Technical Report: Syrian Heritage Conversational Chatbot

Eng.Modar Ibrahim , Eng.Ola AboKaram

March 2, 2025

Contents

List of Figures

Introduction

This technical report outlines the development and implementation of an AI-powered conversational chatbot focused on Syrian heritage, created as part of the AI_AGENT_FINAL_VERSION project. Designed to educate and engage users about Syrian history and culture, this chatbot leverages cutting-edge natural language processing (NLP) techniques and a modular architecture orchestrated by LangGraph. The system intelligently processes user queries, retrieves relevant information from a vector database or external sources, and generates context-aware responses while maintaining conversation memory.

Project Overview

The Syrian Heritage Chatbot is an advanced conversational agent that answers queries ranging from simple greetings to detailed historical questions about Syrian heritage. Built using LangGraph, it employs a stateful workflow with multiple nodes, including intent classification, retrieval-augmented generation (RAG), relevance grading, real-time web search, and summarization. The project integrates various technologies such as LangChain, FAISS, HuggingFace embeddings, and Groq to deliver accurate and responsive interactions.

Objectives

- Deliver precise and informative responses about Syrian history and culture.
- Support multilingual inputs by translating non-English queries.
- Maintain conversation context across interactions using summarization.
- Adaptively retrieve information from a local database or the web based on query complexity.

System Architecture

The chatbot's architecture is a directed acyclic graph managed by LangGraph, where each node represents a processing step and edges dictate the flow based on conditional logic. Below is a high-level overview of the system's components and workflow.

Components

Intent Classifier: Analyzes the user query to determine if it requires retrieval (RAG) or direct response generation.

Retriever (RAG): Fetches relevant documents from a FAISS-based vector database using embeddings.

Grader: Evaluates the relevance of retrieved documents, triggering web searches if necessary.

Web Search: Uses DuckDuckGo and Tavily APIs to gather real-time external information.

Generator: Produces the final response using the query, retrieved documents, web results, and conversation summary.

Summarizer: Updates the conversation summary after each interaction to maintain contextual memory.

Workflow

The workflow begins when a user submits a query, processed as follows:

1. **Intent Classification**: The intent_classifier node receives the query (e.g., "Hello" or "Tell me about the Battle of Yarmouk"). It uses a language model (LLM) to classify the intent:
   - Simple queries (e.g., greetings) bypass RAG and proceed to the generator.
   - Complex queries requiring historical or cultural data trigger the RAG process.
2. **Retrieval (RAG)**: If RAG is needed, the retriever node fetches the top-k relevant documents from the vector database using FAISS and HuggingFace embeddings.
3. **Grading**: The grader node assesses document relevance using an LLM. If any document is irrelevant, it sets a flag to initiate web searches.
4. **Web Search**: The tavily_search and duckduckgo_search nodes perform real-time searches, concatenating results with retrieved documents.
5. **Generation**: The generator node crafts the response using the query, documents, web results, and past conversation summary.
6. **Summarization**: The summarizer node updates the conversation summary, enabling memory for future interactions.

This dynamic flow ensures the chatbot adapts to varying query complexities while maintaining accuracy and context.

Detailed Component Description

Intent Classifier

Purpose: Determines the processing path for each query.
Implementation:

- Utilizes ChatGroq (e.g., `llama3-8b-8192`) with a structured output schema (`IntentClassifierResponseModel`).
- Checks queries against a regex pattern of allowed words (e.g., "hello—marhaba—hi") in

`constants.py`. Matches bypass RAG.

- For other queries, an LLM evaluates if RAG is needed and translates non-English inputs (e.g., "" → "Hello").

Code: `intent_classifier.py` uses prompts from `prompts.py` to return answer ("yes"/"no") and translation.

Retriever (RAG)

Purpose: Retrieves relevant documents from a pre-built vector database.
Implementation:

- Uses FAISS for efficient similarity search with HuggingFace embeddings (`sentence-transformers/all-MiniLM-L12-v2`).
- Retrieves top-k documents (default k=2) based on the query embedding.

Code: `retriever.py` embeds the translated query and searches the FAISS index loaded from `my_vector_db.index`.

