Conversational AI Assistant

Project Report

Course: Generative AI and Agentic AI

Submitted By:

- Rijul, Jayraj Choudhary, Siddhant Kankaria, Jitesh Sidhani

1. Introduction and Business Objective

In the rapidly evolving landscape of artificial intelligence, conversational agents have become increasingly sophisticated. However, a one-size-fits-all approach often fails to meet diverse user needs. Our project aimed to address this by developing a highly versatile and customizable **Conversational AI Assistant**.

The primary **business objective** was to create an intelligent chatbot that not only engages in natural, stateful conversations but also adapts its communication style based on user preferences. We wanted to build a single application that could seamlessly transition between being a formal assistant, a creative partner, or even a document-specific expert.

To achieve this, we focused on implementing three core pillars of functionality:

1. **Dynamic Personas and Tones:** Allowing users to select the AI's personality (e.g., "Helpful Assistant," "Sarcastic Teenager") and tone (e.g., "Formal," "Humorous") to tailor the conversational experience.
2. **Retrieval-Augmented Generation (RAG):** Empowering the assistant with the ability to ingest user-uploaded documents (.pdf, .docx, .pptx) and answer questions based on their specific content, effectively turning it into an on-demand expert.
3. **Interactive User Experience:** Encapsulating these features within an intuitive and responsive web interface that encourages experimentation and practical use.

2. System Architecture and Technology Stack

Application Flow

The data flow within our application is straightforward. It begins with the user interacting with the Gradio UI and ends with the LLM generating a response that is then displayed back to the user.

User Input ⟶ Gradio UI ⟶ RAG Pipeline (if document is present) ⟶ LangGraph Agent ⟶ Groq LLM API ⟶ Response ⟶ Gradio UI ⟶ User

Technology Stack

- **Core Logic Framework: LangGraph**
  We chose LangGraph as the backbone of our application because of its ability to create stateful, cyclic graphs. This was perfect for a chatbot, as it allowed us to naturally manage conversational memory using an AgentState dictionary. Unlike a simple chain, LangGraph provided the structure needed to build a robust agent that remembers past interactions.
- **User Interface: Gradio**
  For the front end, Gradio was the ideal choice. It enabled us to quickly build a feature-rich web interface with interactive components like file uploads, radio buttons, and sliders. This was crucial for making the persona, tone, and RAG features easily accessible to the user.
- **Large Language Model (LLM): Groq API with Llama 3.1 8B**
  Performance was a key consideration. We selected the Groq API because its LPU inference engine offers incredibly low latency, which is essential for a real-time, engaging chat experience. The Llama 3.1 8B model provided a great balance of strong conversational ability and high-speed performance.
- **Vector Store: ChromaDB (In-Memory)**
  For our RAG pipeline, we used ChromaDB as our vector store. We specifically chose to run it in-memory rather than persisting it to disk. This decision was critical in solving file-locking issues on Windows and ensured that each new document was processed in a clean, isolated environment, which was one of the key challenges we overcame.
- **Embeddings Model: all-MiniLM-L6-v2**
  To convert document text into vector embeddings, we used this popular open-source model from Hugging Face. Running the embedding process locally meant that user documents were never sent to an external API, ensuring privacy and removing any dependency on paid embedding services.

3. Core Features and Implementation

Stateful Conversational Agent

The core of our assistant is a stateful agent built with LangGraph. We defined a simple AgentState to hold the list of messages. The graph itself is a simple, elegant loop, where the main "assistant" node calls the LLM and then waits for the next user input. This structure ensures that the entire conversation history is passed to the LLM with each turn, providing the necessary context for coherent dialogue.

The visualization of our simple yet effective graph is shown below:

Dynamic Personas and Tones via Prompt Engineering

One of the main goals of our project was to make the chatbot's personality customizable. We achieved this through careful **prompt engineering**.

- **System Prompts:** The foundation of each persona is a detailed system prompt that gives the LLM its core identity and instructions.
- **Few-Shot Prompting:** To make the personas more robust, we included examples of desired behaviour directly in the prompt (e.g., for the "Sarcastic Teenager," we provided a sample witty response).
- **Dynamic Instructions:** The selected "tone" is dynamically appended to the system prompt as an additional instruction, allowing for layered control over the final output.

Retrieval-Augmented Generation (RAG)

The RAG functionality transforms our chatbot from a general conversationalist into a document expert. Our pipeline follows the standard, effective workflow:

1. **Load:** It uses langchain_community document loaders to handle different file types (.pdf, .docx, .pptx).
2. **Split:** The loaded text is broken down into smaller, overlapping chunks using RecursiveCharacterTextSplitter. This is crucial for ensuring the model can find specific, granular pieces of information.
3. **Embed and Store:** Each chunk is converted into a vector embedding and stored in our in-memory ChromaDB instance.
4. **Retrieve:** When the user asks a question, the retriever performs a similarity search on the vector store to find the top 3 most relevant chunks of text. This context is then passed to the LLM along with the user's question.

