Generative AI COE

Technical Assessment
Job Candidate Generative AI Application

Demo

Click to be redirected to video in Google Drive

1.1 Application / Logic Flow Diagram

1, **User asks question** via Streamlit UI.

2. Langchain's conversational retrieval chain **checks** if there is any **chat history** in the same session.

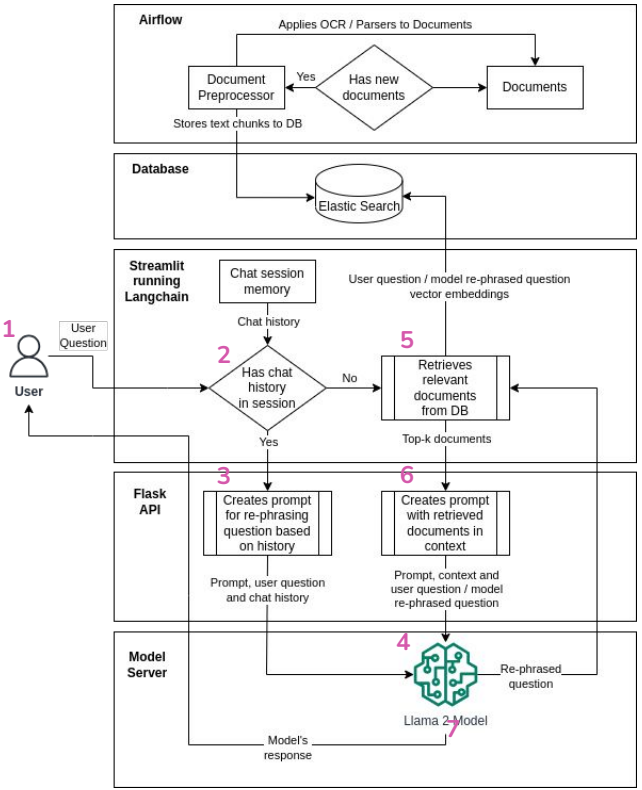
3. If **yes, user question is augmented** to prompt the LLM to re-phrase the question based on the chat history.

4. **LLM re-phrases** the original user question, responds to Flask API which streams the response to Langchain.

5. Langchain **retrieves relevant documents** based on the original user question / model re-phrased question from the Elasticsearch data store.

6. **Final prompt** is created for LLM to respond to.

7. **LLM's response is streamed to the user** via Flask API and Streamlit application running Langchain



1.2 Application / Logic Flow Insights

1. How to chunk documents?

   - All LLMs have limited input context length -> must chuck input documents and retrieve only most relevant documents.
   - But chunking process can lead to loss of contextual information of chunks.
   - Possible workarounds: Use metadata, summarise first then chunk or chunk depending on use-case (e.g. overlap, chunk size etc).

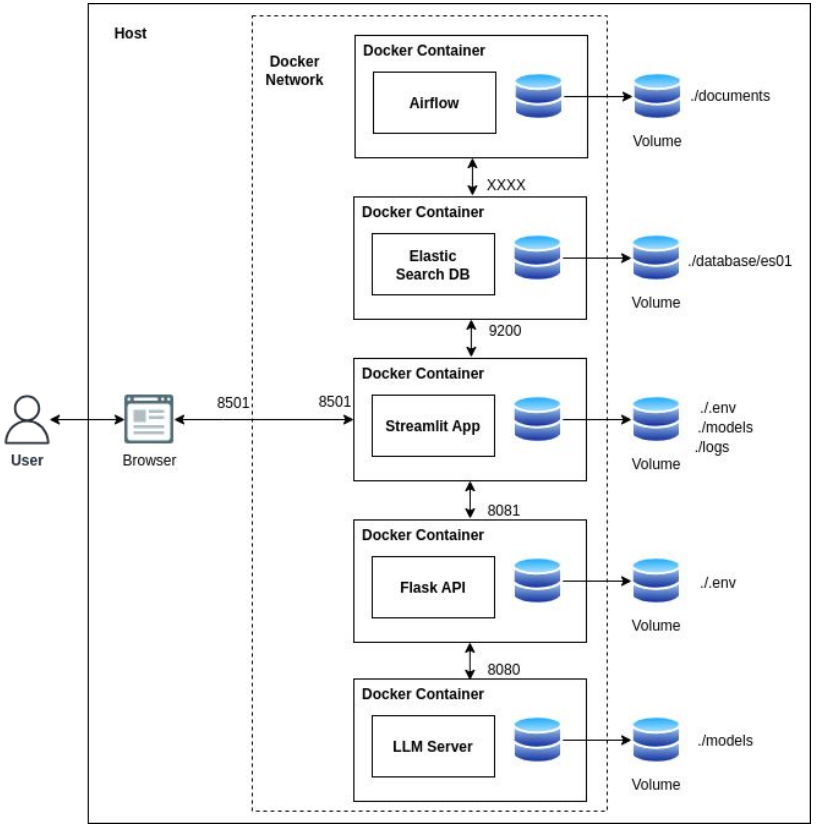
2. Which documents are more relevant?