Grader

Purpose: Ensures retrieved documents are relevant to the query.
Implementation:

- Employs an LLM with a structured output (`GraderResponseModel`) to assign a binary score ("yes"/"no").
- If any document scores "no", the search flag is set to "Yes", triggering web searches.

Code: `grader.py` iterates over documents, applying the `grader_prompt` to filter relevant ones.

Web Search

Purpose: Supplements the database with real-time external data.
Implementation:

- Integrates `TavilySearchResults` and `DuckDuckGoSearchRun` tools via APIs (`TAVILY_API_KEY`).
- Results are concatenated with retrieved documents for generation.

Code: `search.py` defines `TavilySearch` and `DuckDuckGoSearch` classes, returning `tavily_docs` and `duckduck_docs`.

Generator

Purpose: Produces the final user response.
Implementation:

- Combines the query, documents, web results, and conversation summary into a prompt (`generation_prompt`).
- Uses ChatGroq to generate a context-aware response in the query's language.

Code: `generator.py` invokes the LLM, updating the state with the generated response.

Summarizer

Purpose: Maintains conversation memory.
Implementation:

- Summarizes new messages with the existing summary (if any) using an LLM.
- Deletes old messages from the state to manage memory.

Code: `summarizer.py` extends the summary and returns it with RemoveMessage instructions.

Project Structure

The codebase is organized into modular directories within `AI_AGENT`:

- **API/**: FastAPI backend (`app.py`, `schemas.py`) for handling chat requests.
- **config/**: Configuration files (`constants.py`, `prompts.py`, `settings.py`) for prompts, models, and API keys.
- **core/**: LangGraph workflow (`graph.py`, `routers.py`) orchestrating the system.
- **data_scripts/**:
  - **dataset_generation/**: Scripts (`questions_generator.py`, `question_refiner.py`) for creating training datasets.
  - **vector_db_generation/**: Script (`vector_db_generation.py`) for building the vector database.
- **models/**: Schemas and abstract classes (`schemas.py`) for state and responses.
- **notebooks/**: Jupyter notebook (`train_embeddings_intent.ipynb`) for training an SVC-based intent classifier.
- **services/**: Component implementations (`generator.py`, `grader.py`, etc.).
- **user_interface/**: Streamlit app (`user_interface.py`) for user interaction.
- **utils/**: Utility functions (`utils.py`) for text cleaning and color formatting.
- **.env**: Environment variables (e.g., `GROQ_API_KEY`, `TAVILY_API_KEY`).

Data Preparation

Vector Database Generation

Process:

- Converts PDFs (`ancient-syria.pdf`, `History_of_syria.pdf`) to text using `pytesseract` and `pdf2image`.

- Cleans text with `clean_pdf_text()` and splits it into chunks (512 characters, 128 overlap) using `RecursiveCharacterTextSplitter`.
- Embeds chunks with HuggingFace embeddings and stores them in a FAISS index (`my_vector_db.index`) with a pickle file (`my_vector_db.pkl`).

Code: `vector_db_generation.py`.

Dataset Generation

Purpose: Creates datasets for intent classification ("need rag" vs. "no need rag").
Process:

- **"Need RAG"**: Generates questions from text chunks using `QA_genration_prompt`, saved as CSV files (e.g., `data_89.csv`).
- **"No Need RAG"**: Produces simple questions (e.g., "Hello!") using `general_QA_prompt`, with rephrasing via `general_QA_prompt_rephrase`.
- Refines "need rag" questions to sound natural using `PARAPHRASE_PROMPT` in `question_refiner.py`.

Code: `questions_generator.py`, `question_refiner.py`.

User Interface

Implementation: A Streamlit application (`user_interface.py`) provides an interactive chat interface.
Features:

- Displays conversation history and summary.
- Maintains session state via `session_id`.
- Communicates with the FastAPI backend at `http://127.0.0.1:8000/chat`.

Key Design Choices and Challenges

Design Choices

- **Multilingual Support**: Translates queries in the intent classifier for inclusivity.
- **Relevance Grading**: Filters irrelevant documents to improve response quality.
- **Web Search Fallback**: Enhances robustness when local data is insufficient.
- **State Management**: Uses a State TypedDict to track conversation variables.

