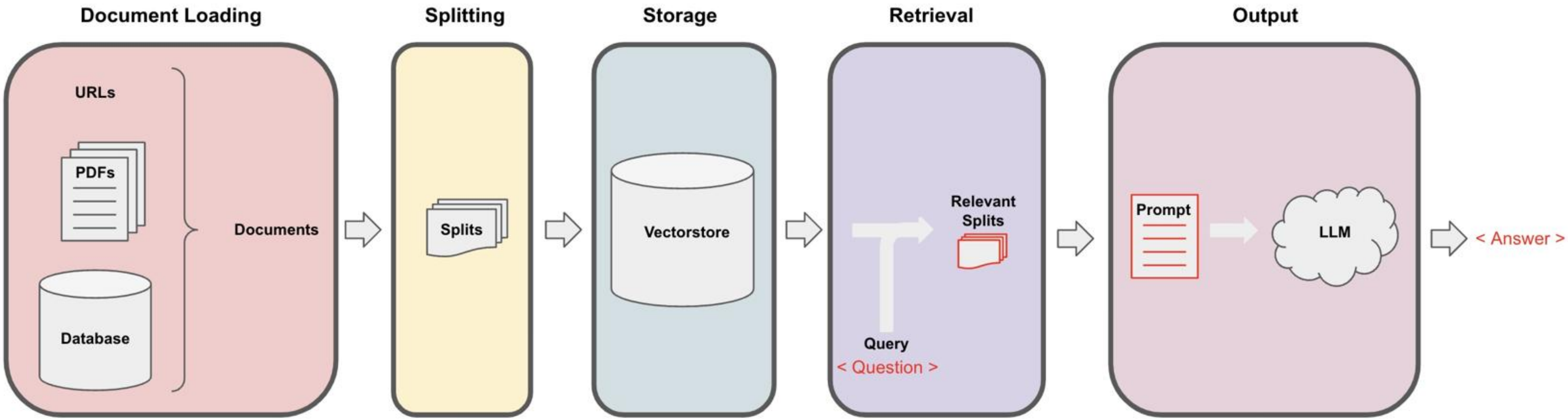
Advanced RAG Techniques

(Some topics are optional content for Agentic AI Course)

1st Dec 2025

Recap: Langchain Pipeline for Question Answering



- The above subsystems can be viewed as independent modules that obey well defined interfaces
- This means that one can replace a module with another technique, so long as interfaces are respected

Quiz

- How to choose the chunking technique?
- What happens if the chunk size is too large? Too small?
- How do we know which chunk size is optimal?
- What happens if the embedding is not accurate?

RAG Limitations

- Retrieval Challenges:
  - Precision and recall issues, leading to the selection of misaligned or irrelevant chunks, and the missing of crucial information.
- Generation Difficulties:
  - The model may hallucinate, where it produces content not supported by the retrieved context. Issues of irrelevance, toxicity, or bias in the outputs, detracting from the quality and reliability of the responses.
- Augmentation Hurdles:
  - Redundancy when similar information is retrieved from multiple sources, leading to repetitive responses.
- Generation models might overly rely on augmented information, leading to outputs that simply echo retrieved content without adding insightful or synthesized information.

