Multi-Source RAG Chatbot: Project Documentation

Sami RAJICHI

April 18, 2025

Abstract

This document details the design, implementation, and usage of the Multi-Source RAG (Retrieval-Augmented Generation) Chatbot project. The chatbot leverages Langchain and LangGraph to create a stateful agent capable of answering queries using multiple information sources: its native Large Language Model (LLM) capabilities, a private vectorstore populated with AI-focused documents, and real-time web search. The project features a Streamlit-based user interface for interaction and configuration, along with an integrated evaluation framework using LangSmith for performance monitoring and analysis. Emphasis is placed on practical implementation choices, such as utilizing the Groq API for fast LLM inference within resource constraints, and a clear explanation of the underlying graph-based logic.

Contents

1 Introduction

Modern AI chatbots are increasingly expected to provide accurate, relevant, and up-to-date information. While Large Language Models (LLMs) possess vast general knowledge, they often lack specialized domain expertise or access to real-time information. Retrieval-Augmented Generation (RAG) addresses this by retrieving relevant information from external knowledge sources before generating a response.

This project implements a multi-source RAG chatbot specifically focused on the domain of Artificial Intelligence. It aims to provide a flexible and robust system that can dynamically choose the best information source – its internal knowledge, a curated set of AI documents stored in a vector database, or live web search results – to answer user queries effectively.

I utilized Langchain for its powerful abstractions over LLM interactions and LangGraph for managing the complex, stateful decision-making process involved in selecting and utilizing different information sources. The user interface is built with Streamlit for rapid development and ease of use. A key consideration was resource accessibility, leading to the integration of the Groq API for leveraging powerful LLMs via their free tier, despite potential rate limits or intermittent server issues. The project also incorporates an evaluation suite using LangSmith to benchmark performance and track key metrics like response latency and routing accuracy.

2 Project Architecture

The project is organized into distinct modules for clarity and maintainability.

2.1 Directory Structure

```
Multi-Source-RAG-Chatbot/
 data/                 # AI documents (PDFs, TXT) for vectorstore
 src/                  # Main source code
    __init__.py
    chatbot_graph.py  # LangGraph state machine definition
    data_loader.py    # Document loading & vectorstore setup
    tools.py          # Web search tool configuration
    config.py         # Paths and constants
    app.py            # Streamlit frontend application
    utils/            # Utility functions
        __init__.py
        env_utils.py  # Environment variable handling
        file_utils.py # File system operations
        chat_utils.py # Chat UI helper functions
        eval_utils.py # Evaluation logic
 tests/                # Evaluation scripts and data
    benchmark_questions.csv # Benchmark questions for evaluation
    evaluate.py       # (Implied usage by eval_utils.py)
 .env                  # API keys and environment variables
 requirements.txt      # Python dependencies
 Dockerfile            # Containerization configuration
 docker-compose.yml    # Multi-container setup (if needed)
 README.md             # Project overview
```

2.2 Core Components

- **‘data/‘**: Contains the source documents (PDF, TXT) defining the specialized AI knowledge base.
- **‘src/data_loader.py‘**: Handles loading these documents, splitting them into manageable chunks, generating embeddings, and populating the ChromaDB vectorstore located at ‘CHROMA_PATH‘.
- **‘src/tools.py‘**: Configures the web search tool (e.g., Tavily Search API) used for retrieving up-to-date information.
- **‘src/chatbot_graph.py‘**: The core logic engine. Defines the LangGraph state machine, including nodes for routing, retrieval (vectorstore, web), generation (RAG, native LLM), and error handling.
- **‘src/app.py‘**: The Streamlit application providing the user interface. It handles user input, displays chat history, manages application state (selected model, mode), interacts with the chatbot graph, and provides controls for data/vectorstore management and evaluation.
- **‘src/utils/‘**: Contains helper functions categorized for environment variable checks (‘env_utils.py‘), file operations (‘file_utils.py‘), chat display enhancements (‘chat_utils.py‘), and evaluation execution/display (‘eval_utils.py‘).
- **‘tests/‘**: Holds the ‘benchmark_questions.csv‘ file containing questions and expected retrieval modes for evaluation, run via the Streamlit UI’s Evaluation Mode which utilizes ‘eval_utils.py‘.
- **‘.env‘**: Stores sensitive API keys (Groq, Tavily, LangSmith) and configuration settings. Crucial for security and deployment.
- **‘requirements.txt‘**: Lists all Python dependencies.
- **‘Dockerfile‘ / ‘docker-compose.yml‘**: Enable containerized deployment for consistency across environments.