Challenges

- **Document Relevance**: Ensuring the grader accurately identifies relevant documents.
- **Web Search Integration**: Seamlessly combining external results with local data.
- **Memory Management**: Balancing summary updates with performance.

Technologies Used

- **LangChain**: Conversational pipeline framework.
- **LangGraph**: Workflow orchestration.
- **FAISS**: Vector similarity search.
- **HuggingFace Embeddings**: Text encoding (`all-MiniLM-L12-v2`).
- **Groq**: Fast LLM inference (e.g., `llama3-8b-8192`).
- **Streamlit**: Front-end interface.
- **FastAPI**: Backend API.
- **Tavily/DuckDuckGo**: Web search APIs.
- **Pytesseract/pdf2image**: PDF text extraction.

Conclusion

The Syrian Heritage Chatbot is a fully functional system that effectively answers queries about Syrian history and culture. It combines local retrieval with real-time web searches and maintains conversation context, offering a robust and user-friendly experience. Challenges such as document relevance and web integration were addressed through careful design and testing.

Current Status

- The chatbot is operational, with a working API and UI.
- It handles both simple and complex queries with high accuracy.

Future Improvements

- Expand the vector database with more historical texts.
- Enhance the intent classifier with SVC-based embeddings from `train_embeddings_intent.ipynb`.
- Add interactive UI features like query suggestions or multimedia responses.
- Reduce latency while improving quality.
- Improve retrieval strategy and research better methods.

This project demonstrates the power of modern NLP in preserving and sharing cultural heritage, with significant potential for further development.