4. RAG Pipeline Evaluation

Building a RAG system is only half the battle; we also needed to verify that it was working correctly. For this, we used the **Ragas** framework to conduct a quantitative evaluation of our pipeline's performance.

We created a test document (project_apollo_overview.pdf) with specific facts and a corresponding set of questions and ground-truth answers. We then used our RAG pipeline to generate answers and scored them against several key metrics.

Evaluation Metrics and Results

- **context_recall:** This measures if our retriever successfully found all the relevant information needed to answer the question. **Our system scored a perfect 1.0 on this metric**, proving that the retrieval part of our pipeline is highly effective.
- **answer_correctness:** This measures how factually accurate the generated answer is. Our scores were consistently high (e.g., **0.85, 0.91, 1.0**), showing that the LLM was able

to correctly synthesize answers from the provided context.
- **faithfulness:** This checks if the LLM's answer is strictly based on the provided context. Our model scored a perfect **1.0** on this metric, indicating it did not invent information.

Results Summary Table:
Question	context_recall	answer_correctness	faithfulness
Who is the team lead for Project Apollo?	1.0	0.851	1.0
What is the primary objective of the project?	1.0	0.912	1.0
What is the codename of the assistant?	1.0	1.000	1.0

These strong results gave us confidence that our RAG implementation is not only functional but also reliable and accurate.

5. Challenges Faced and Our Solutions

Throughout the development process, we encountered several technical hurdles. Overcoming them was a significant part of our learning experience.

1. **Challenge: Dependency and Visualization Issues**
   - **Problem:** We initially tried to visualize our LangGraph workflow using the pygraphviz library, which consistently failed to install on our Windows machines due to complex C-library dependencies.
   - **Solution:** We researched alternative methods and discovered that LangGraph has a built-in function to generate a **Mermaid diagram**. This approach required no complex installations and allowed us to visualize our graph directly in the notebook, solving the problem efficiently.
2. **Challenge: RAG State Management Bug**
   - **Problem:** Our biggest challenge was a persistent bug where the chatbot would answer questions based on a *previously* uploaded document, even after we "cleared" it. We realized that our ChromaDB instance was saving data to the disk and not being properly deleted between sessions.
   - **Solution:** We re-architected our RAG pipeline to use an **in-memory ChromaDB instance**. This ensured that the vector store was completely wiped clean every time the application was run or a document was cleared, permanently fixing the data leakage issue.
3. **Challenge: LLM Behavior with Structured Output**
   - **Problem:** In an early version, we tried to force the LLM to provide structured output using Pydantic models. However, this was confusing the llama-3.1-8b-instant model, causing it to fail and return validation errors.
   - **Solution:** We simplified our approach. Instead of demanding a structured output, we

requested a direct, plain-text response from the LLM. This proved to be far more robust and reliable for our conversational use case, and we removed the unnecessary Pydantic models from our code.

4. **Challenge: Windows File Locking PermissionError**
    - **Problem:** Even when we tried to delete the disk-based ChromaDB, we ran into a PermissionError on Windows because the Python process would not release its lock on the database files.
    - **Solution:** This was the final confirmation that a disk-based approach was too problematic for our interactive application. Our final move to a fully **in-memory vector store** solved this issue completely, as there were no files on disk to lock or delete.

6. Conclusion and Future Work

This project was a comprehensive journey into building a modern, feature-rich AI application. We successfully created a **Conversational AI Assistant** that meets all our initial objectives. It is customizable, context-aware, and capable of acting as a document expert through its robust RAG pipeline.

More importantly, the process of debugging and overcoming the various challenges taught us invaluable lessons about dependency management, stateful application design, and the practicalities of working with LLMs. We are proud to have built a polished, well-documented, and quantitatively evaluated application that showcases the concepts we've learned.

Future Work

While our current application is complete, there are several exciting avenues for future development:

- **Support for More File Types:** Extend the RAG pipeline to handle images, audio, or even URLs.
- **Integration of Agentic Tools:** Equip the assistant with tools like a web search (e.g., Tavily) or a calculator to answer a wider range of questions.
- **More Complex Agentic Workflows:** Expand the LangGraph agent to have multiple steps or nodes, allowing it to perform more complex tasks like summarizing a document and then emailing the summary.
- **Deployment:** Package the application and deploy it to a cloud service like Hugging Face Spaces or a dedicated server to make it accessible to a wider audience.