Part 2: Advanced Techniques

Approach to enhancing RAG

- RAG is a multi step workflow involving several subsystems like vector store

- To improve RAG performance, each of these steps can be improved

RAG: Seven Failure Points

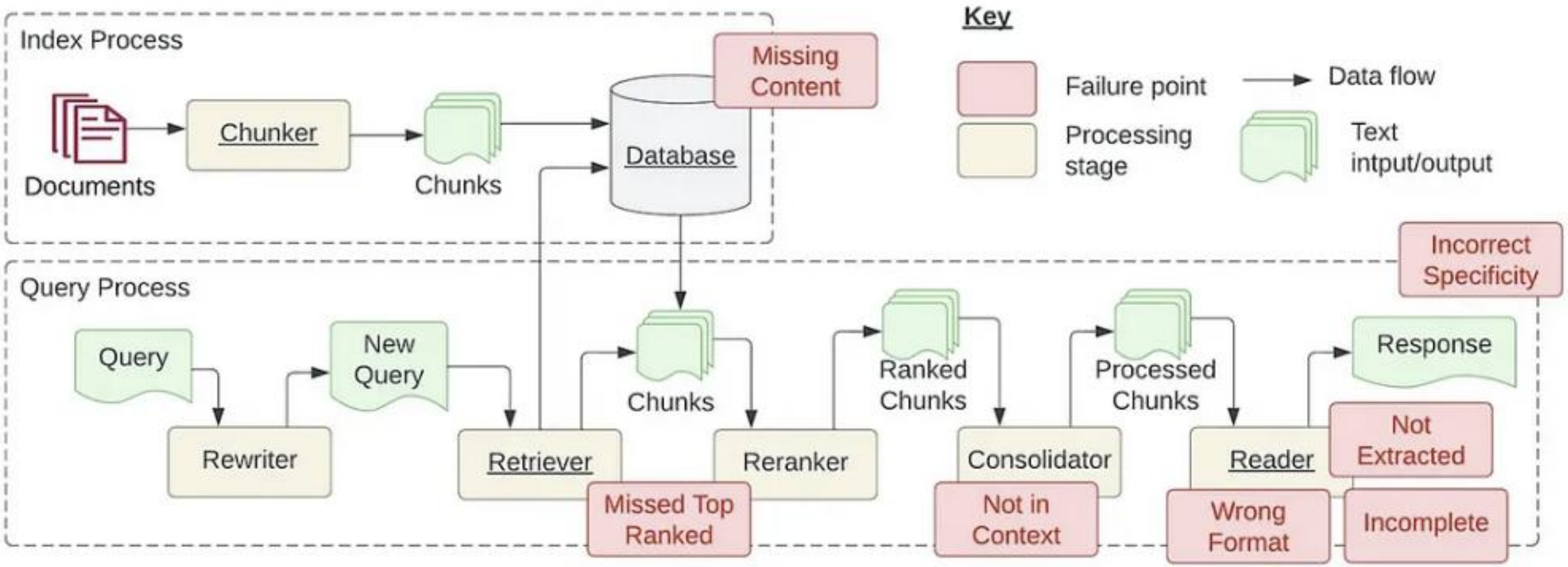


Figure 1: Indexing and Query processes required for creating a Retrieval Augmented Generation (RAG) system. The indexing process is typically done at development time and queries at runtime. Failure points identified in this study are shown in red boxes. All required stages are underlined. Figure expanded from [19].

Image source: Seven Failure Points When Engineering a Retrieval Augmented Generation System

Failure Points

5 FAILURE POINTS OF RAG SYSTEMS

From the case studies we identified a set of failure points presented below. The following section addresses the research question *What are the failure points that occur when engineering a RAG system?*

FP1 Missing Content The first fail case is when asking a question that cannot be answered from the available documents. In the happy case the RAG system will respond with something like “Sorry, I don’t know". However, for questions that are related to the content but don’t have answers the system could be fooled into giving a response.

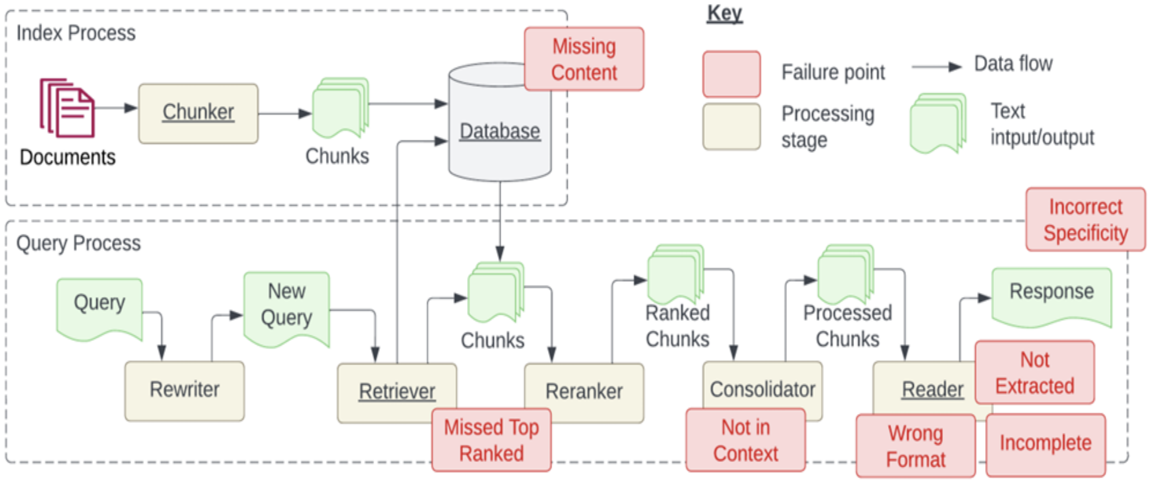
FP2 Missed the Top Ranked Documents The answer to the question is in the document but did not rank highly enough to be returned to the user. In theory, all documents are ranked and used in the next steps. However, in practice the top K documents are returned where K is a value selected based on performance.

FP3 Not in Context - Consolidation strategy Limitations Documents with the answer were retrieved from the database but did not make it into the context for generating an answer. This occurs when many documents are returned from the database and a consolidation process takes place to retrieve the answer.