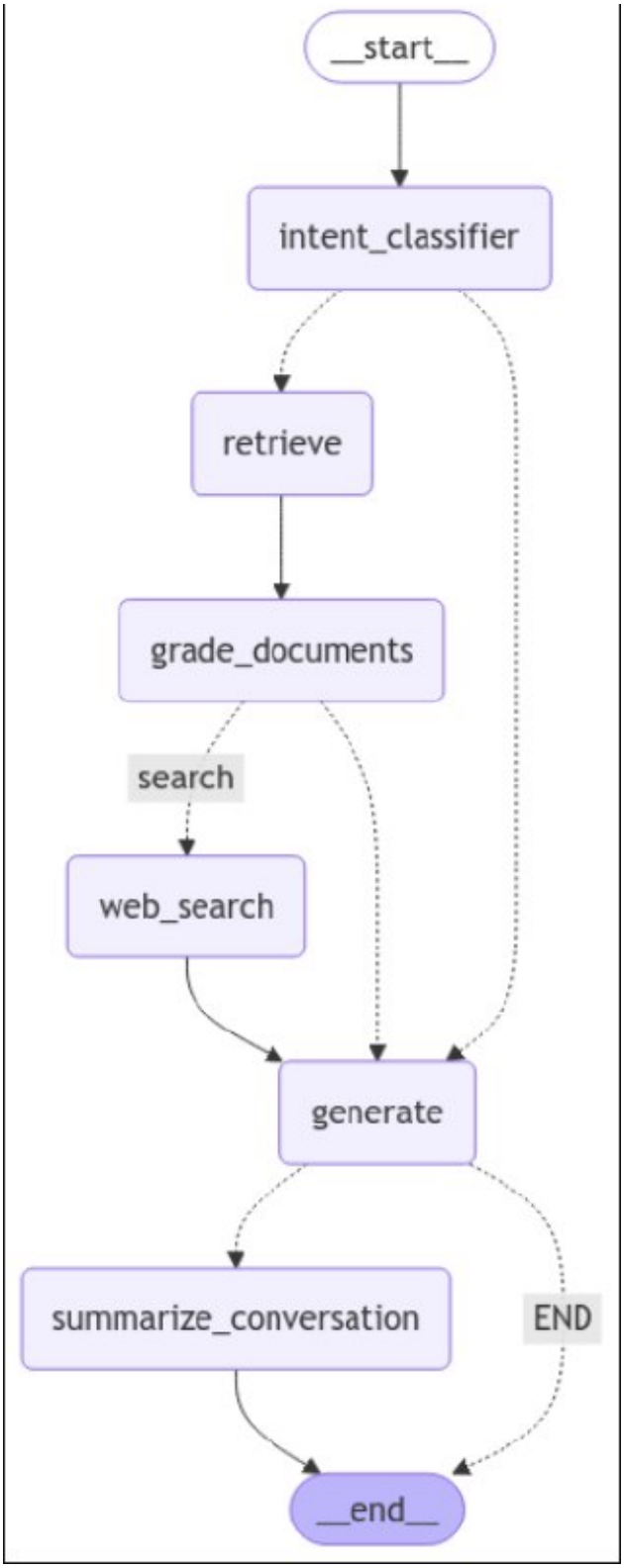

Figure 1: Workflow Graph

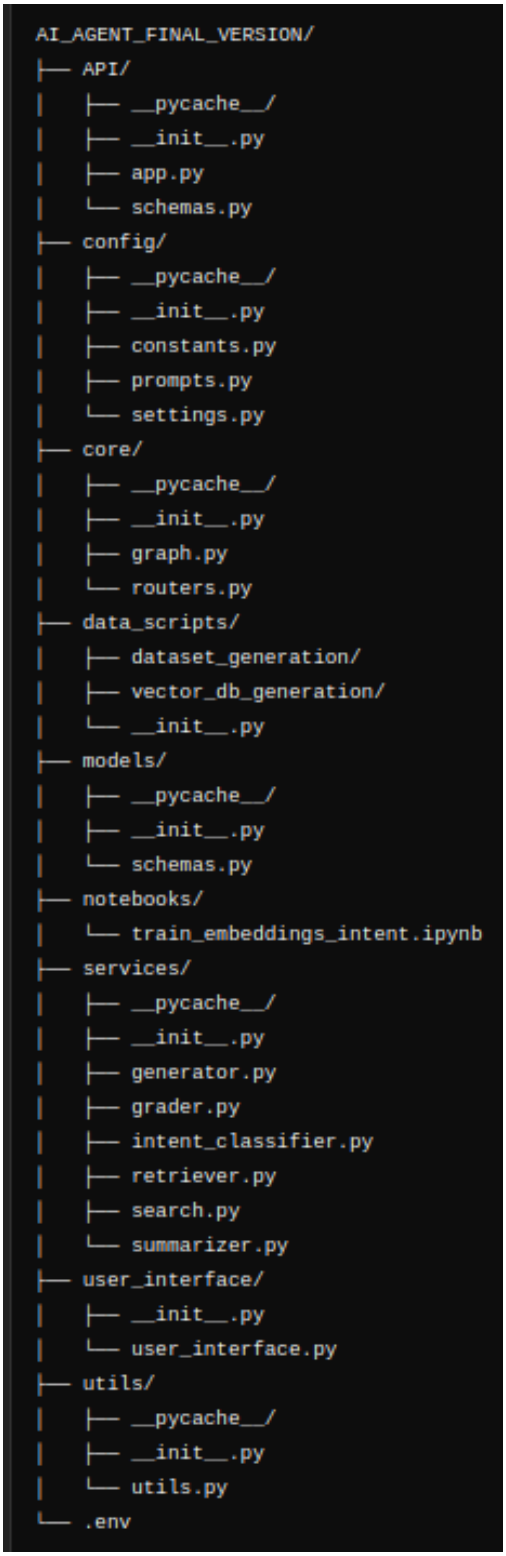

Figure 2: Directory Structure Visualization