3 Chatbot Logic: The LangGraph State Machine

The heart of the chatbot is the state machine defined in ‘chatbot_graph.py‘ using LangGraph. This allows for complex, conditional workflows based on the user query, chat history, and system state.

3.1 State Definition

The graph operates on a shared state object (‘ChatbotState‘), a TypedDict containing:

- ‘query‘: The current user question.
- ‘documents‘: List of retrieved document contents (from vectorstore or web).
- ‘chat_history‘: A list of previous user/assistant messages.
- ‘answer‘: The final generated response.
- ‘retrieval_mode‘: The final mode determined or used (’llm_native’, ’vectorstore’, ’web_search’, ’error’).

- ‘forced_mode‘: Optional user override (’llm_native’, ’vectorstore’, ’web_search’, or None).
- ‘generation_source‘: Tracks the source used for RAG generation (’vectorstore’ or ’web_search’).
- ‘error‘: Stores any error message encountered during execution.

3.2 Query Routing

The initial routing determines the best approach to answer the query.

1. **Forced Mode Check (‘route_or_generate‘ edge):** If the user explicitly selects a mode (e.g., "Vectorstore Only") via the Streamlit UI (‘forced_mode‘), the graph bypasses the router and directly enters the corresponding retrieval or generation node.
2. **Automatic Routing (‘route_query_node‘):** If no mode is forced ("Dynamic (Default)"), this node is executed.
   - An LLM call is made using a specialized prompt (‘COMPREHENSIVE_ROUTER_PROMPT‘) that instructs the LLM to analyze the query and chat history and decide between ’llm_native’, ’vectorstore’, or ’web_search’.
   - **Keyword Overrides:** After the LLM decision, simple regex checks are performed. If the query contains keywords strongly indicating current events (’latest’, ’news’, etc.), the decision is overridden to ’web_search’. If it contains AI-specific keywords (’Transformer’, ’RAG’, ’fine-tuning’, etc.), it’s overridden to ’vectorstore’. This adds a layer of deterministic control.
   - The final decision updates the ‘retrieval_mode‘ in the state.
3. **Routing Decision (‘decide_after_router‘ edge):** Based on the ‘retrieval_mode‘ set by the router node, the graph transitions to the appropriate next step (retrieval or native generation).

Note: The current implementation focuses on simplified routing without an explicit document relevance validation step after retrieval, relying on the RAG prompt structure and retrieval quality.

3.3 Retrieval Mechanisms

- **Vectorstore Retrieval (‘retrieve_vectorstore_node‘):** If routing selects ’vectorstore’, this node queries the ChromaDB vectorstore using the user’s query. It retrieves the top-k relevant document chunks (‘k=7‘ for broader context).
  - **Success:** Updates state with retrieved ‘documents‘ and sets ‘generation_source‘ to ’vectorstore’.
  - **No Documents Found / Error:** If no documents are found or an error occurs, the behavior depends on ‘forced_mode‘. If forced, it proceeds to an error state. If automatic, it logs a warning and transitions to the ‘generate_llm_native‘ node as a fallback (‘decide_after_vectorstore_retrieval‘ edge).
- **Web Search Retrieval (‘retrieve_web_node‘):** If routing selects ’web_search’, this node uses the configured web search tool (Tavily) to find relevant online information.
  - **Success:** Updates state with retrieved web snippets as ‘documents‘ and sets ‘generation_source‘ to ’web_search’.
  - **No Results / Error:** Similar to vectorstore retrieval, failure leads to an error state if forced, or fallback to ‘generate_llm_native‘ in automatic mode (‘decide_after_web_retrieval‘ edge).