FP4 Not Extracted Here the answer is present in the context, but the large language model failed to extract out the correct answer. Typically, this occurs when there is too much noise or contradicting information in the context.

FP5 Wrong Format The question involved extracting information in a certain format such as a table or list and the large language model ignored the instruction.

FP6 Incorrect Specificity The answer is returned in the response but is not specific enough or is too specific to address the user’s need. This occurs when the RAG system designers have a desired outcome for a given question such as teachers for students. In this case, specific educational content should be provided with answers not just the answer. Incorrect specificity also occurs when users are not sure how to ask a question and are too general.



FP7 Incomplete Incomplete answers are not incorrect but miss some of the information even though that information was in the context and available for extraction. An example question such as “What are the key points covered in documents A, B and C?” A better approach is to ask these questions separately.

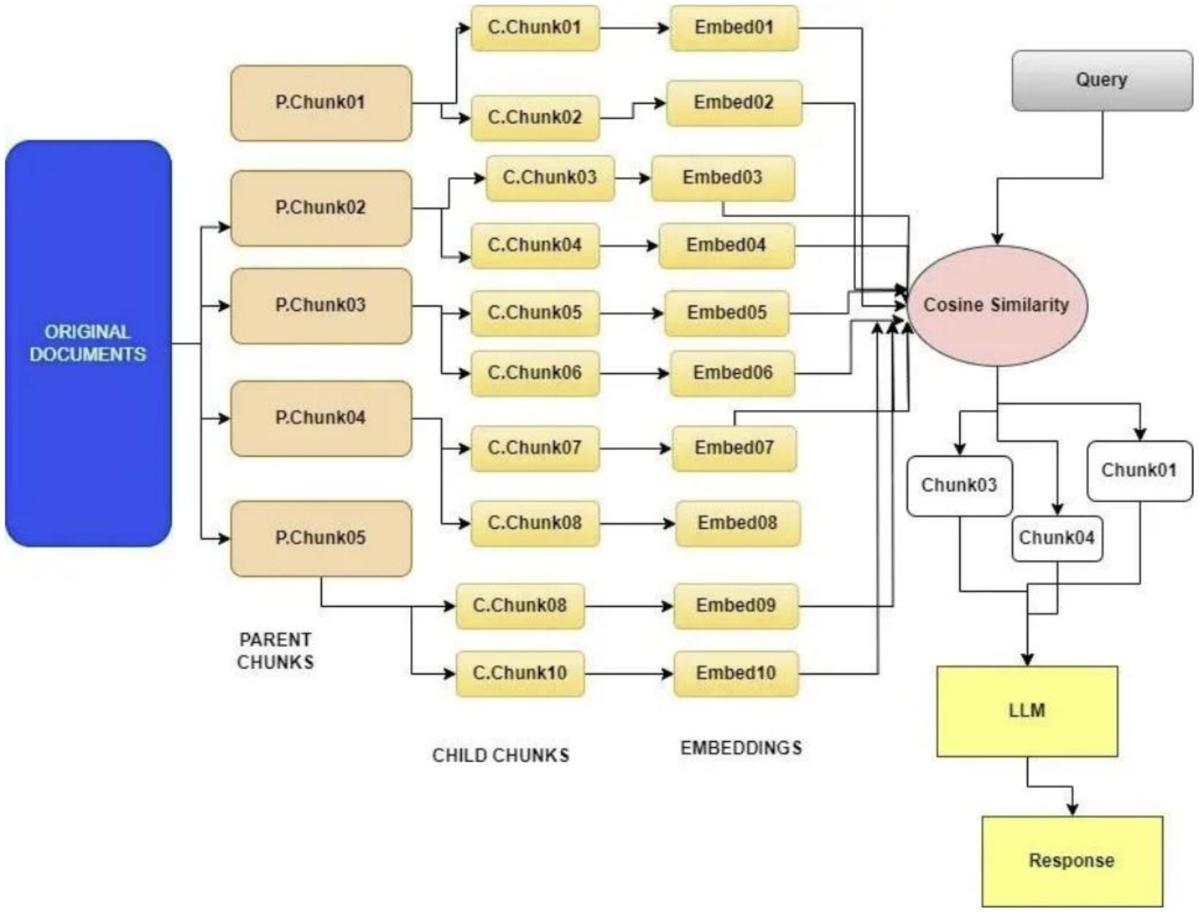
Metadata to enhance RAG quality

- Metadata, such as concept and level tags, can improve the quality of indexed data.
  - In our example, we can tag all documents pertaining to the latest IPL as “IPL 2024”
  - Notion is similar to using hashtags in tweets
- If you search for items and recency is a criterion, you can sort over a date metadata
- If you search over scientific papers and you know in advance that the information you’re looking for is always located in a specific section, say the experiment section for example, you can add the article section as metadata for each chunk and filter on it to match experiments only
- Metadata is useful because it brings an additional layer of structured search on top vector search.

