Report on Building a Multilingual Speech Recognition Model for RAG Without Training

ME20B2017 - R Soorya Narayanan

IIITDM Kancheepuram

Introduction

Overview:

To build a multilingual speech recognition model without training, using a pre-trained multilingual speech recognition model, such as Multilingual Whisper, to enable RAG to perform tasks in multiple languages.

We will work on building an AI chatbot from start-to-finish. We will be using LangChain, OpenAI, and Pinecone vector DB, to build a chatbot capable of learning from the external world using Retrieval Augmented Generation (RAG), a functioning chatbot and RAG pipeline that can hold a conversation and provide informative responses based on a knowledge base.

The chatbot will be able to take queries from audio/video files in multiple languages, which will be transcribed and translated using Whisper AI, to be given as a query to the RAG implemented chatbot. The response or result of the query will be summarized using Hugging Face Transformers.

Purpose and Objectives:

- A multilingual speech recognition model without training, using a pre-trained multilingual speech recognition model, such as Multilingual Whisper.
- Use RAG for doing Translation and Summarization
- The input will be audio/video files.

Multilingual Speech Recognition Model (Whisper AI)

Whisper is a general-purpose speech recognition model. It is trained on a large dataset of diverse audio and is also a multitasking model that can perform multilingual speech recognition, speech translation, and language identification.

Utilization of Pre-trained Models (ChatGPT - 3.5)

ChatGPT-3.5 is an AI chatbot developed by OpenAI and launched in November 2022.
It is designed to engage in conversations, answer questions, and help with various tasks.
ChatGPT-3.5 it detailed and articulate across a wide range of topics. But, the 3.5 version has also been criticized for it sometimes uneven accuracy.

Integrating RAG (Retrieval Augmented Generation)

RAG is a technique for augmenting LLM knowledge with additional, often private or real-time, data. LLMs can reason about wide-ranging topics, but their knowledge is limited to the public data up to a specific point in time that they were trained on. If you want to build AI applications that can reason about private data or data introduced after a model's cutoff date, you need to augment the knowledge of the model with the specific information it needs. The process of bringing the appropriate information and inserting it into the model prompt is known as Retrieval Augmented Generation (RAG).

Database for Pinecone vector DB

We will be using a dataset sourced from the Llama 2 ArXiv paper and other related papers to help our chatbot answer questions about the latest and greatest in the world of GenAI.

source : https://huggingface.co/datasets/jamescalam/llama-2-arxiv-papers-chunked/tree/main

Model Development

Building a Chatbot (no RAG)

We will be relying heavily on the LangChain library to bring together the different components needed for our chatbot. To begin, we'll create a simple chatbot without any retrieval augmentation. We do this by initializing a ChatOpenAI object. For this we do need an OpenAI API key.

```
In [1]:
import os
from langchain.chat_models import ChatOpenAI

os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY") or "YOUR_API_KEY"

chat = ChatOpenAI(
    openai_api_key=os.environ["OPENAI_API_KEY"],
    model='gpt-3.5-turbo'
)
```

Chats with OpenAI's gpt-3.5-turbo and gpt-4 chat models are typically structured (in plain text) like this:

```
System: You are a helpful assistant.

User: Hi AI, how are you today?

Assistant: I'm great thank you. How can I help you?

User: I'd like to understand string theory.

Assistant:
```

The final "Assistant:" without a response is what would prompt the model to continue the conversation. In the official OpenAI ChatCompletion endpoint these would be passed to the model in a format like:

```
[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hi AI, how are you today?"},
    {"role": "assistant", "content": "I'm great thank you. How can I help you?"}
    {"role": "user", "content": "I'd like to understand string theory."}
]
```

In LangChain there is a slightly different format. We use three message objects like so:

In [2]:

```python
from langchain.schema import (
    SystemMessage,
    HumanMessage,
    AIMessage
)

messages = [
    SystemMessage(content="You are a helpful assistant."),
    HumanMessage(content="Hi AI, how are you today?"),
    AIMessage(content="I'm great thank you. How can I help you?"),
    HumanMessage(content="I'd like to understand string theory.")
]
```

We generate the next response from the AI by passing these messages to the ChatOpenAI object.

