Brance Applied AI Researcher/ML Engineer Task

Name: Kinshuk Dua

Linkedin Profile: [**www.linkedin.com/in/dua/**](http://www.linkedin.com/in/dua/)

Date Challenge Received: 12-07-23 (13-07-23 work started)

Date Solution Delivered: 14-07-23



1. Problem Statement

What was the task and how you understood it.

The task at hand is to build a Retrieval Augmented Generation (RAG) Chatbot. The user should be able to provide a file i.e. a knowledge document and get personalized answers based on the document. I wanted to build a modular, efficient and clean codebase that can be easily modified and iterated upon. I want it to support both local models as well as APIs (like OpenAI’s). With all this in mind, I decided to make a barebones API for the same that can be consumed by any frontend to create a working chatbot.

My first step was to research about existing RAG systems, libraries and papers.

One of the first paper I looked at is the seminal RAG paper- <https://arxiv.org/abs/2005.11401>

Then I looked at the more modern approaches, mainly using LangChain as an orchestrator.

I knew LangChain has a variety of integrations and agents for this particular task. However, the documentation for langchain is all over the place, there are 10 different ways to do this particular task all with their own unique quirks. After a lot of consideration, I was able to narrow down the way I chose to do it. All the reasoning and details are in the approach section.

2. Approach

Your approach to the problem. Mention any assumptions made.

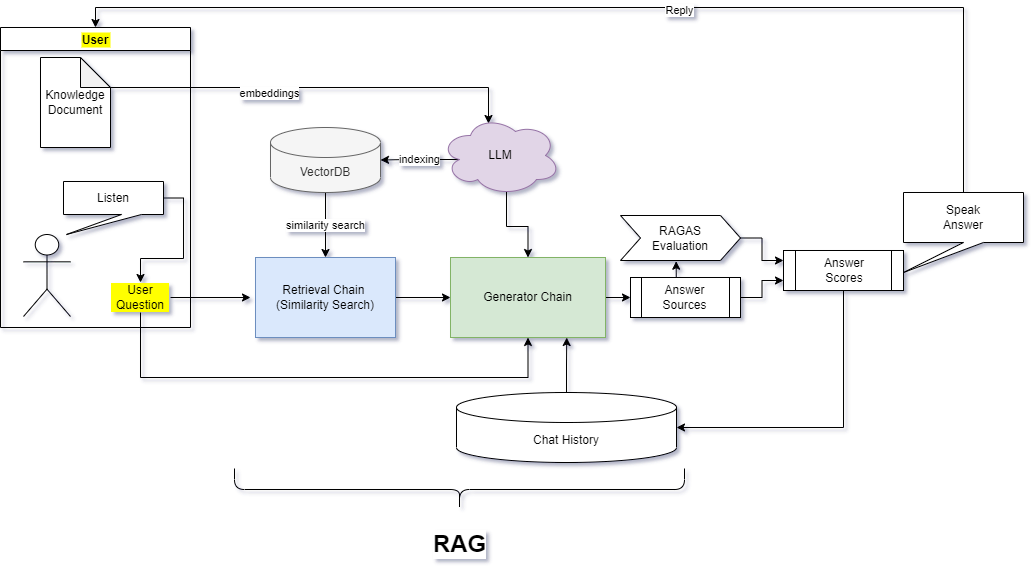
Approach

The entire system is built on a modular approach, any part that requires modification or additions can be edited directly without effecting the other modules. Langchain is used as an orchestrator, python is the sole language used in the codebase, it supports multiple LLMs.

The project has 6 core systems- a) RAG, b) LLM Integration, c) Speech, d) Output Evaluation, e) Prompts (for multi-lingual support) and f) the backend for creating the API. All of the details are explained in the next section. I won’t talk about the code much, since it is self-evident and highly clear but I would love to explain more in the interview (is possible)

3. Solution

Details about your solution. Illustrate performance and design with diagrams.



**a) RAG**

The RAG system has two parts, the **retriever** and the **generator**.

**Retriever**

In cases where the documents are large, it is ideal to split the document(s) into chunk and store the chunks through their embeddings (using the LLM used for generation). These embeddings can then easily be stored in a vectorDB and the retrieval can be based on a similarity search on the embeddings to find the relevant chunk.

There are a few different ways and libraries to do this in langchain. Some popular ones are

1. FAISS (in memory)
2. Chroma (in memory or disk based vectorDB)
3. ElastiSearch (Cloud based/separate server)

We go with Chroma for ease of protyping. However, it will be just as easy to switch to ElastiSearch (for scaling) by modifying only 3-4 lines of code.

There are two text splitters available-

1. MarkdownHeaderSplitter – Preferable cause it adds the headers as additional metadata
2. RecursiveTextSplitter – Works better for generic text.

**Generator**

The Generator, is the LLM used for rephrasing and relaying the information in a human way.

My system supports both an OpenAI APIs as well as a local llama.cpp based model (even finetuned or perhaps LoRa based variants). More details in the integration module.

The langchain method I chose was the **ConversationalRetrievalChain**. I will justify the reasoning with the alternatives below.

1. VectorStore Agent – Not concise enough for our usecase and does not support chat history. Would be better for an Autonomous agents based task.
2. Manual retrieving and prompting – Too clunky and too many lines of boilerplate code.
3. RetrievalQA – No conversational support
4. VectorstoreIndexCreator – Clunkier, no separation of Retrieval and Generator component, requires unclean calls to “load\_qa\_chain”

Therefore, the best way to do this was with ConversationalRetrievalChain, as it was built for this very purpose.

All the code is packed into a self-contained small class for modularity and testing.

**b) LLM Integration**

There is a config file <config.ini>, the code reads from the config file and prepares the required LLM and embeddings for use.

Currently supported LLMs are-

1. OpenAI API
2. Llama.cpp with support for GPU acceleration and apple silicon support

**c) Speech**

Speaking is handled via a self-contained SpeechProcessor class. It used gTTS for speaking (with automatic language detection) and offline Whisper model for listening (again with automatic language detection).

**d) Output Evaluation**

To prevent hallucination and present an evaluation for the output, there are multiple approaches-

1. Train a custom model – Too much hassle and not enough data
2. Use similarity on embedding – Not accurate enough
3. Asking an LLM itself – Might not work well + expensive API/resources

I will use the 1st approach but instead of training a custom model I will use the RAGAS library which has its own pre trained model exactly for the RAG task.

This is also a self contained function, that takes the answer, question and context and returns two scores

1. Answer Relevance
2. Context Relevance

After generating the output for the SampleQuestions, I have rated all of them too.

**e) Custom Prompts**

The ConversationRetreivalChain supports custom prompts, we leverage the fact to force the LLM to reply in the same language as the question asked. This along with auto language detecting speech module is enough to make our chatbot multi-lingual.

**f) Backend**

Technically not part of the deliverable but I decided to encapsulate all the work done in a Flask backend that creates an API for ease of use and testing.

I have also created a pytest file for each of the important modules.

4. Future Scope

Thoughts on how you could have improved the solution.

1. Adding additional file formats – easy to add, just follow the same structure as TextQuery and add PDFQuery, YoutubeQuery etc. Use the file loaders and integrations available in langchain.
2. Add support for more LLMs – Follow the integrations module and just create an embedding and llm file for the docquery module to import.
3. Scalability – Multiple document loaders can be combined (1 extra line of code). For scalable database we can switch to elastisearch
4. Additional Tests – Add additional tests in the tests folder.
5. Add dockerfile – Add a docker file for ease of deployment.
6. GUI – add a GUI via a simple frontend app