Chunk Size optimization

- A small chunk gives more accurate matching when cosine similarity is used while it can miss some of the relevant context
- A large chunk can encompass a larger information but can also include irrelevant data for the query posed
- A large chunk may result in a slower response as more tokens need to be processed
- Based on downstream task optimal length of the chunk need to determine and how much overlap you want to have for each chunk.
  - High-level tasks like **summarization** requires bigger chunk size and low-level tasks like **coding** requires smaller chunks

Parent Document Retrieval




```
class SQLRuntime(object):
    def __init__(self, dbname=None):
        if dbname is None:
            dbname = my_db
        conn = sqlite3.connect(dbname)  # creating a connection
        self.cursor = conn.cursor()  # we need the cursor to execute statement
        return

    1 usage
    def list_tables(self):
        result = self.cursor.execute("SELECT name FROM sqlite_master WHERE type='table';").fetchall()
        table_names = sorted(list(zip(*result))[0])
        return table_names

    1 usage
    def get_schema_for_table(self, table_name):
        result = self.cursor.execute("PRAGMA table_info('%s')" % table_name).fetchall()
        column_names = list(zip(*result))[1]
        return column_names

    def get_schemas(self):
        schemas = {}
        table_names = self.list_tables()
        for name in table_names:
            fields = self.get_schema_for_table(name)  # fields of the table name
            schemas[name] = fields
        return schemas

    def execute(self, statement):
        code = 0
        msg = {
            "text": "SUCCESS",
            "reason": None,
            "traceback": None,
        }
        data = None
```

Parent Document Retrieval

- The ParentDocumentRetriever strikes that balance by splitting and storing small chunks of data.
- During retrieval, it first fetches the small chunks but then looks up the parent ids for those chunks and returns those larger documents.
- The term “parent document” refers to the document that a small chunk originated from.
- This can either be the whole raw document OR a larger chunk.

```python
loaders = [
    TextLoader("../../paul_graham_essay.txt"),
    TextLoader("../../state_of_the_union.txt"),
]
docs = []
for loader in loaders:
    docs.extend(loader.load())
```

```python
# This text splitter is used to create the parent documents
parent_splitter = RecursiveCharacterTextSplitter(chunk_size=2000)
# This text splitter is used to create the child documents
# It should create documents smaller than the parent
child_splitter = RecursiveCharacterTextSplitter(chunk_size=400)
# The vectorstore to use to index the child chunks
vectorstore = Chroma(
    collection_name="split_parents", embedding_function=OpenAIEmbeddings()
)
# The storage layer for the parent documents
store = InMemoryStore()
```

```python
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore,
    docstore=store,
    child_splitter=child_splitter,
    parent_splitter=parent_splitter,
)
```

```python
retriever.add_documents(docs)
```

```python
sub_docs = vectorstore.similarity_search("justice breyer")
```

Query Rewriting

- Query rewriting aligns the semantics of a query and a document.
- An LLM is used to rephrase the user's query and give it another shot. It's important to note that two questions that might look the same to a human may not appear similar in the embedding space.
- The Multi-query Retrieval method utilizes LLMs to generate multiple queries from different perspectives for a given user input query, advantageous for addressing complex problems with multiple sub-problems.
- For each query, it retrieves a set of relevant documents and takes the unique union across all queries to get a larger set of potentially relevant documents.
- By generating multiple perspectives on the same question, the MultiQuery Retriever might be able to overcome some of the limitations of the distance-based retrieval and get a richer set of results.