3.4 Answer Generation

- **RAG Generation (‘generate_answer_rag_node‘):** Invoked after successful retrieval from either vectorstore or web search.
  - Selects the appropriate RAG prompt (‘VECTORSTORE_RAG_PROMPT‘ or ‘WEB_SEARCH_RAG_PROMPT‘) based on the ‘generation_source‘.
  - These prompts explicitly instruct the LLM to answer *only* based on the provided context (documents/search results).
  - Formats the retrieved ‘documents‘ as context and calls the LLM with the context and the user’s query.
  - Updates the state with the generated ‘answer‘. The final ‘retrieval_mode‘ reflects the source used.
- **Native LLM Generation (‘generate_llm_native_node‘):** Used when routing selects ’llm_native’ or as a fallback from failed retrieval attempts (in automatic mode).
  - Uses a standard conversational prompt, including recent chat history.
  - Calls the LLM with the history and query.
  - Updates the state with the generated ‘answer‘ and sets ‘retrieval_mode‘ to ’llm_native’.

3.5 Error Handling

- Errors during routing, retrieval, or generation nodes update the ‘error‘ field in the state.
- Conditional edges check for errors and can redirect the flow to the ‘handle_error_node‘.
- The ‘handle_error_node‘ logs the error and sets a user-friendly error message as the final ‘answer‘, setting ‘retrieval_mode‘ to ’error’.

3.6 Graph Execution Flow

The graph execution follows these conditional paths:

1. **Entry Point:** Check ‘forced_mode‘. If set, jump directly to the corresponding node (e.g., ‘retrieve_vectorstore‘). If not set (Dynamic), go to ‘route_query‘.
2. **After ‘route_query‘:** Based on its decision (‘retrieval_mode‘), go to ‘retrieve_vectorstore‘, ‘retrieve_web‘, or ‘generate_llm_native‘.
3. **After ‘retrieve_vectorstore‘:** If documents found, go to ‘generate_answer_rag‘. If no documents/error (and not forced), go to ‘generate_llm_native‘. If error/no documents (and forced), go to ‘handle_error‘.
4. **After ‘retrieve_web‘:** If results found, go to ‘generate_answer_rag‘. If no results/error (and not forced), go to ‘generate_llm_native‘. If error/no results (and forced), go to ‘handle_error‘.
5. **End:** All generation nodes (‘generate_answer_rag‘, ‘generate_llm_native‘) and the ‘handle_error‘ node transition to the END state, returning the final ‘ChatbotState‘.

4 Technology Stack and Justifications

The project leverages several key technologies:

4.1 Groq API for LLM Inference

A significant decision was the use of the Groq API to access powerful LLMs like Llama 3 and Mixtral.

- **Justification:** Running large, high-performance models locally requires substantial computational resources (GPU memory, processing power) which were not readily available for this project. Groq provides an API with a generous free tier, enabling access to state-of-the-art models with exceptionally low latency (due to their LPU architecture). This allowed for rapid development and experimentation without incurring high costs or needing powerful hardware.
- **Trade-offs:** While beneficial for development, free tiers often come with limitations. I occasionally encountered rate limiting or '503 Internal Server Error' responses during peak usage times. This is a known characteristic of relying on shared, free resources. For production deployment, upgrading to a paid tier or exploring alternative hosting solutions (like Hugging Face Inference Endpoints, or cloud provider services) would be necessary for guaranteed reliability and uptime. However, for development and demonstrating capabilities under resource constraints, Groq proved to be an invaluable tool.

4.2 Langchain and LangGraph

- **Langchain:** Provides the core abstractions for interacting with LLMs, document loaders, text splitters, embeddings, vectorstores, and tools (like web search). It significantly simplifies the integration of these components.
- **LangGraph:** Chosen specifically for managing the multi-step, conditional logic of the chatbot. Representing the workflow as a state machine graph makes it easier to define, visualize (using 'get_graph().print_ascii()'), debug, and modify the complex decision-making process compared to deeply nested 'if/else' statements.

4.3 Vector Database (ChromaDB)

- ChromaDB is used as the vectorstore (implied by 'CHROMA_PATH'). It's an open-source, embedding database that is easy to set up locally for development. It efficiently stores document embeddings and allows for fast similarity searches required for the RAG retrieval step.

