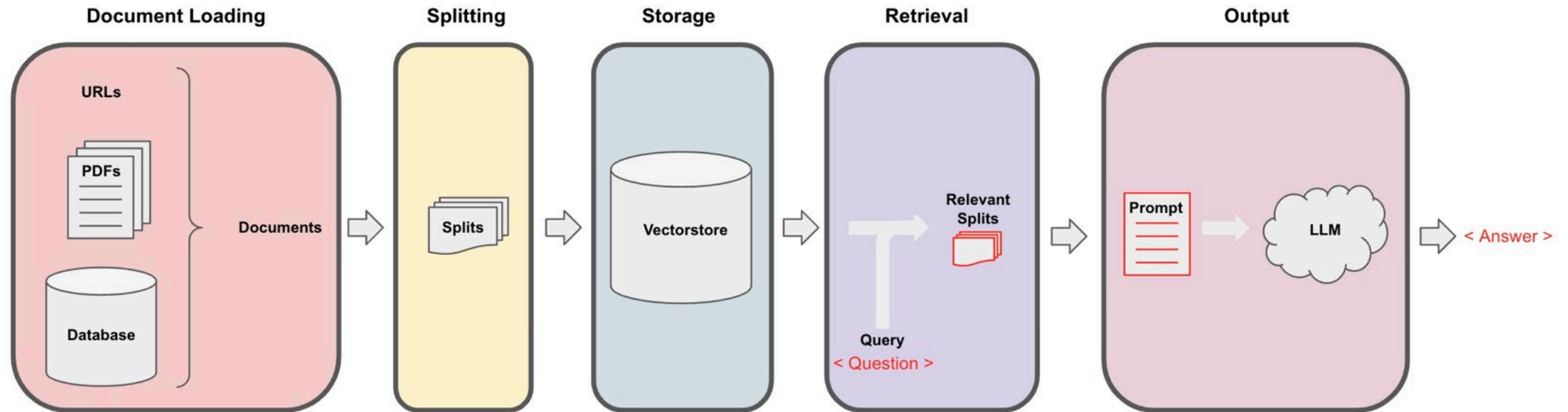


RAG Techniques – Part 2

Palacode Narayana Iyer Anantharaman

13th Sep 2024

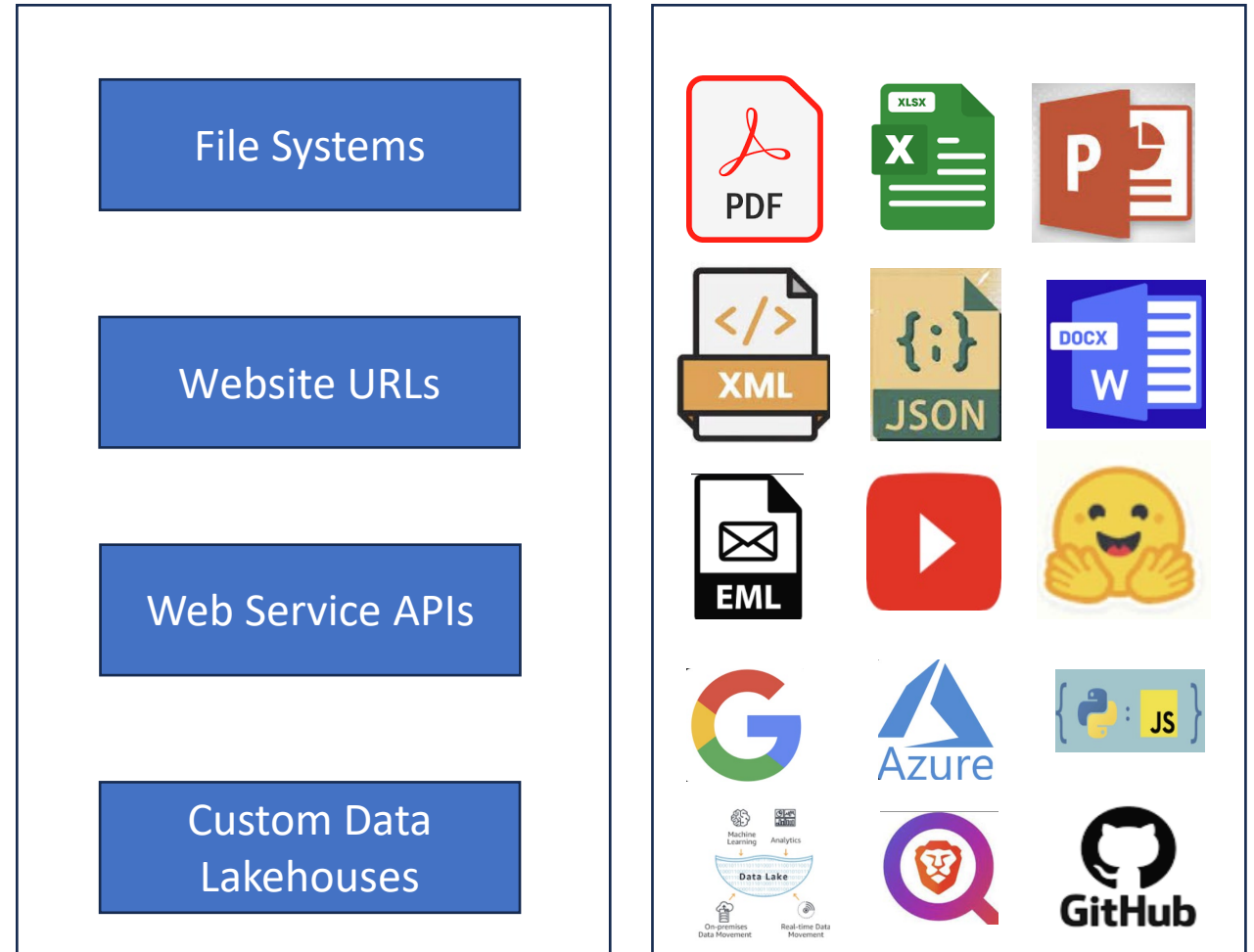
Langchain Pipeline for Question Answering



