Agentic RAG Chatbot with Google Gemini

Table of Contents

Overview

This project presents a sophisticated, agent-based Retrieval-Augmented Generation (RAG) system designed to answer questions based on user-uploaded documents. It supports various document formats including PDF, DOCX, PPTX, CSV, and TXT. The system leverages a multi-agent architecture where specialized agents communicate via a structured message-passing system known as the Model Context Protocol (MCP). This iteration of the RAG Chatbot is powered by Google's Gemini models for both generating embeddings and producing responses, ensuring high-quality and relevant interactions.

Features

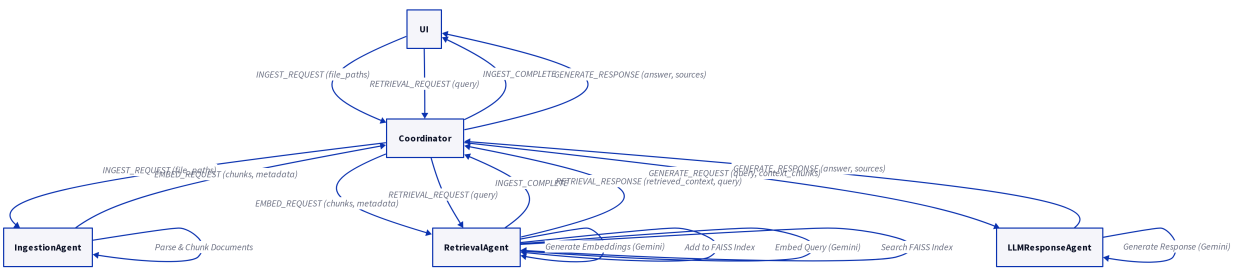
- **Multi-Format Document Support**: Seamlessly upload and process documents in PDF, DOCX, PPTX, CSV, and TXT/Markdown formats.
- **Agentic Architecture**: A modular and extensible system comprising three distinct agents: `IngestionAgent`, `RetrievalAgent`, and `LLMResponseAgent`.

- **Model Context Protocol (MCP)**: Agents communicate efficiently and transparently using an in-memory messaging protocol, facilitating clear and traceable workflows.
- **Vector Search**: Utilizes `FAISS` for rapid and efficient in-memory similarity search, enabling quick retrieval of relevant document chunks.
- **Powered by Google Gemini**: Integrates Google's cutting-edge Gemini models:
  - **Embeddings**: Employs `models/embedding-001` for generating dense vector representations of document chunks and queries.
  - **LLM**: Uses `gemini-1.5-flash` for fast, powerful, and contextually aware response generation.
- **Interactive UI**: Features a user-friendly chat interface built with Streamlit, allowing for easy document uploads, multi-turn conversations, and transparent display of source context for generated answers.

Architecture & System Flow

The application operates on a coordinator-agent pattern. The Streamlit user interface serves as the primary entry point, channeling user interactions and requests to a central `Coordinator`. The `Coordinator` is responsible for orchestrating the communication and workflow between the various specialized agents. It acts as an in-memory pub/sub mechanism for the Model Context Protocol (MCP), ensuring messages are routed to the appropriate agents for processing.

System Flow Diagram:



Tech Stack

- **UI Framework**: Streamlit
- **Core Logic**: Python 3.9+
- **LLM & Embeddings**: Google Gemini (`gemini-1.5-flash` , `embedding-001`)
- **Vector Store**: FAISS (CPU)

- **Document Parsing**: `pypdf`, `python-docx`, `python-pptx`, `pandas`
- **Text Processing**: LangChain (for text splitting)
- **Data Validation**: Pydantic (for MCP)

Setup and Installation

To get this project up and running on your local machine, follow these steps:

1. Clone the Repository

First, clone the project repository from GitHub using the following command:

Bash

```bash
git clone https://github.com/harishmogili21/Agentic-RAG-Chatbot.git
cd Agentic-RAG-Chatbot/agentic_rag_chatbot
```

2. Create a Virtual Environment

It is highly recommended to use a virtual environment to manage project dependencies and avoid conflicts with other Python projects. Choose the appropriate command for your operating system:

For Unix/macOS:

Bash

```bash
python3 -m venv venv
source venv/bin/activate
```

For Windows:

Bash

```bash
python -m venv venv
.\venv\Scripts\activate
```

3. Install Dependencies

Once your virtual environment is activated, install all the necessary packages listed in the `requirements.txt` file:

Bash

```
pip install -r requirements.txt
```

4. Set Up API Keys

This project relies on the Google Gemini API for its language model and embedding functionalities. You will need to obtain an API key:

1. Get a free API key from Google AI Studio.
2. Create a `.env` file in the root directory of the `agentic_rag_chatbot` project (i.e., `/Agentic-RAG-Chatbot/agentic_rag_chatbot/.env`).
3. Add your Google API key to this `.env` file in the following format:

How to Run the Application

After completing the setup and installation steps, you can launch the Streamlit application:

```bash
streamlit run app.py
```

Open your web browser and navigate to the local URL provided by Streamlit (typically `http://localhost:8501`).

How to Use the App

1. **Upload Documents**: Use the file uploader in the sidebar to select one or more documents (PDF, DOCX, PPTX, CSV, or TXT). The system will process these files.
2. **Confirmation**: Wait for the "Files processed and ready!" confirmation message to appear in the sidebar.
3. **Ask Questions**: Type your question related to the uploaded documents into the chat input box at the bottom of the page and press Enter.
4. **Receive Answers**: The chatbot will generate an answer based on the retrieved context from your documents. You can also expand the "View Sources" section to see the exact chunks of text used to formulate the answer.

Project Structure

The project is organized into a clear and modular structure to enhance maintainability and understanding:

Plain Text

```
/Agentic-RAG-Chatbot
├── agentic_rag_chatbot/
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── base_agent.py         # Abstract base class for all agents
│   │   ├── ingestion_agent.py    # Agent for parsing and chunking documents
│   │   ├── retrieval_agent.py    # Agent for embedding and retrieval
(Gemini)
│   │   └── response_agent.py     # Agent for generating the final LLM
response (Gemini)
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── document_parser.py    # Utility functions to parse different
file formats
│   │   └── mcp.py                # Pydantic models for the Model Context
Protocol
│   ├── app.py                    # Main Streamlit application file and
Coordinator logic
│   ├── requirements.txt          # Python dependencies
│   ├── .env                      # Environment variables (e.g., API keys) -
NOT committed to Git
│   └── README.md                 # Project documentation (this file)
└── .gitignore
```

Agents Deep Dive

This section provides a detailed look into the responsibilities and functionalities of each agent within the system.

`Coordinator`

The `Coordinator` acts as the central hub for inter-agent communication. It implements an in-memory publish-subscribe mechanism for the Model Context Protocol (MCP), routing messages from one agent to another or to the UI callback handler. It ensures that the correct agent receives and processes messages based on their `receiver` and `type` attributes.

`IngestionAgent`

Role: Responsible for processing raw documents, extracting their content, and breaking them down into manageable chunks.

Process: Upon receiving an `INGEST_REQUEST` message, it reads the specified files using `document_parser.py`, splits the content into chunks using `RecursiveCharacterTextSplitter` from

LangChain, and attaches metadata (like the source filename) to each chunk. These chunks are then sent to the `Coordinator` as an `EMBED_REQUEST` for the `RetrievalAgent`.

`RetrievalAgent`

Role: Manages the creation of vector embeddings for document chunks and retrieves the most relevant chunks based on a user query.

Process: When an `EMBED_REQUEST` is received, it uses a `SentenceTransformer` model (configured with `sentence-transformers/all-MiniLM-L6-v2`) to generate embeddings for the incoming chunks. These embeddings are then added to a `FAISS` index for efficient similarity search. The agent also persists the FAISS index and chunk metadata to disk (`faiss_index.bin`, `faiss_chunks.pkl`) for faster loading in subsequent sessions. Upon a `RETRIEVAL_REQUEST`, it embeds the user's query and performs a similarity search against the FAISS index to retrieve the top `k` most relevant document chunks. These retrieved chunks, along with the original query, are then sent back to the `Coordinator` as a `RETRIEVAL_RESPONSE`.