4.4 Web Search Integration (Tavily)

- The Tavily Search API is integrated via Langchain's tools to provide real-time web search capabilities. This allows the chatbot to answer questions requiring current information beyond the scope of the static documents or the LLM's training data.

4.5 Frontend Framework: Streamlit

- Streamlit was chosen for its speed and ease of use in building interactive data applications and demos. It allows for rapid UI development purely in Python, making it ideal for creating the chat interface, configuration sidebar, and evaluation dashboard.

5 Setup and Installation

5.1 Prerequisites

- Python 3.12+

- Pip (Python package installer)
- Git (for cloning the repository)
- Access to the internet (for downloading packages and API calls)

5.2 Installation Steps

1. **Clone the Repository:**

```
    git clone https://github.com/sami-rajichi/Multi-Source-RAG-Chatbot.
git
    cd Multi-Source-RAG-Chatbot

```

2. **Create a Virtual Environment (Recommended)**
3. **Install Dependencies:**

```
    pip install -r requirements.txt

```

4. **Configure Environment Variables:**
   - Create a file named '.env' in the project's root directory.
   - Populate it with your API keys. You can use the template provided via the Streamlit UI (Sidebar -> Config -> Download .env Template) or create it manually:

```
            # .env file contents
            GROQ_API_KEY="gsk_..."
            TAVILY_API_KEY="tvly-..."
            LANGCHAIN_TRACING_V2="true"
            LANGCHAIN_ENDPOINT="https://api.smith.langchain.com"
            LANGCHAIN_API_KEY="ls__..."
            LANGCHAIN_PROJECT="Your-LangSmith-Project-Name" # e.g.,
"Multi-Source-Chatbot-Eval"
            # Optional: Set a default model
            EMBEDDING_MODEL_NAME=sentence-transformers/all-MiniLM-L6
-v2
            GROQ_MODEL_NAME="meta-llama/llama-4-maverick-17b-128e-
instruct"
            # Optional: Set Web Search Provider (currently only
tavily supported)
            WEB_SEARCH_PROVIDER="tavily"
```

   - Alternatively, use the "Upload .env file" feature in the Streamlit sidebar. The app will guide you if variables are missing.

5. **Add Data Files:**
   - Place your PDF and TXT documents containing the AI knowledge base into the 'data/' directory.

6. **Build Initial Vectorstore:**
   - Run the Streamlit application:

```
            streamlit run src/app.py

```

   - Navigate to the Sidebar -> Data Management tab.
   - Click the "Rebuild Vectorstore" button. This will process the files in 'data/', create embeddings, and save them to the 'chroma$_db$/'$directory(definedby'CHROMA_PATH')$.$Theappwilllikelyrerun$

6 Usage Guide

Access the chatbot's functionalities through the Streamlit interface.

6.1 Initial Configuration (Sidebar)

Before starting a chat, it's crucial to examine the sidebar:

1. **Environment Configuration:** Check the status of your '.env' file. If variables are missing, download the template or upload your '.env' file.
2. **Model Settings:**
   - Select the desired Groq LLM model from the dropdown). Changing the model will reset the chatbot engine.
   - Choose the 'Retrieval Mode':
     - **Dynamic (Default):** Allows the chatbot's router logic to automatically select the best source (LLM Native, Vectorstore, Web Search).
     - **LLM Native Only / Vectorstore Only / Web Search Only:** Forces the chatbot to use only the specified mode, bypassing the router. Useful for testing specific capabilities or when you know the desired source.
3. **Data Management:**
   - **Vectorstore Status:** Check if the vectorstore exists.
   - **Manage Vectorstore:** Delete the existing store or Rebuild it if you've added/removed files in the 'data/' directory.
   - **Manage Data Files:** Delete all source documents from the 'data/' directory.
   - **Add New Data:** Upload new PDF/TXT files. After uploading, use the "Process Uploads & Rebuild VS" button to add them to the knowledge base.

6.2 Chat Mode

This is the default mode for interacting with the chatbot.

- The main panel displays the conversation history. User messages are shown with a person icon, assistant responses with a robot icon.
- Type your query into the chat input box at the bottom and press Enter.
- The assistant will process the query based on the selected Retrieval Mode (dynamic or forced). A "Thinking..." indicator appears during processing.
- The assistant's response is displayed, along with a caption indicating the 'retrieval_mode' used (e.g., 'Mode: vectorstore', 'Mode: web_search', 'Mode: llm_native', 'Mode: error') and the processing time.
- Use the trash icon button next to the input box to clear the current chat history.