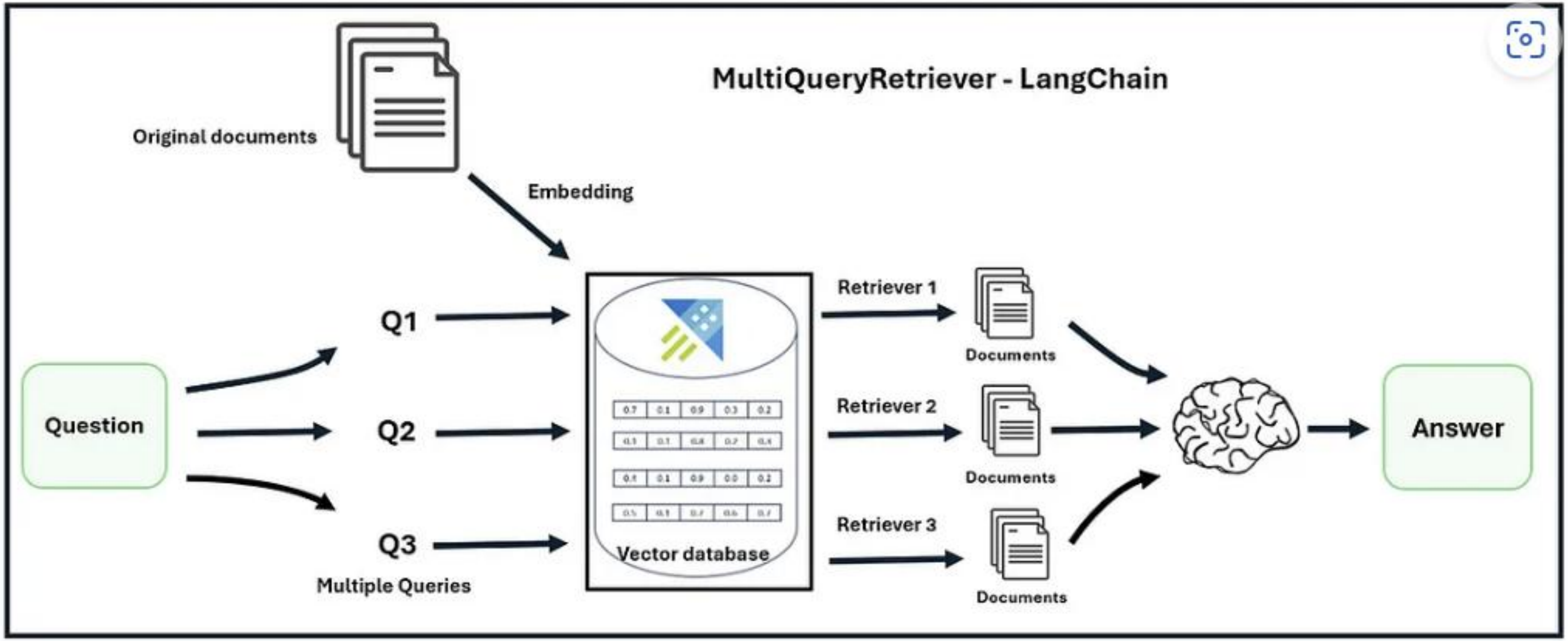
Ref: LangChain documentation [MultiQueryRetriever | 🦜️🔗 Langchain]

MultiQueryRetriever

Distance-based vector database retrieval embeds (represents) queries in high-dimensional space and finds similar embedded documents based on "distance". But, retrieval may produce different results with subtle changes in query wording or if the embeddings do not capture the semantics of the data well. Prompt engineering / tuning is sometimes done to manually address these problems, but can be tedious.

The `MultiQueryRetriever` automates the process of prompt tuning by using an LLM to generate multiple queries from different perspectives for a given user input query. For each query, it retrieves a set of relevant documents and takes the unique union across all queries to get a larger set of potentially relevant documents. By generating multiple perspectives on the same question, the `MultiQueryRetriever` might be able to overcome some of the limitations of the distance-based retrieval and get a richer set of results.