   - Some organisations have lots of documents with same information / conflicting information -> which is more accurate?
   - Possible workarounds: Up-weigh based on metadata (recency, author's seniority or user interactions), provide source document or cluster during retrieval.

3. Prompting

   - Must engineer prompts: ensures that LLM does not answer irrelevant questions, LMM does not hallucinate and LLMs received the system prompt in the right format for model.

2.1 Architecture Diagram



Start containers with:
docker compose up

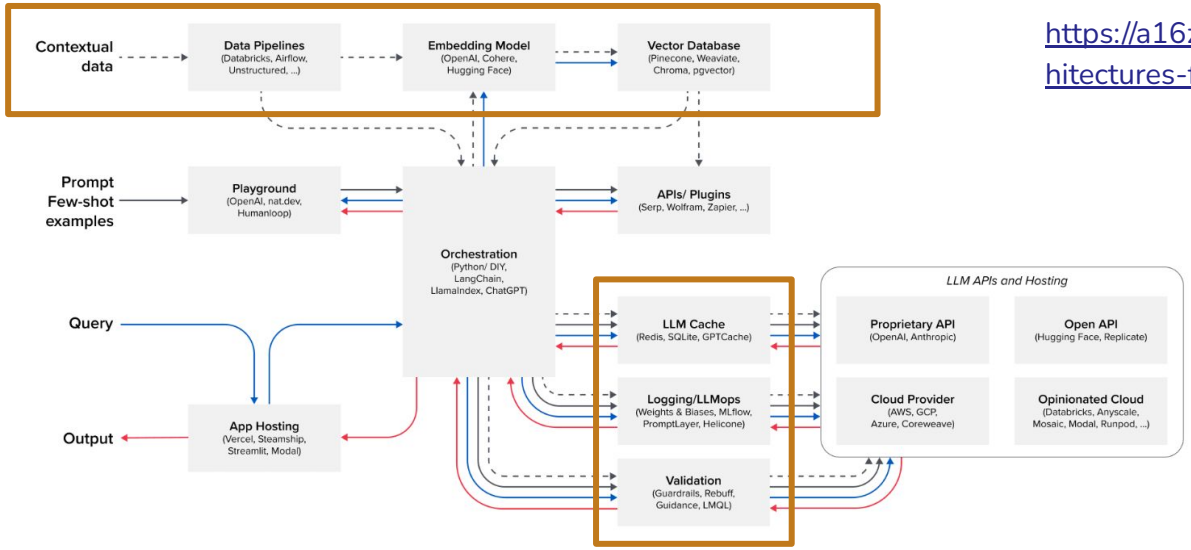
2.2 Architecture Insights

Overall deployment considerations

- Differentiation of services by frontend, backend and LLM server based on purpose.
  - Frontend Streamlit: Serves UI to users and for defining the chat application interaction logic using Langchain.
  - Backend Flask API: Pre-processes user inputs into prompts specific to the task (e.g. chatbot for job candidates based on documents) and specific prompt templates used by the LLM.
  - Model Server: Hosts the LLM - LLama 2 Chat model.
  - **Advantages**:
    - Ability to change components easily. For example, change the model server to a different model, change the UI or change the database vector store independently.
    - Ability to add new components easily. For example, using different models for rephrasing questions and responding to final questions.
    - Ability to scale vertically or horizontally depending on the load requirements. For example, to use more powerful machines for model server.
    - Ability to re-purpose stack for other tasks. For example, instead of a chatbot for a job candidate can easily change the prompts in the API to do document answering for enterprise search.
    - Allows LLM server to be shared with other applications and not specifically for one application.

Improvements - Reference Architecture

Emerging LLM App Stack



https://a16z.com/emerging-architectures-for-llm-applications/

a16z Enterprise

Enhancements

1) **Text Pre-processing**
   - Orchestrator (Airflow)
   - Documents to Text Parser (PDF, Images, HTML, Markdown etc)
   - Pre-processing methods (Entities, meta-data etc)

2) **Operational Considerations**
   - Database at Scale (ElasticSearch)
   - LLM Cache (GPTCache)
   - LLM Logging (User query, Prompts, Model Responses, User Feedback etc.)
   - LLM Response Guard Rails (Safety LLM, filters (PIIs, Bias) etc.)

Thank you!