6.3 Evaluation Mode

Switch to this mode using the "Evaluation Mode" button at the top.

1. **Upload Benchmark CSV:** Upload a CSV file containing evaluation questions. The CSV should ideally have columns like ‘question_id‘, ‘question‘, and ‘expected_mode‘ (matching ’llm_native’, ’vectorstore’, ’web_search’, or ’N/A’). The ‘benchmark_questions.csv‘ in the ‘tests/‘ directory serves as an example.
2. **Select Number of Questions:** Use the slider to choose how many questions from the CSV to run the evaluation on.
3. **Start Evaluation:** Click the "Start Evaluation" button. The app will iterate through the selected questions, run each through the chatbot (using the currently selected model and *dynamic* routing), and collect results. A progress bar and status text will be shown. **Note:** There’s a built-in delay (‘REQUEST_DELAY‘) between questions to avoid hitting API rate limits.
4. **View Results:** Once complete, a summary dashboard is displayed:
   - **Metrics:** Total Questions, Error Rate, Mode Accuracy (if ‘expected_mode‘ was provided), Average Latency.
   - **Charts:** Latency distribution, Mode Correctness counts.
   - **Detailed Results Table:** Shows each question, expected vs. actual mode, mode correctness, the generated answer, latency, LangSmith Run ID (if configured), and any errors.
   - **Download Results:** A button is provided to download the detailed results table as a CSV file.

7 Evaluation and LangSmith Integration

Systematic evaluation is crucial for understanding the chatbot’s performance and identifying areas for improvement.

7.1 Evaluation Methodology

The evaluation process, orchestrated by ‘src/utils/eval_utils.py‘ and triggered via the Streamlit UI, follows these steps:

1. Load benchmark questions from the uploaded CSV (‘benchmark_questions.csv‘).
2. For each selected question:
   - Record the start time.
   - Invoke the ‘chatbot_runnable‘ with the question, an empty chat history, and importantly, *without* a ‘forced_mode‘ to test the dynamic routing logic. A ‘LangChainTracer‘ is attached to the invocation config.
   - Record the end time and calculate latency.
   - Extract the ‘answer‘, ‘actual_mode‘, and any ‘error‘ from the returned state.
   - Compare ‘actual_mode‘ with ‘expected_mode‘ (if available in the CSV) to determine ‘mode_correctness‘.
   - Store these results (question, answer, latency, modes, correctness, LangSmith run ID, error).

- Pause briefly (‘REQUEST_DELAY‘) before the next question.

3. Aggregate the results into a Pandas DataFrame.
4. Display summary statistics, charts, and the detailed results table in the Streamlit UI.

7.2 LangSmith Integration

LangSmith (https://smith.langchain.com/) is used for tracing, monitoring, and debugging the chatbot’s execution, especially during evaluation.

- **Configuration:** Requires setting the ‘LANGCHAIN_TRACING_V2‘, ‘LANGCHAIN_ENDPOINT‘, ‘LANGCHAIN_API_KEY‘, and ‘LANGCHAIN_PROJECT‘ environment variables in the ‘.env‘ file.
- **Tracing:** During evaluation (in ‘run_evaluation‘), a ‘LangChainTracer‘ instance is created and passed in the ‘RunnableConfig‘. This automatically captures detailed information about each step within the LangGraph execution (node inputs/outputs, timings, LLM calls, tool usage) and sends it to the specified LangSmith project. Each full invocation of the chatbot graph corresponds to a ’run’ in LangSmith.
- **Feedback Logging:** After each question in the evaluation loop:
  - The latency (‘latency_seconds‘) is calculated.
  - The ‘mode_correctness‘ (True/False -> 1/0) is determined.
  - Using the LangSmith client (‘Client().create_feedback‘), two pieces of feedback are explicitly logged against the specific run ID captured by the tracer:
    - A feedback item with ‘key="latency_seconds"‘ and ‘score=<calculated_latency>‘.
    - A feedback item with ‘key="mode_correctness"‘ and ‘score=<0_or_1>‘.
- **Benefits:** This allows me to go beyond simple pass/fail evaluation. In the LangSmith dashboard, I can:
  - Visualize the execution trace for each evaluation question to debug routing errors or slow steps.
  - Filter and analyze runs based on the logged feedback (e.g., "show me runs where mode was incorrect", "sort runs by latency").
  - Monitor performance trends over time if evaluations are run repeatedly (e.g., after code changes).
  - Gain deeper insights into the LLM calls made during routing and generation.

