new topic to every team member, thus there was a lack of experience on our parts.