Multi query Retriever



MultiQuery Retriever

Example

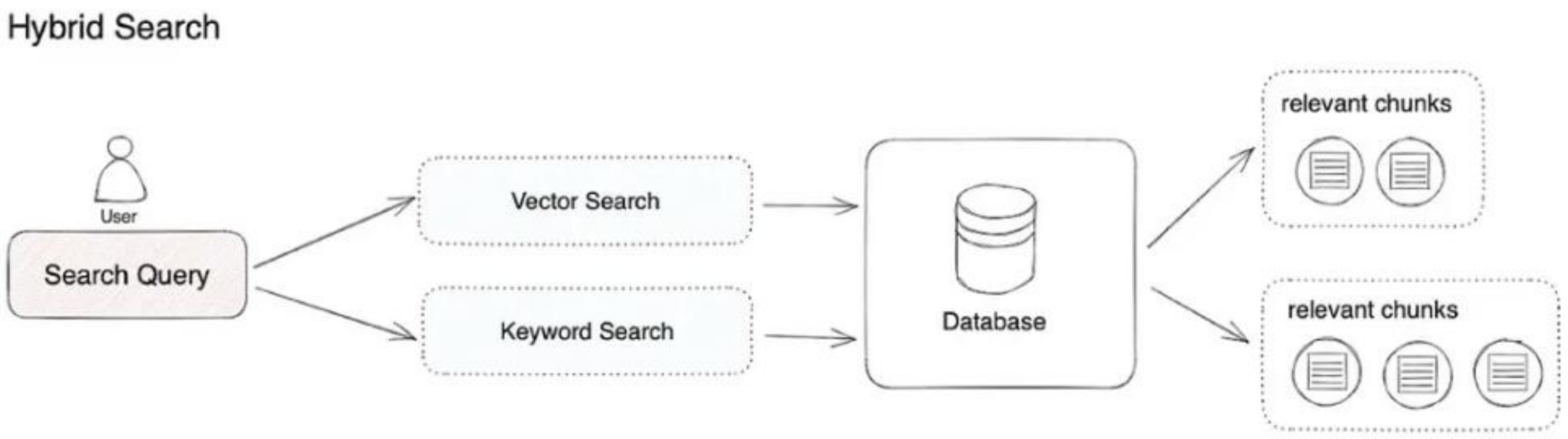
USER

You are an AI language model assistant. Your task is to generate five different versions of the given user question to retrieve relevant documents from a vector database. By generating multiple perspectives on the user question, your goal is to help the user overcome some of the limitations of the distance-based similarity search.
Provide these alternative questions separated by newlines.
Original question: I am interested to know more about quantum mechanics, such as, is it related to quantum computing or nano computing and also will I get a job if I study it?

AI

1. Could you please provide documents relating to the connections between quantum mechanics, quantum computing, and nano computing for my further exploration?
2. Can you help me find resources about the field of quantum mechanics, including its applications in quantum technology such as computing and nanotechnology?
3. I'd like to delve deeper into the concept of quantum mechanics – can you show me documents that discuss its relevance to advanced technologies like quantum computing and nano engineering, as well as potential career opportunities?
4. Could you help me locate materials on quantum mechanics and its implications for modern industries, particularly in the contexts of quantum computing and nanotechnology?
5. I'm intrigued by quantum mechanics and its role in modern technology – can you find documents that explain how it relates to both quantum computing and nano computing, as well as potential careers for those studying this field?

Hybrid Search Retrievers



Post Retrieval Optimizations: Re Ranking

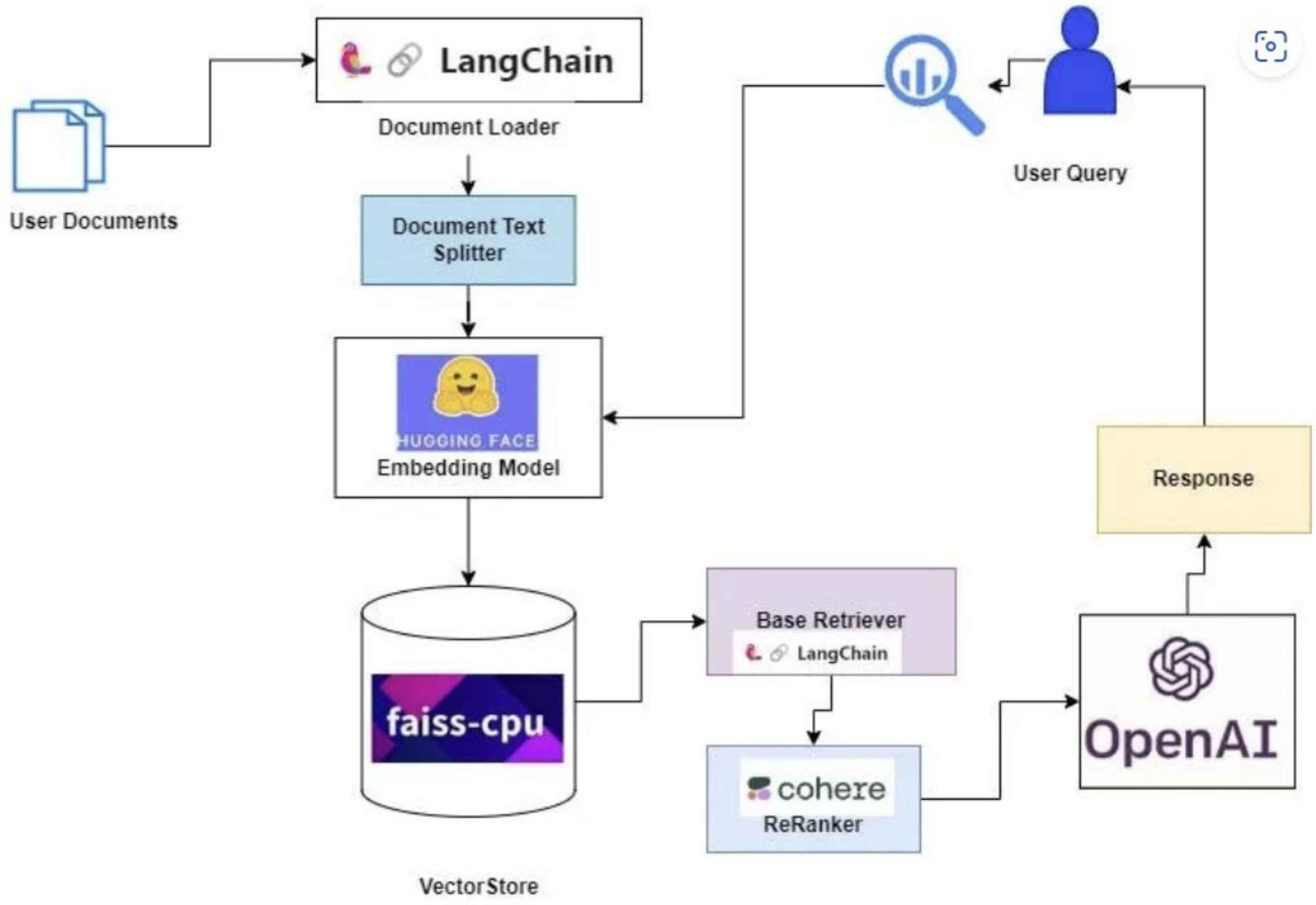
- Reranking retrieval results before sending them to the LLM has significantly improved RAG performance.