- 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

RAG Big Picture: Data Sources and Data formats

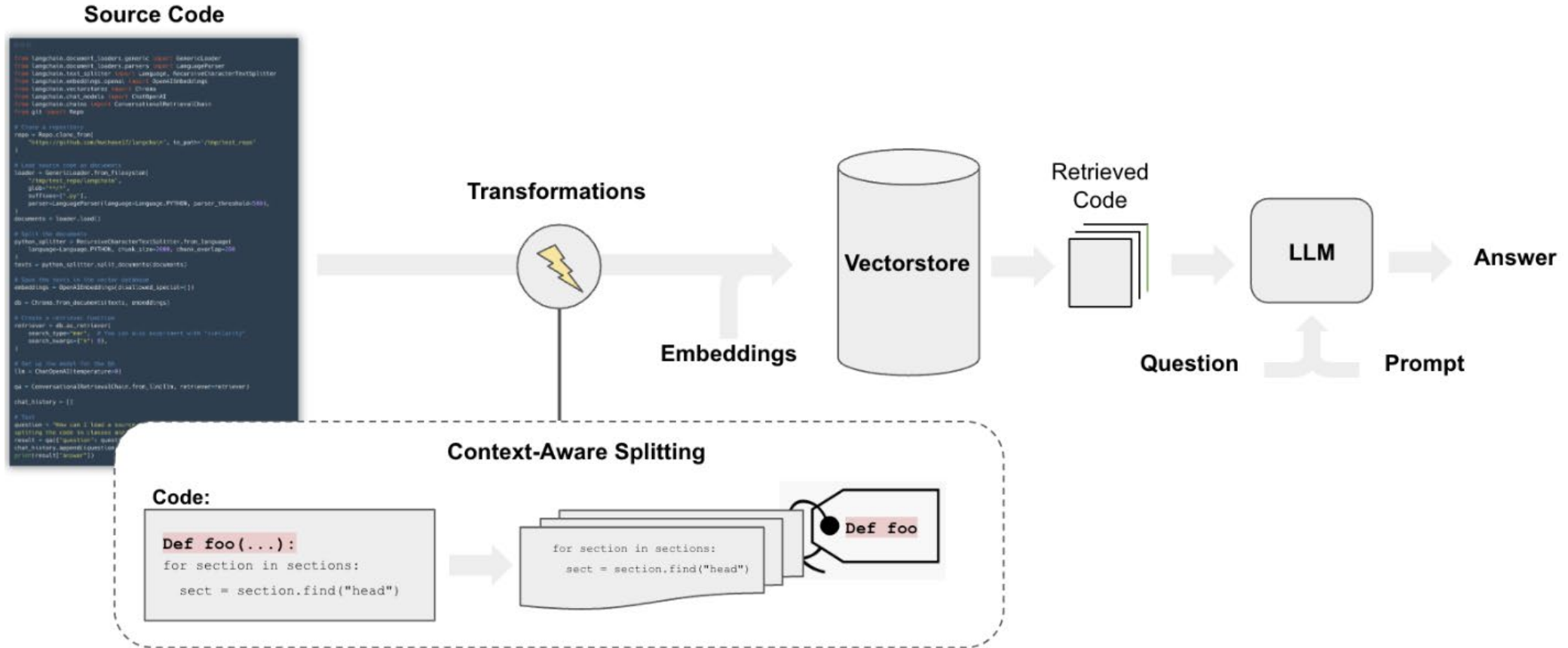
- We work with different data sources that arise from local file systems to web/cloud based resources.
- SaaS products, e-mails, Web Service APIs, Cloud databases etc are massive sources of data.
- Data can be unstructured (e.g. natural language texts in PDF documents) or structured (json data, excel tables storing numerical values), images, videos, audio, tables, source code, etc
- Data is represented in a variety of formats – pdfs, xlsx, pptx, py, json....
- Langchain doc loaders allow access to data using a common interface: Document class.



