Agentic RAG Chatbot: RAGENT

Introduction

The **Agentic RAG Chatbot** is an intelligent question-answering system designed to provide accurate and context-aware responses by leveraging information from various user-uploaded documents. Unlike traditional chatbots, it employs a modular, **agent-based architecture** combined with **Retrieval-Augmented Generation (RAG)**, making it robust, scalable, and highly maintainable.

Overall Architecture

The system operates as a unified application where distinct, specialized "agents" collaborate to fulfill user requests.

- **User Interface (UI):** Built with Streamlit, it's the user-facing component for document uploads and chat interactions.
- **Central Dispatcher (main.py):** Orchestrates communication, routing messages between the UI and the various agents.
- **Specialized Agents:**
    - **Ingestion Agent:** Handles the processing of new documents.
    - **Retrieval Agent:** Finds relevant information for user queries.
    - **LLM Response Agent:** Generates answers using a Large Language Model.
- **Model Context Protocol (MCP):** A custom, structured messaging system that enables seamless communication between these agents.
- **Vector Store (FAISS):** A high-performance database optimized for searching numerical representations (embeddings) of text.
- **Google Gemini API:** Provides the intelligence for both converting text into searchable embeddings and generating human-like responses.

Technology Stack & Purpose

Here's a breakdown of the key technologies used and their specific roles in the project:

1. **Python 3.9+:**
    - **Purpose:** The foundational programming language for the entire application. Its rich ecosystem and readability make it ideal for AI/ML projects.
    - **Used In:** Every Python file in the project.
2. **Streamlit:**
    - **Purpose:** A powerful and user-friendly Python library for rapidly building interactive web applications. It serves as the chatbot's entire frontend, handling document uploads, chat input, and displaying responses. Its session state management is crucial for maintaining agent instances and chat history across user interactions.
    - **Used In:** main.py (for UI, session management, and app orchestration).
3. **Google Gemini API (google-generativeai library):**
    - **Purpose:** Provides access to Google's cutting-edge Large Language Models (LLMs). In this project, it's used for two critical functions:
        - **Text Embeddings:** Converting raw text (document chunks, user queries) into high-dimensional numerical vectors. These embeddings capture the semantic meaning of the text, enabling efficient similarity searches.
        - **Generative AI:** Powering the core conversational capabilities of the chatbot, allowing it to understand context and generate coherent, human-like answers.
    - **Used In:** utils/utils_vector_store.py (for generating embeddings) and agents/llm_response_agent.py (for generating responses).
4. **FAISS (Facebook AI Similarity Search):**
    - **Purpose:** A highly optimized library for efficient similarity search and clustering of dense vectors. After document chunks are converted into numerical embeddings by the Gemini API, FAISS stores these embeddings and allows for lightning-fast retrieval of the most "similar" chunks to a given query embedding. This is fundamental to the RAG process.

- **Used In:** utils/utils_vector_store.py (for indexing and searching embeddings).

5. **Document Parsing Libraries:**
    - **PyPDF2:** Specifically for extracting text from PDF documents.
    - **python-docx:** For reading and extracting text content from Microsoft Word (.docx) files.
    - **pandas:** Used for reading and processing tabular data from CSV files, converting it into a string format for text extraction.
    - **python-pptx:** For extracting text from Microsoft PowerPoint (.pptx) presentation slides.
    - **Purpose:** These libraries collectively enable the chatbot to ingest and understand information from a wide variety of document formats, making the system versatile.
    - **Used In:** utils/utils_document_processor.py.
6. **sentence-transformers:**
    - **Purpose:** A Python framework for state-of-the-art sentence, text, and image embeddings. While Gemini provides the primary embedding model, sentence-transformers is a common dependency in RAG setups, often used by other libraries (like LangChain) or for specific local embedding models. Its inclusion ensures all potential embedding-related dependencies are met.
    - **Used In:** Listed in requirements.txt to ensure compatibility and availability for embedding operations.
    -
7. **Python Virtual Environments (venv):**
    - **Purpose:** Standard practice for Python project management. It creates isolated environments for each project, ensuring that the specific versions of libraries required for this chatbot don't conflict with libraries used by other Python projects on your system.
    - **Used In:** Project setup and deployment instructions.
8. **Git / GitHub:**
    - **Purpose:** Git is a robust version control system used for tracking changes in the codebase, allowing for collaboration and easy rollback to previous versions. GitHub is a web-based platform that hosts Git repositories, providing a central location for the project's code, issue tracking, and deployment integration (like Streamlit Cloud).
    - **Used In:** Managing the project's source code history and hosting the repository