- **A high score in vector similarity search does not mean that it will always have the highest relevance.**

- The core concept involves re-arranging document records to prioritize the most relevant items at the top, thereby limiting the total number of documents. This not only resolves the challenge of context window expansion during retrieval but also enhances retrieval efficiency and responsiveness.

- Increase the similarity_top_k in the query engine to retrieve more context passages, which can be reduced to top_n after reranking.

Re-Ranking



Re-Ranking the retrieved document using Cohere Reranker

Chat History Basic - Demo

- See: http://localhost:8889/notebooks/basic_concepts_chat_history.ipynb

Chat History

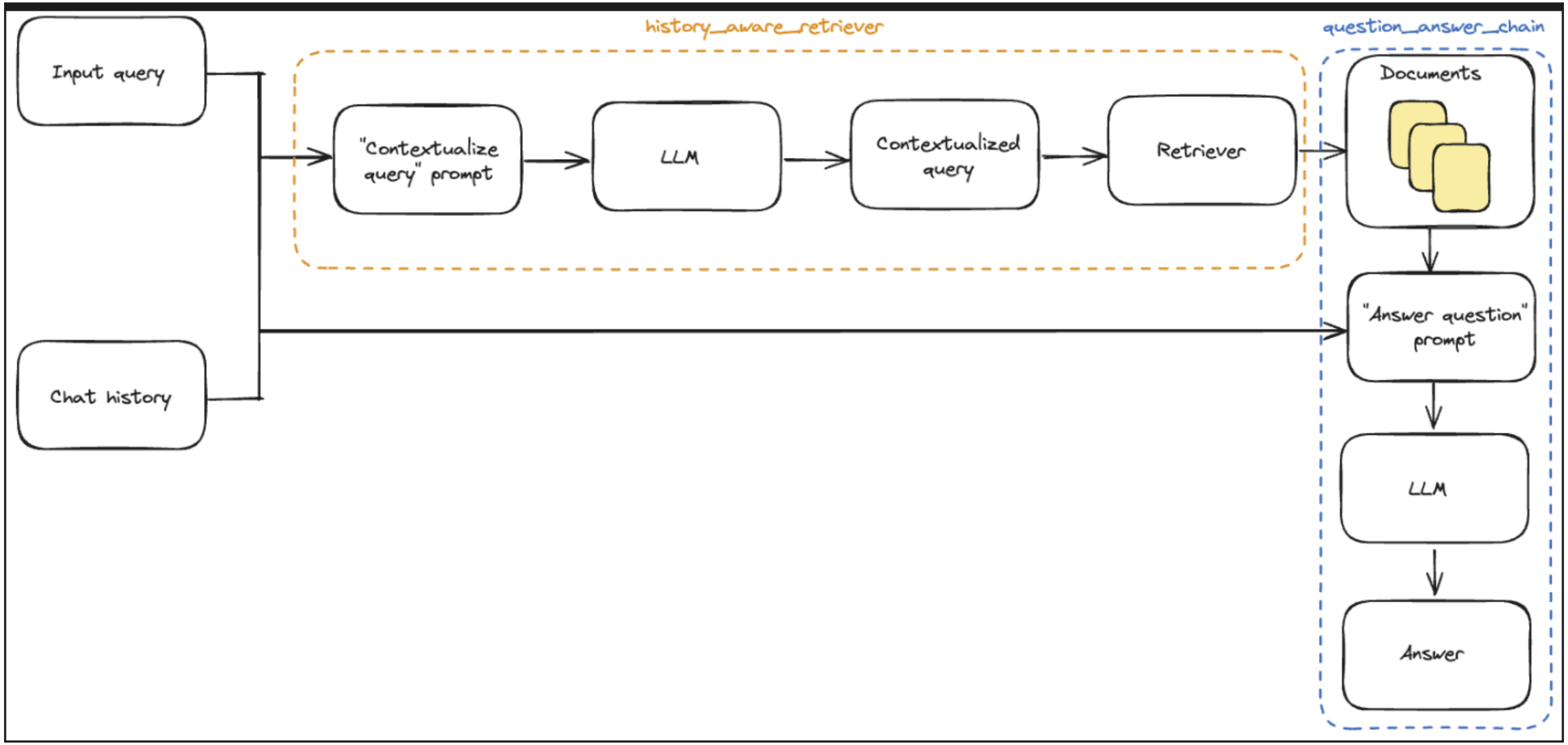
- In a chat based RAG application, often it is useful to support chat history