RAG Case Study #2: Chat with source code

- Ask the RAG to list specific functions (searching for certain functions)
- Get the function and ask the LLM to code review it
- Get some test cases from LLM for the specified classes
- Ask the LLM to performance optimize the given module
- Refactor some code using the LLM

RAG over code



Chat with your source code

- Splitting strategy that does a few things:
 - Keeps each top-level function and class in the code is loaded into separate documents.
 - Puts remaining into a separate document.
 - Retains metadata about where each split comes from
- Demo: Code splitter with LangChain
- Refer:
https://python.langchain.com/docs/integrations/document_loaders/source_code

Demo: RAG on Source code

- Develop a RAG based chatbot that chats with your source code.
- You can support only Python for this exercise.
- E.g. Try it on your own code in your product or clone any github python repository and chat with that code.
- See the sample code and the demo provided here: `ingest_code.py` and `coding_model.py`

Hands On Exercise: Cricket Analysis

- You are given a dataset pertaining to Rohit Sharma's international centuries. Download this from: [https://www.kaggle.com/datasets/mayankanand2701/rohit-sharma-international-centuries?select=Rohit Sharma Centuries.xlsx](https://www.kaggle.com/datasets/mayankanand2701/rohit-sharma-international-centuries?select=Rohit+Sharma+Centuries.xlsx)
- You are required to implement a RAG Q&A system ingesting this excel sheet.
- Use the excel document loader for your implementation: https://python.langchain.com/v0.2/docs/integrations/document_loaders/microsoft_excel/
- Write a suitable prompt to instruct the LLM (see my other examples)
- Generate a number of questions using ChatGPT or any other LLM
- Discuss RAG's limitations in the class.