In [3]:

```python
res = chat(messages)
res
```

Out[3]:

```
AIMessage(content="String theory is a theoretical framework in physics that aims to provide a unified description of the fu
ndamental particles and forces in the universe. It suggests that at the most fundamental level, particles are not point-lik
e entities but instead tiny, vibrating strings.\n\nHere are some key points to understand about string theory:\n\n1. Buildi
ng blocks: According to string theory, the basic building blocks of the universe are not particles but rather tiny, one-dim
ensional strings. These strings can vibrate in different ways, and the different modes of vibration give rise to different
particle properties, such as mass and charge.\n\n2. Extra dimensions: Unlike traditional theories, string theory requires m
ore than the usual three dimensions of space and one dimension of time. It suggests the presence of additional, compactifie
d dimensions that are too small for us to detect directly. These extra dimensions play a crucial role in shaping the behavi
or of the strings and can help explain certain fundamental aspects of the universe.\n\n3. Quantum mechanics and gravity: St
ring theory combines quantum mechanics and general relativity, the theory of gravity. It provides a framework for understan
ding how gravity can be described in terms of microscopic vibrating strings, resolving some of the conceptual conflicts bet
ween quantum mechanics and general relativity.\n\n4. Multiple versions: There are various versions of string theory, includ
ing Type I, Type IIA, Type IIB, heterotic SO(32), and heterotic E8×E8. These different versions arise from different assump
tions about the properties of the strings and their interactions.\n\n5. Unification of forces: One of the significant achie
vements of string theory is its potential to unify the fundamental forces of nature - gravity, electromagnetism, the strong
nuclear force, and the weak nuclear force. In string theory, these forces emerge as different vibrational modes of the stri
ngs, providing a unified description of all interactions.\n\nIt's important to note that string theory remains a highly the
oretical and mathematically complex field. It is still being actively researched, and many aspects of the theory are not ye
t fully understood or experimentally confirmed. Nonetheless, it offers intriguing possibilities for understanding the funda
mental nature of the universe.", additional_kwargs={}, example=False)
```

Dealing with Hallucinations

We have our chatbot, but as mentioned — the knowledge of LLMs can be limited. The reason for this is that LLMs learn all they know during training. An LLM essentially compresses the "world" as seen in the training data into the internal parameters of the model. We call this knowledge the parametric knowledge of the model.

By default, LLMs have no access to the external world.

The result of this is very clear when we ask LLMs about more recent information, like about the new (and very popular) Llama 2 LLM.

In [6]:
```
# add latest AI response to messages
messages.append(res)

# now create a new user prompt
prompt = HumanMessage(
    content="What is so special about Llama 2?"
)
# add to messages
messages.append(prompt)

# send to OpenAI
res = chat(messages)
```

In [7]:
```
print(res.content)
```

```
I apologize, but I'm not familiar with Llama 2. Could you please provide more context or clarify what you are referring to? T
hat way, I can try to assist you better.
```

Importing the Data

In this task, we will be importing our data. We will be using the Hugging Face Datasets library to load our data. Specifically, we will be using the "jamescalam/llama-2-arxiv-papers" dataset. This dataset contains a collection of ArXiv papers which will serve as the external knowledge base for our chatbot.

In [15]:
```
from datasets import load_dataset

dataset = load_dataset(
    "jamescalam/llama-2-arxiv-papers-chunked",
    split="train"
)

dataset
```

Out[15]:
```
Dataset({
    features: ['doi', 'chunk-id', 'chunk', 'id', 'title', 'summary', 'source', 'authors', 'categories', 'comment', 'journal
_ref', 'primary_category', 'published', 'updated', 'references'],
    num_rows: 4838
})
```

Dataset Overview

The dataset we are using is sourced from the Llama 2 ArXiv papers. It is a collection of academic papers from ArXiv, a repository of electronic preprints approved for publication after moderation. Each entry in the dataset represents a "chunk" of text from these papers.

Because most Large Language Models (LLMs) only contain knowledge of the world as it was during training, they cannot answer our questions about Llama 2 — at least not without this data.

Building the Knowledge Base

We now have a dataset that can serve as our chatbot knowledge base. Our next task is to transform that dataset into the knowledge base that our chatbot can use. To do this we must use an embedding model and vector database.

We begin by initializing our connection to Pinecone, this requires a free API key.

In [17]:
```
import pinecone

# get API key from app.pinecone.io and environment from console
pinecone.init(
    api_key=os.environ.get('PINECONE_API_KEY') or 'YOUR_API_KEY',
    environment=os.environ.get('PINECONE_ENVIRONMENT') or 'YOUR_ENV'
)
```

Then we initialize the index. We will be using OpenAI's text-embedding-ada-002 model for creating the embeddings, so we set the dimension to 1536.