Codebase Structure & Explanation (File by File)

The project is logically divided into modules to promote modularity and separation of concerns.

1. main.py (The Application Entry Point & UI Orchestrator)

- **Purpose:** This is the primary script that runs your Streamlit web application. It initializes the entire system, manages the user interface, and acts as the central hub for inter-agent communication.
- **Key Functionality:**
    - **UI Setup:** Configures the Streamlit page, including the title, layout, and default dark theme (.streamlit/config.toml also contributes here).
    - **Session State Management:** Uses st.session_state to store instances of agents (IngestionAgent, RetrievalAgent, LLMResponseAgent) and chat history. This ensures that the application's state persists as users interact with the UI.
    - **Agent Initialization:** Creates single instances of VectorStore and each agent, passing necessary dependencies (like the VectorStore to the IngestionAgent and RetrievalAgent).
    - **send_message_to_agent(message: MCPMessage):** This crucial function is the core of your in-memory Model Context Protocol (MCP) implementation. It receives an MCPMessage, identifies the receiver agent, and then calls that agent's process_message method directly, passing the message along. This simulates a robust messaging system within a single application.
    - **User Interaction Flow:** Handles file uploads (sending DOCUMENT_UPLOAD messages) and user queries (sending QUERY_REQUEST messages), displaying the responses received from the agents.

2. mcp/mcp_message_protocol.py (Model Context Protocol Definition)

- **Purpose:** This file defines the standardized format and types of messages exchanged between different components (agents, UI) of the chatbot system. It's the "language" they all speak.
- **Key Classes:**
    - **MCPMessageType(Enum):** An enumeration that defines all possible types of messages (e.g., DOCUMENT_UPLOAD, QUERY_REQUEST, LLM_RESPONSE, ERROR). Using an enum ensures type safety and clarity.
    - **MCPMessage:** The core class representing a single message. It includes:
        - sender: Who sent the message.
        - receiver: Who the message is for.

        - message_type: The purpose of the message (from MCPMessageType).
        - payload: A dictionary containing the actual data relevant to the message (e.g., file paths, query text, retrieved chunks, LLM answers).
        - trace_id: A unique identifier to track a sequence of related messages through the system.
    - **BaseMCPMessageHandler:** A foundational class that all your agents inherit from. It provides a common _create_response_message helper method, ensuring that all agents generate their responses in a consistent MCP format.

3. agents/ Directory (The Specialized Workers)

This directory contains the Python files for each of your intelligent agents. Each agent inherits from BaseMCPMessageHandler and implements its specific logic within its process_message method.

- **agents/ingestion_agent.py (Ingestion Agent)**
    - **Purpose:** Responsible for the initial processing of raw documents, transforming them into a format suitable for retrieval.
    - **Key Functionality:**
        - Receives DOCUMENT_UPLOAD messages from the UI (via main.py).
        - Uses the DocumentProcessor to extract text from various file types (PDF, DOCX, CSV, PPTX, TXT, MD).
        - Splits the extracted text into smaller, overlapping "chunks" to optimize for LLM context windows and retrieval accuracy.
        - Uses the VectorStore to generate numerical embeddings for each chunk and adds them to the FAISS index, along with their original text and metadata (like source filename).
        - Sends an LLM_RESPONSE (acting as a status update) back to the UI, indicating the success or failure of the ingestion process.
- **agents/retrieval_agent.py (Retrieval Agent)**
    - **Purpose:** Focuses on finding the most relevant pieces of information (document chunks) from the vector store based on a user's query.
    - **Key Functionality:**
        - Receives QUERY_REQUEST messages from the UI.
        - Takes the user's query and performs a similarity search using the VectorStore (which leverages FAISS). This search identifies document chunks whose embeddings are most similar to the query's embedding.