7.3 Interpreting Results

The evaluation provides insights into:

- **Routing Accuracy (‘Mode Accuracy‘):** How often does the dynamic router choose the source we expected? Low accuracy might indicate issues with the router prompt, keyword overrides, or ambiguity in the benchmark questions.
- **Latency:** Average and distribution of response times. High latency could point to slow LLM responses (check Groq status), inefficient retrieval, or complex graph logic.
- **Error Rate:** Frequency of failed runs. Investigate specific errors using the detailed results table and LangSmith traces.
- **Qualitative Assessment:** Reviewing the generated ‘answer‘ column in the detailed results helps assess response quality and relevance, even if the mode was ’correct’.

8 Development Process and AI Assistance

The development process was iterative, starting with basic RAG and gradually adding multi-source capabilities, routing logic, the Streamlit UI, and the evaluation framework.

Regarding the use of AI coding tools, it's important to clarify the specific approach taken:

- **No Integrated IDE Auto-Generation:** I did not utilize AI code generation tools directly integrated into the IDE, such as GitHub Copilot or specialized model servers providing real-time suggestions within VSCode.
- **Conversational AI Assistance:** Instead, development involved actively interacting with large language models through their respective web interfaces. Specifically:
  - **Deepseek Models (via Website):** I frequently used the latest models available on the Deepseek platform (https://chat.deepseek.com/) for brainstorming logic, understanding Langchain/LangGraph concepts, generating code snippets for specific functions (e.g., file handling, Streamlit components), debugging errors, and drafting parts of the documentation or prompts.
  - **Gemini Pro (via Google AI Studio):** Similarly, Google's Gemini Pro model accessed via AI Studio (https://aistudio.google.com/) was consulted for similar tasks – seeking explanations, code examples, alternative approaches, and refining logic.
- **Methodology:** The interaction was primarily conversational. I would describe the problem or the desired functionality, provide relevant context (like existing code snippets or error messages), and ask the AI for suggestions, explanations, or code implementations. The generated code or advice was then reviewed, adapted, integrated, and tested manually within the project codebase in VSCode. This approach leverages the AI's knowledge base and reasoning capabilities as an interactive assistant rather than an automated code writer.

This method allowed me to benefit from the power of modern LLMs for specific problem-solving and code generation tasks while maintaining full control over the final codebase and architecture.

9 Conclusion

This project successfully implements a multi-source RAG chatbot capable of leveraging LLM native knowledge, a specialized vectorstore, and web search to answer AI-related queries. By utilizing LangGraph, I created a flexible and manageable state machine for complex routing and retrieval logic. The integration with the Groq API provided access to powerful LLMs despite local resource constraints, demonstrating a practical approach for development. The Streamlit UI offers an intuitive interface for interaction, configuration, and evaluation. Furthermore, the LangSmith integration provides essential tools for monitoring, debugging, and quantitatively evaluating the chatbot's performance, particularly its routing accuracy and latency. While acknowledging the potential limitations of free-tier API usage, the project serves as a robust foundation for a sophisticated, context-aware chatbot.

A Resources

- **GitHub Repository:** https://github.com/sami-rajichi/Multi-Source-RAG-Chatbot
- **Hosted Application URL:** https://multi-source-rag-chatbot.streamlit.app/
- **Langchain Documentation:** https://python.langchain.com/

- **LangGraph Documentation:** https://python.langchain.com/docs/langgraph
- **LangSmith:** https://smith.langchain.com/
- **Groq API:** https://groq.com/
- **Streamlit Documentation:** https://docs.streamlit.io/
- **Tavily Search API:** https://tavily.com/