In [20]:
```
import time

index_name = 'llama-2-rag'

if index_name not in pinecone.list_indexes():
    pinecone.create_index(
        index_name,
        dimension=1536,
        metric='cosine'
    )
    # wait for index to finish initialization
    while not pinecone.describe_index(index_name).status['ready']:
        time.sleep(1)

index = pinecone.Index(index_name)
```

Then we connect to the index:

In [21]:
```
index.describe_index_stats()
```

Out[21]:
```
{'dimension': 1536,
 'index_fullness': 0.0,
 'namespaces': {},
 'total_vector_count': 0}
```

Our index is now ready but it is empty. It is a vector index, so it needs vectors. As mentioned, to create these vector embeddings we will OpenAI's text-embedding-ada-002 model — we can access it via LangChain like so:

In [23]:
```
from langchain.embeddings.openai import OpenAIEmbeddings

embed_model = OpenAIEmbeddings(model="text-embedding-ada-002")
```

Using this model we can create embeddings like so:

```
In [24]:
texts = [
    'this is the first chunk of text',
    'then another second chunk of text is here'
]

res = embed_model.embed_documents(texts)
len(res), len(res[0])
```

```
Out[24]: (2, 1536)
```

From this we get two (aligning to our two chunks of text) 1536-dimensional embeddings.

We're now ready to embed and index all our our data. We do this by looping through our dataset and embedding and inserting everything in batches.

```
In [26]:
from tqdm.auto import tqdm  # for progress bar

data = dataset.to_pandas()  # this makes it easier to iterate over the dataset

batch_size = 100

for i in tqdm(range(0, len(data), batch_size)):
    i_end = min(len(data), i+batch_size)
    # get batch of data
    batch = data.iloc[i:i_end]
    # generate unique ids for each chunk
    ids = [f"{x['doi']}-{x['chunk-id']}" for i, x in batch.iterrows()]
    # get text to embed
    texts = [x['chunk'] for _, x in batch.iterrows()]
    # embed text
    embeds = embed_model.embed_documents(texts)
    # get metadata to store in Pinecone
    metadata = [
        {'text': x['chunk'],
         'source': x['source'],
         'title': x['title']} for i, x in batch.iterrows()
    ]
    # add to Pinecone
    index.upsert(vectors=zip(ids, embeds, metadata))
```

```
0%|          | 0/49 [00:00<?, ?it/s]
```

Retrieval Augmented Generation

We've built a fully-fledged knowledge base. Now it's time to connect that knowledge base to our chatbot. To do that we'll be diving back into LangChain and reusing our template prompt from earlier.

To use LangChain here we need to load the LangChain abstraction for a vector index, called a vectorstore. We pass in our vector index to initialize the object.

In [28]:
```python
from langchain.vectorstores import Pinecone

text_field = "text"  # the metadata field that contains our text

# initialize the vector store object
vectorstore = Pinecone(
    index, embed_model.embed_query, text_field
)
```

Using this vectorstore we can already query the index and see if we have any relevant information given our question about Llama 2.

We return a lot of text and it's not that clear what we need or what is relevant. Fortunately, our LLM will be able to parse this information much faster than us. All we need is to connect the output from our vectorstore to our chat chatbot. To do that we can use the same logic as we used earlier.

In [34]:
```python
def augment_prompt(query: str):
    # get top 3 results from knowledge base
    results = vectorstore.similarity_search(query, k=3)
    # get the text from the results
    source_knowledge = "\n".join([x.page_content for x in results])
    # feed into an augmented prompt
    augmented_prompt = f"""Using the contexts below, answer the query.

    Contexts:
    {source_knowledge}

    Query: {query}"""
    return augmented_prompt
```

Using this we produce an augmented prompt:

In [36]:
```python
# create a new user prompt
prompt = HumanMessage(
    content=augment_prompt(query)
)
# add to messages
messages.append(prompt)

res = chat(messages)

print(res.content)
```

```
Llama 2 is a collection of pretrained and fine-tuned large language models (LLMs) that range in scale from 7 billion to 70 bi
llion parameters. These LLMs, such as L/l.sc/a.sc/m.sc/a.sc/t.sc and L/l.sc/a.sc/m.sc/a.sc/t.sc-C/h.sc/a.sc/t.sc, are specifi
cally optimized for dialogue use cases.

The special aspect of Llama 2 is that its fine-tuned LLMs outperform open-source chat models on various benchmarks, demonstra
ting superior performance. In fact, based on humane evaluations for helpfulness and safety, Llama 2 models are considered as
potential substitutes for closed-source models. Closed-source models like ChatGPT, BARD, and Claude are heavily fine-tuned to
align with human preferences, enhancing usability and safety.

The development and release of Llama 2 contribute to the progress of AI alignment research, as it provides transparent and re
producible approaches to fine-tuning and safety. This is in contrast to closed-source models, which often lack transparency a
nd hinder community advancements in the field.
```