        - Extracts the text content and source metadata from the retrieved chunks.
        - Packages this retrieved context (top chunks and their sources) into a CONTEXT_RESPONSE MCP message and forwards it to the LLMResponseAgent.

- **agents/llm_response_agent.py (LLM Response Agent)**
    - **Purpose:** The "brain" of the chatbot, responsible for interacting with the Large Language Model to generate the final answer to the user's question.
    - **Key Functionality:**
        - Receives CONTEXT_RESPONSE messages (containing query and retrieved chunks) from the RetrievalAgent. It can also handle direct QUERY_REQUEST messages if no documents are uploaded or retrieval fails.
        - **Prompt Construction:** Crafts a sophisticated prompt for the LLM that includes the user's original question and the retrieved document context. It also adds instructions for the LLM to use the provided context and cite sources.
        - **LLM Interaction (_call_llm method):** Makes a **synchronous API call** to the Google Gemini API (self.client.generate_content()). This synchronous approach is vital for compatibility with Streamlit's single-threaded execution environment, preventing asynchronous "event loop" errors.
        - **Response Generation:** Extracts the answer text from the LLM's response.
        - Sends an LLM_RESPONSE MCP message (containing the generated answer and sources) back to the UI for display.

4. utils/ Directory (Helper Utilities)

This directory contains reusable modules that provide core functionalities shared across agents.

- **utils/utils_document_processor.py (Document Processor)**
    - **Purpose:** Centralizes the logic for extracting text from various document types and then breaking that text into smaller, manageable pieces (chunks).
    - **Key Functionality:**
        - **parse_document(file_path):** Detects the file type (PDF, DOCX, CSV, PPTX, TXT, MD) based on its extension and uses the appropriate parsing library (PyPDF2, python-docx, pandas, python-pptx) to extract all text content.

        - **split_text_into_chunks(text, chunk_size, chunk_overlap):** Implements a text splitting algorithm. It takes a large block of text and divides it into segments of a specified chunk_size, with a defined chunk_overlap between consecutive chunks. This overlap helps maintain context when retrieving information.

- **utils/utils_vector_store.py (Vector Store Manager)**
    - **Purpose:** Manages the FAISS index, which is the core component for storing and searching vector embeddings.
    - **Key Functionality:**
        - **Embedding Model Initialization:** Sets up the Google Gemini embedding model (models/text-embedding-004) to convert text into numerical vectors.
        - **add_texts(texts, metadatas):** Takes a list of text chunks and their associated metadata (like original source filename, page number). It generates an embedding for each chunk using the Gemini API and then adds these embeddings to the FAISS index. It also maintains a parallel list of the original text and metadata, so retrieved embeddings can be mapped back to their original content.
        - **search(query, k):** Takes a user's query, generates its embedding, and then uses the FAISS index to quickly find the k most similar embeddings. It returns the original text and metadata of the corresponding document chunks.
        - **clear_store():** Provides a way to reset the FAISS index and clear all stored document data, allowing the user to start with a fresh knowledge base.

Conclusion

The Agentic RAG Chatbot demonstrates a robust and intelligent agent-based architecture and a structured communication protocol. This approach enhances the chatbot's ability to provide accurate, context-aware answers from diverse documents. The integration of powerful tools like Streamlit, Google Gemini API, and FAISS positions this project as a strong example of modern AI application development.