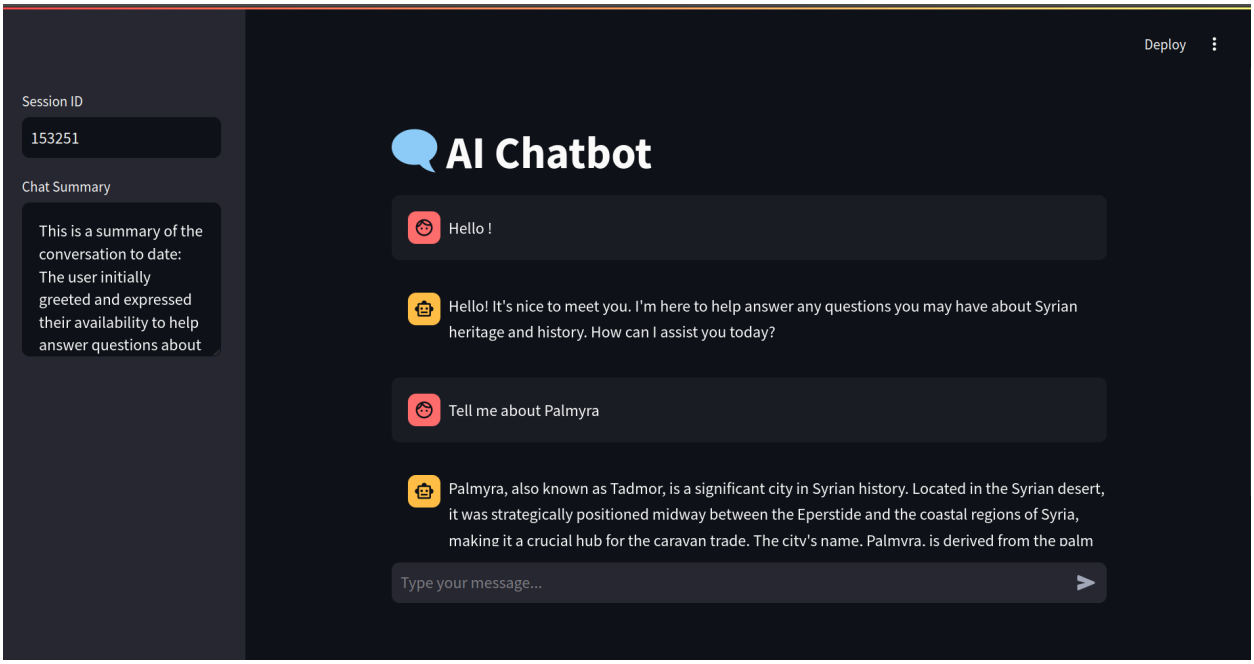

Figure 3: User Interface Screenshot

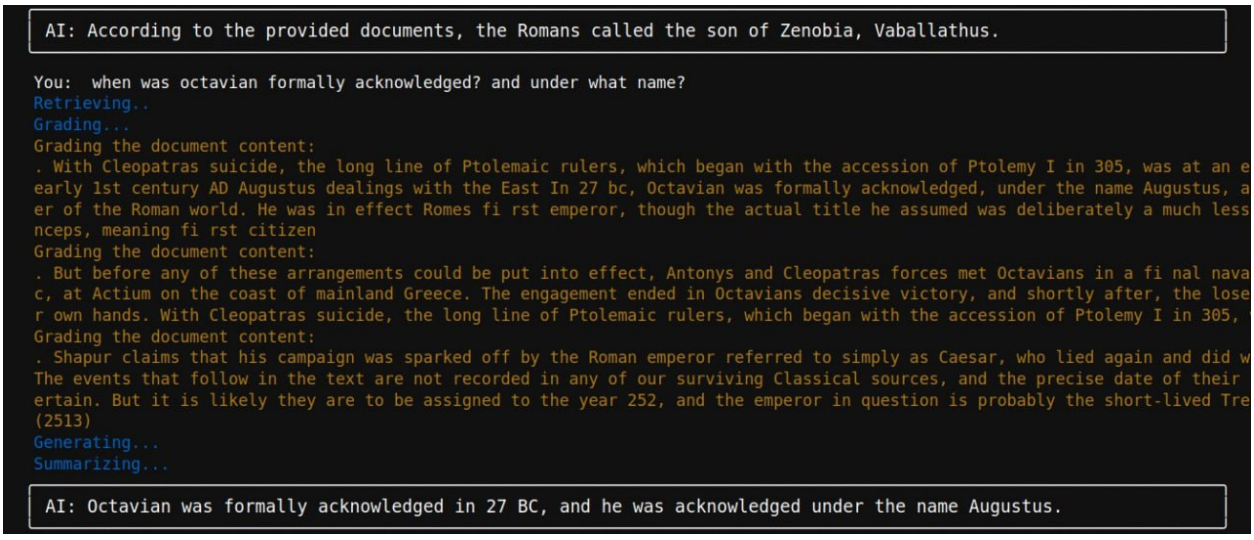

Figure 4: Example RAG process backend process 1