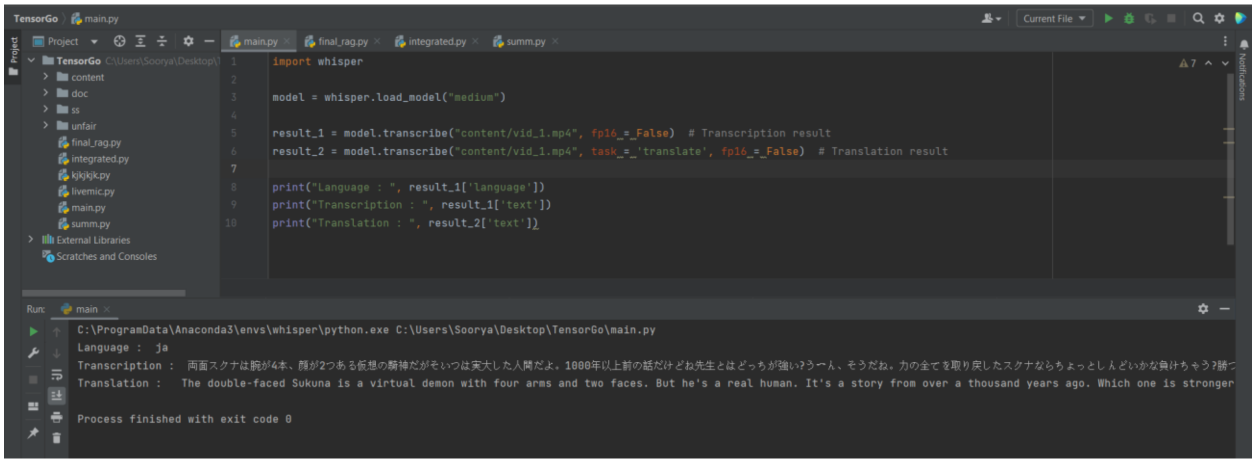
We can continue with more Llama 2 questions.

In [38]:
```python
prompt = HumanMessage(
    content=augment_prompt(
        "what safety measures were used in the development of llama 2?"
    )
)

res = chat(messages + [prompt])
print(res.content)
```

```
The safety measures used in the development of Llama 2 include safety-specific data annotation and tuning, conducting red-tea
ming, and employing iterative evaluations. These measures were taken to increase the safety of the models and ensure responsi
ble development. The paper also provides a thorough description of the fine-tuning methodology and approach to improving LLM
safety. By sharing these details and being open about the process, the aim is to enable the community to reproduce fine-tuned
LLMs and continue to improve their safety, promoting responsible development in the field.
```

Multilingual Speech recognition model (Whisper AI)

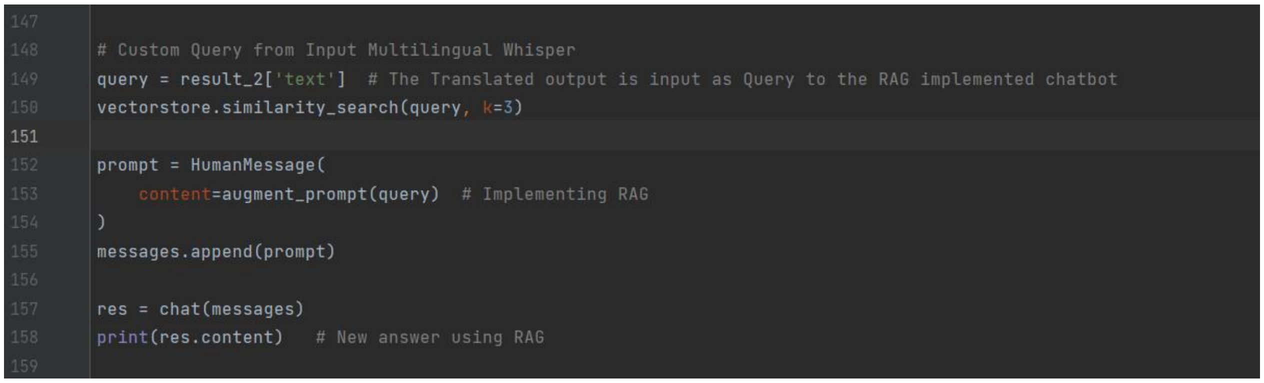
Given below is a simple implementation of Whisper AI on a video file with Japanese audio. The output of transcription and translation for the audio of the file can be seen as the output of the code:



```python
import whisper

model = whisper.load_model("medium")

result_1 = model.transcribe("content/vid_1.mp4", fp16 = False)  # Transcription result
result_2 = model.transcribe("content/vid_1.mp4", task = 'translate', fp16 = False)  # Translation result

print("Language : ", result_1['language'])
print("Transcription : ", result_1['text'])
print("Translation : ", result_2['text'])
```

```
C:\ProgramData\Anaconda3\envs\whisper\python.exe C:\Users\Soorya\Desktop\TensorGo\main.py
Language :  ja
Transcription :  両面スクナは腕が4本、顔が2つある仮想の騎神だがそいつは実大した人間だよ。1000年以上前の話だけどね先生とはどっちが強い?うーん、そうだね。力の全てを取り戻したスクナならちょっとしんどいかな負けちゃう?勝つ
Translation :   The double-faced Sukuna is a virtual demon with four arms and two faces. But he's a real human. It's a story from over a thousand years ago. Which one is stronger

Process finished with exit code 0
```