- For example:
  - User: What is the highest score made by Rohit Sharma in T20I matches?
  - Assistant: 121
  - User: Against whom?
  - Assistant: ….

- One way to implement this is by concatenating the current query with past context, but this is inefficient as it takes lot of context.

- It is a better idea to use an LLM to compress the context and create a stand alone query.

Supporting chat history



Condensing questions with another LLM

- This chain has two steps. First, it condenses the current question and the chat history into a standalone question.
- This is necessary to create a standalone vector to use for retrieval.
- After that, it does retrieval and then answers the question using retrieval augmented generation with a separate model.
- Part of the power of the declarative nature of LangChain is that you can easily use a separate language model for each call.
- This can be useful to use a cheaper and faster model for the simpler task of condensing the question, and then a more expensive model for answering the question.

Demo – Chat History Implementation

- See manage_conversation_history.ipynb

- chat_history_for_rag.ipynb

Implementation without condensation

```
def chat_history_with_memory():
    memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
    qa = ConversationalRetrievalChain.from_llm(llm, get_retriever().as_retriever(), memory=memory)
    query = "What is the total sales of Microsoft in the year 2017?"
    result = qa({"question": query})
    print(result)
    query = "How about the previous year?"
    result = qa({"question": query})
    print(result)
    return


def chat_history_explicit():
    qa = ConversationalRetrievalChain.from_llm(llm, get_retriever().as_retriever())
    chat_history = []
    query = "What does Torrent Pharma produce?"
    result = qa({"question": query, "chat_history": chat_history})
    print(result)

    chat_history = [(query, result["answer"])]
    query = "Does it produce anti diabetic drugs?"
    result = qa({"question": query, "chat_history": chat_history})
    print(result)
```

Implementation with Condensation

```
def condense_question():
    qa = ConversationalRetrievalChain.from_llm(
        llm,
        get_retriever().as_retriever(),
        condense_question_llm=llm,  # can be another LLM specialized on summarization
    )

    chat_history = []
    query = "What does Torrent Pharma produce?"
    result = qa({"question": query, "chat_history": chat_history})
    print(result)

    chat_history = [(query, result["answer"])]
    query = "Given the kind of drugs produced, tell me what is their strategy and how it differs from Micro Labs"
    result = qa({"question": query, "chat_history": chat_history})
    print(result)
```

Exercise

- Use the pharma dataset, say, 1mg_Biocon and create a RAG
- Implement chat history
  - Implement history without condensation
  - Implement with condensation
- Refer: https://chat.langchain.com/

Using custom prompt for condensation

By default, ConversationalRetrievalQA uses CONDENSE_QUESTION_PROMPT to condense a question. Here is the implementation of this in the docs

```
from langchain.prompts.prompt import PromptTemplate

_template = """Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question, in its original language.

Chat History:
{chat_history}
Follow Up Input: {question}
Standalone question:"""

CONDENSE_QUESTION_PROMPT = PromptTemplate.from_template(_template)
```

Using custom prompt for condensation

But instead of this any custom template can be used to further augment information in the question or instruct the LLM to do something.

from langchain.prompts.prompt import PromptTemplate

custom_template = """Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question. At the end of standalone question add this 'Answer the question in German language.' If you do not know the answer reply with 'I am sorry'.

Chat History:

{chat_history}

Follow Up Input: {question}

Standalone question:"""

CUSTOM_QUESTION_PROMPT = PromptTemplate.from_template(custom_template)

Quiz

- Refer the demo of Hands On exercise

- How would you improve its performance using Advanced RAG techniques?

Hands On Exercise

- Refer to the slide deck on Hands On Exercise

Summary

- RAG system involves:
  - Chunking the text and generating embedding for the chunks
  - Retrieving the chunks by semantic similarity search
  - Generate response based on the text of the top_k chunks

- Each of the above steps can lead to inaccuracies, implying that the data presented to the LLM is not entirely optimal
- Output generation is constrained by quality of LLM and context/sequence length restrictions
- Are the traditional search engines better than RAG?