`LLMResponseAgent`

Role: Generates the final natural language response to the user's query, utilizing the retrieved context.

Process: Upon receiving a `GENERATE_REQUEST` message, this agent constructs a prompt for the Google Gemini LLM (`gemini-1.5-flash`). This prompt includes the user's original query and the context chunks provided by the `RetrievalAgent`. The Gemini model then generates an answer based *only* on the provided context. If no relevant context is found, it informs the user accordingly. The generated answer and the sources are then sent back to the `Coordinator` as a `GENERATE_RESPONSE` for display in the UI.

Model Context Protocol (MCP) Explained

The Model Context Protocol (MCP) is a lightweight, in-memory message-passing system that facilitates communication between the different agents in the RAG Chatbot. It is defined by the `MCPMessage` Pydantic model in `utils/mcp.py`.

`MCPMessage` Structure

```python
class MCPMessage(BaseModel):
    sender: str
    receiver: str
    type: str
```

```
    payload: Dict[str, Any]
    trace_id: Optional[str] = None
```

- `sender` : The name of the agent or component sending the message (e.g., "UI", "IngestionAgent").
- `receiver` : The name of the intended recipient agent or component (e.g., "Coordinator", "RetrievalAgent").
- `type` : A string indicating the purpose or nature of the message (e.g., "INGEST_REQUEST", "EMBED_REQUEST", "RETRIEVAL_RESPONSE", "GENERATE_RESPONSE").
- `payload` : A dictionary containing the actual data or content of the message. The structure of the payload varies depending on the `type` of the message.
- `trace_id` : An optional unique identifier to trace a sequence of related messages through the system, useful for debugging and monitoring.

Message Flow Examples

1. **Document Ingestion Flow:**
    - `UI` sends `INGEST_REQUEST` to `IngestionAgent` (via `Coordinator`).
    - `IngestionAgent` processes documents and sends `EMBED_REQUEST` to `Coordinator` (for `RetrievalAgent`).
    - `RetrievalAgent` creates embeddings and sends `INGEST_COMPLETE` to `Coordinator` (for `UI`).
2. **Query Processing Flow:**
    - `UI` sends `RETRIEVAL_REQUEST` to `RetrievalAgent` (via `Coordinator`).
    - `RetrievalAgent` retrieves context and sends `RETRIEVAL_RESPONSE` to `Coordinator` (for `LLMResponseAgent`).
    - `LLMResponseAgent` generates answer and sends `GENERATE_RESPONSE` to `Coordinator` (for `UI`).

This structured communication ensures a clear separation of concerns and a robust, traceable workflow within the multi-agent system.

Troubleshooting

- **`GOOGLE_API_KEY not found` error**: Ensure you have created a `.env` file in the `agentic_rag_chatbot` directory and added your `GOOGLE_API_KEY` as specified in the Setup and Installation section.

- `FAISS` **index not loading**: If you encounter issues with the FAISS index, try deleting `faiss_index.bin` and `faiss_chunks.pkl` files from the `agentic_rag_chatbot` directory. The system will then re-ingest documents and rebuild the index.
- **Streamlit UI not loading**: After running `streamlit run app.py`, if the browser doesn't open automatically or shows an error, manually navigate to `http://localhost:8501`.
- **Documents not processing**: Check the console output for any errors during document upload. Ensure the uploaded file types are supported (PDF, DOCX, PPTX, CSV, TXT, MD).

Contributing

Contributions are welcome! If you have suggestions for improvements, bug fixes, or new features, please feel free to:

1. Fork the repository.
2. Create a new branch (`git checkout -b feature/YourFeature`).
3. Make your changes.
4. Commit your changes (`git commit -m 'Add some feature'`).
5. Push to the branch (`git push origin feature/YourFeature`).
6. Open a Pull Request.

License

This project is licensed under the MIT License - see the `LICENSE` file for details. (Note: A `LICENSE` file is not currently present in the repository, but it is good practice to include one.)