Now for the chatbot we will integrate the model such that the query is the transcribed and translated audio from a video/audio file:



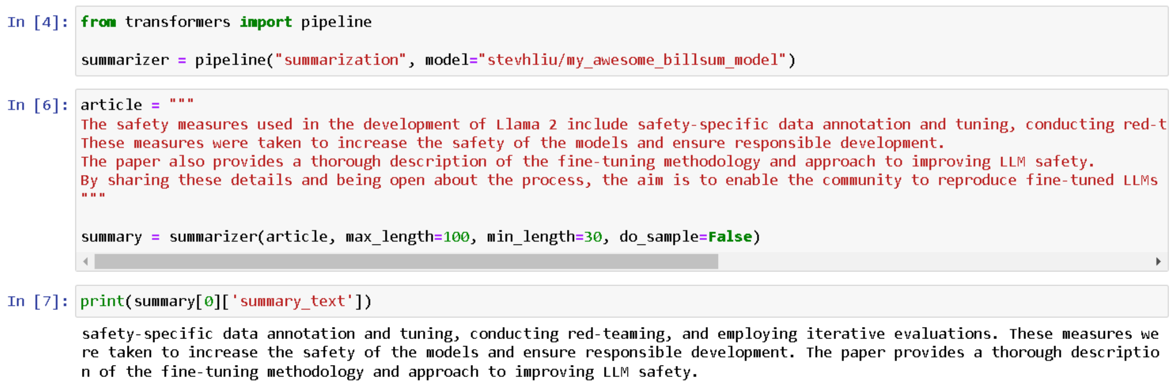
```python

# Custom Query from Input Multilingual Whisper
query = result_2['text']  # The Translated output is input as Query to the RAG implemented chatbot
vectorstore.similarity_search(query, k=3)

prompt = HumanMessage(
    content=augment_prompt(query)  # Implementing RAG
)
messages.append(prompt)

res = chat(messages)
print(res.content)   # New answer using RAG

```

Summarization

Text summarization is the process of condensing a large text document into a shorter version while preserving its key information and meaning.

Transformers are used for text summarization because they are highly effective in processing and understanding large amounts of text data.

The simplest way to try out a model for inference is to use it in a pipeline(). Instantiate a pipeline for summarization with your model, and pass your article to it:

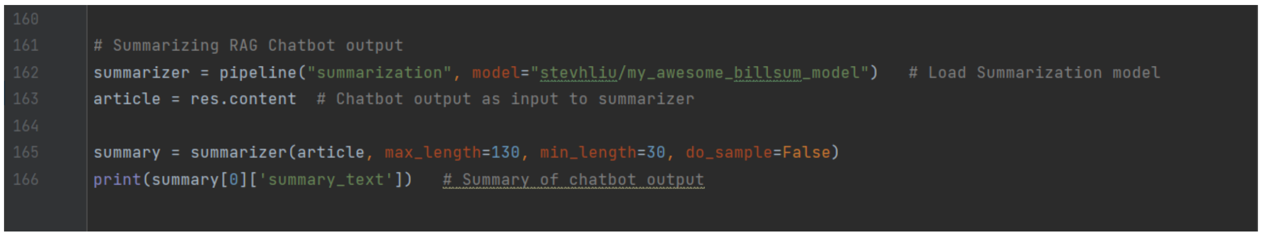


```
In [4]: from transformers import pipeline

        summarizer = pipeline("summarization", model="stevhliu/my_awesome_billsum_model")
```

```
In [6]: article = """
        The safety measures used in the development of Llama 2 include safety-specific data annotation and tuning, conducting red-t
        These measures were taken to increase the safety of the models and ensure responsible development.
        The paper also provides a thorough description of the fine-tuning methodology and approach to improving LLM safety.
        By sharing these details and being open about the process, the aim is to enable the community to reproduce fine-tuned LLMs
        """

        summary = summarizer(article, max_length=100, min_length=30, do_sample=False)
```

```
In [7]: print(summary[0]['summary_text'])

        safety-specific data annotation and tuning, conducting red-teaming, and employing iterative evaluations. These measures we
        re taken to increase the safety of the models and ensure responsible development. The paper provides a thorough descriptio
        n of the fine-tuning methodology and approach to improving LLM safety.
```

For using the query output for summarization, we can pass it to the pipeline as an article:



```

# Summarizing RAG Chatbot output
summarizer = pipeline("summarization", model="stevhliu/my_awesome_billsum_model")   # Load Summarization model
article = res.content  # Chatbot output as input to summarizer

summary = summarizer(article, max_length=130, min_length=30, do_sample=False)
print(summary[0]['summary_text'])   # Summary of chatbot output
```

Evaluation Criteria

Evaluation in this context revolves around assessing the performance and efficacy of the multilingual speech recognition model and the RAG (Retrieval Augmented Generation) system. Here are steps to carry out the evaluation:

For the Multilingual Speech Recognition Model:

- Transcription Quality Assessment:
  - Evaluate the accuracy of transcribed text generated by the model.
  - Measure against ground truth data or manually transcribed samples in various languages.
- Multilingual Capability:
  - Test the model's performance across different languages.
  - Assess its ability to transcribe accurately in various language settings.
- Comparative Analysis:
  - Compare the model's performance against existing speech recognition models.
  - Consider factors like accuracy, speed, and resource consumption.

For the RAG System:

- Translation and Summarization Tasks:
  - Utilize the transcribed text from the speech recognition model for translation and summarization.
  - Evaluate the accuracy, coherence, and relevance of the translations and summaries generated by the RAG system.
- Semantic Similarity Assessment:
  - Check how well the RAG system retrieves relevant information from the knowledge base for given queries.
  - Evaluate the semantic similarity between query-based outputs and retrieved knowledge.
- Quality of Output:
  - Assess the quality of generated responses against the context and relevance of the queries.
  - Measure the accuracy and completeness of responses provided by the RAG system.

Conclusion

I think it's very clear what sort of impact something like RAG has on the system and also just how we Implement that now this is naive or the simplest way of implementing RAG and it's assuming that there's a question with every single query which is not always going to be the case. We might say 'Hi, how are you', but obviously your chatbot doesn't need to go and refer to an external knowledge base to answer that so that is one of the downsides of using this approach but there are many benefits obviously we get this much better retrieval performance.

We get a ton of information in the vector database and can answer many more questions accurately and we can also cite where we're getting that information from this approach is much faster than other alternative rag approaches like using agents and we can also filter out the number of tokens that we only use two main tokens so we can also filter out a number of tokens of feeding back into the LLM by setting a similarity threshold so that we're not returning things that are very obviously irrelevant and if we do that that'll help us mitigate one of the other issues with this approach which is just token usage and costs. Obviously, we're feeding way more information into our LLM which is going to slow them down a little bit. It is also going to cost us more especially if we're using OpenAI you're paying per token and if we feed too much information in there your LLM can actually the performance can degrade quite a bit especially when it's trying to follow instructions so there are always those sort of things to consider as well but overall if done well by not feeding too much into the context window this approach is very good and when we need other approaches or an external knowledge base we can look at rag with agents or rag with guardrails which is both of those are alternative approaches that have their own pros and cons but effectively you get the same outcome

References

- https://github.com/pinecone-io
- https://huggingface.co/datasets/jamescalam/llama-2-arxiv-papers-chunked/tree/main
- https://medium.com/@lokaregns/text-summarization-with-hugging-face-transformers-a-beginners-guide-9e6c319bb5ed
- https://python.langchain.com/docs/use_cases/question_answering/
- https://openai.com/research/whisper