Technical Architecture: Agentic RAG Assistant

By Shahriyar Khan

1. Overview

This document outlines the technical architecture of the Agentic Retrieval-Augmented Generation (RAG) Assistant. The system leverages a Large Language Model (LLM) to answer queries by retrieving relevant information from a private document collection. It features dynamic intent dispatching to handle various user tasks like information retrieval, summarization, and content formatting. The user interacts with the agent through a web-based interface built with Streamlit.

2. System Components

The architecture is composed of several key modules, each with distinct responsibilities.

2.1. User Interface (ui.py)

- **Framework:** Streamlit
- **Responsibilities:**
  - Provides a chat-like web interface for user interaction.
  - Manages and displays the conversation history.
  - Captures user queries and sends them to the core agent logic for processing.
  - Renders the final response received from the agent.
- **Session Management:** Utilizes Streamlit's built-in session state (st.session_state) to maintain chat history and context for each user session.

2.2. Core Agent Logic (agentic_rag_assistant.py)

- **The "Brain" of the System:** This module orchestrates the entire query-handling process.
- **Responsibilities:**
  - **LLM Initialization:** Initializes and configures the connection to external LLM APIs (Google Gemini, Llama). API keys are loaded securely from the .env file.
  - **Vector DB Configuration:** Sets up the connection to the Chroma vector database and prepares the retriever component.
  - **Tool Definition:** Implements the core functionalities as distinct "tools":
    - retrieve_information: Performs the core RAG process.
    - summarize_content: Summarizes provided text.
    - format_response: Reformats text for specific outputs like Slack or email.
  - **Intent Dispatcher:** Contains the logic to dynamically determine the user's intent and route the query to the appropriate tool.
  - **Orchestration:** The run_agent() function serves as the main entry point that

manages the step-by-step execution of a user query from intent detection to final response generation.

2.3. Data Ingestion & Utilities (utils.py)

- **Role:** Handles all offline data preparation and preprocessing.
- **Responsibilities:**
  - **Document Loading:** Reads and parses source documents (e.g., PDFs) from the /documents directory using PyPDFLoader.
  - **Text Splitting:** Breaks down large documents into smaller, semantically coherent chunks using RecursiveCharacterTextSplitter.
  - **Embedding Generation:** Uses a pre-trained model (HuggingFaceEmbeddings) to convert each text chunk into a high-dimensional vector (embedding).
  - **Vector Persistence:** Loads the generated embeddings and their corresponding text chunks into the Chroma vector database for long-term storage and efficient retrieval.

2.4. Vector Database (chroma_db/)

- **Technology:** ChromaDB
- **Role:** A specialized database that stores vector embeddings.
- **Function:** Enables efficient and scalable similarity searches. Given a query vector, it can quickly find the text chunks with the most similar vector representations, which correspond to the most semantically relevant content.

2.5. Configuration & Prompts

- **.env:** A file to store environment variables, primarily secret API keys for LLM services (e.g., GOOGLE_API_KEY). This keeps sensitive data separate from the codebase.
- **prompts.json:** A structured JSON file that stores all the prompts used by the agent. This includes prompts for intent detection, question-answering, summarization, and formatting, allowing for easy modification without changing the core code.

3. Workflows and Data Flow

The system operates in two main phases: an offline **Ingestion Phase** and an online **Query Phase**.

3.1. Data Flow: Ingestion Phase (Offline)

This is the preprocessing step to populate the vector database.

1. **Load Documents:** The process is initiated by running utils.py. It starts by loading all raw files (PDFs, etc.) from the documents/ folder.
2. **Split into Chunks:** Each document is split into smaller text chunks to ensure the context provided to the LLM is focused and within its token limit.
3. **Generate Embeddings:** Each text chunk is passed through the Hugging Face

embedding model, which outputs a numerical vector for each chunk.

4. **Persist to ChromaDB:** The text chunks and their corresponding vector embeddings are stored together in the chroma_db/ directory. The database indexes these vectors for fast retrieval.

3.2. Data Flow: Query Phase (Online)

This workflow describes the end-to-end process when a user submits a query.

1. **User Input:** The user types a message in the Streamlit UI (ui.py).
2. **Agent Invocation:** The UI calls the run_agent() function in agentic_rag_assistant.py, passing the user's query and the chat history.
3. **Dynamic Intent Dispatching:**
   - The run_agent() function first sends the user's query to the Gemini LLM, using a specialized prompt from prompts.json designed for intent classification.
   - The LLM analyzes the query and determines the primary intent: RAG, SUMMARIZE, or FORMAT.
4. **Tool Execution (Conditional Logic):**
   - If Intent is RAG:
     a. The retrieve_information tool is selected.
     b. The user's query is converted into a vector embedding.
     c. This query vector is used to perform a similarity search in ChromaDB, which returns the top k most relevant text chunks.
     d. The original query and the retrieved text chunks (the "context") are combined into a new, detailed prompt.
     e. This prompt is sent to the Gemini LLM via an API call. The LLM generates a response that directly answers the query using only the provided context.
   - If Intent is SUMMARIZE:
     a. The summarize_content tool is selected.
     b. The target text (from the query or chat history) is sent to the Gemini LLM with a summarization prompt.
     c. The LLM generates and returns a concise summary.
   - If Intent is FORMAT:
     a. The format_response tool is selected.
     b. The text to be formatted is sent to the Gemini LLM with a formatting prompt (e.g., "Format this for a Slack message with bullet points").
     c. The LLM returns the newly formatted text.
5. **Return Response:** The final generated text from the selected tool is returned to ui.py.
6. **Display Output:** The Streamlit UI receives the response and displays it to the user, appending it to the chat history.

This modular architecture allows for easy extension with new tools, models, or data sources while maintaining a clear separation of concerns.