Dataset Description

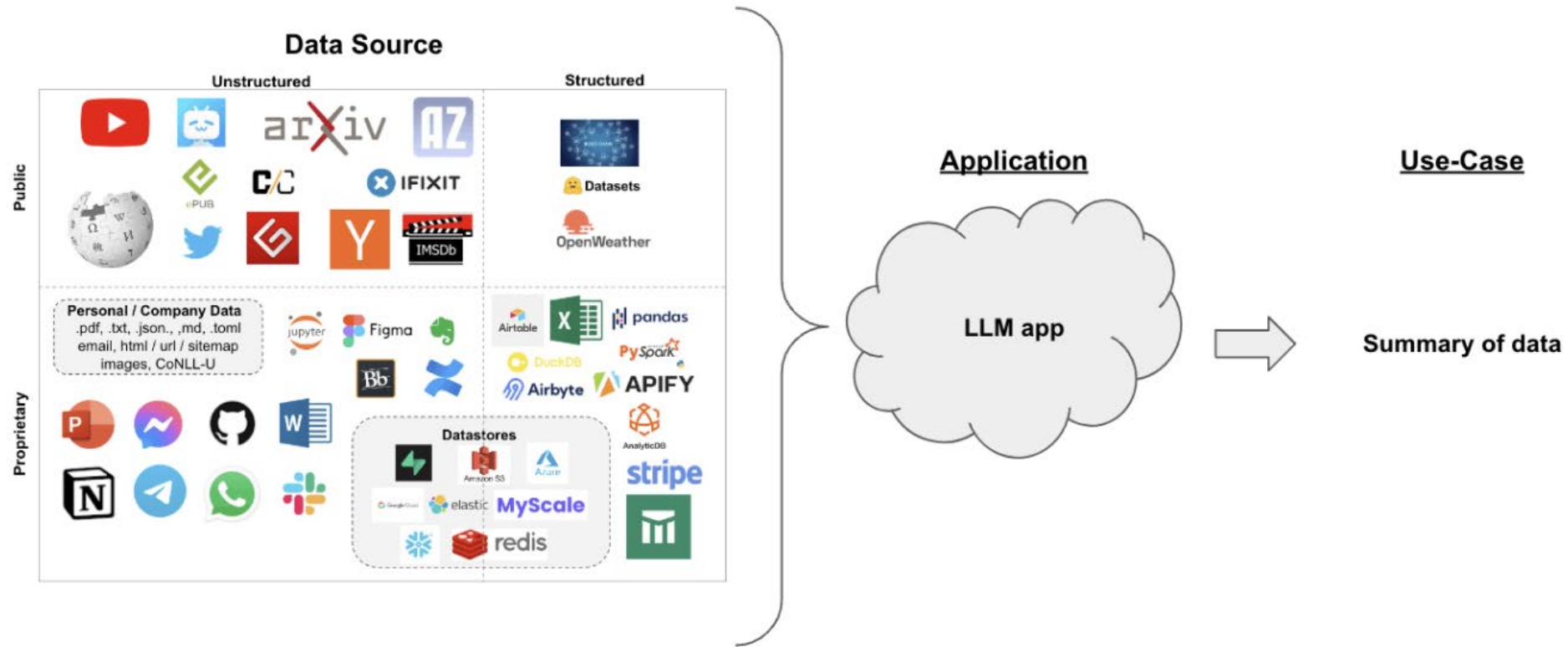
This **dataset** contains information about Rohit Sharma's centuries in international cricket. Each row represents a specific century scored by Rohit Sharma, detailing various aspects of the match and his performance. Dataset contains the following columns :

- **S.No.** : Serial number or index of the century in chronological order.
- **Date** : Date on which Rohit Sharma scored the century.
- **Score** : Runs scored by Rohit Sharma in that innings.
- **Strike Rate** : Strike rate at which Rohit Sharma scored (calculated as runs per 100 balls).
- **Type of Match** : Indicates the format of the match (Test, ODI, T20).
- **Position** : Batting position of Rohit Sharma in that innings.
- **Innings** : Innings number (1st or 2nd) played by Rohit Sharma.
- **Dismissed** : Indicates whether Rohit Sharma was dismissed (Yes) or remained not out (No).
- **Man of the Match** : Whether Rohit Sharma was awarded Man of the Match for his performance in that match (Yes or No).
- **Captain** : Indicates if Rohit Sharma was the captain of the team during that match (Yes or No).
- **Against** : Name of the opposition team against whom Rohit Sharma scored the century.
- **Venue** : Name of the cricket ground where the match was played.
- **H/A/N** : Indicates if the match was played at Home (India), Away (outside India), or Neutral venue.
- **Result** : Outcome of the match for Rohit Sharma's team (Won or Lost).

Hands On: Election 2019 data

- Use the elections 2019 dataset that we used in the previous session
- Ingest the excel using UnstructuredExcelLoader as in the cricket exercise
- Write the model code and set the prompts appropriately
- Implement the RAG system and test
- Determine if this approach is better than SQL approach in terms of accuracy.

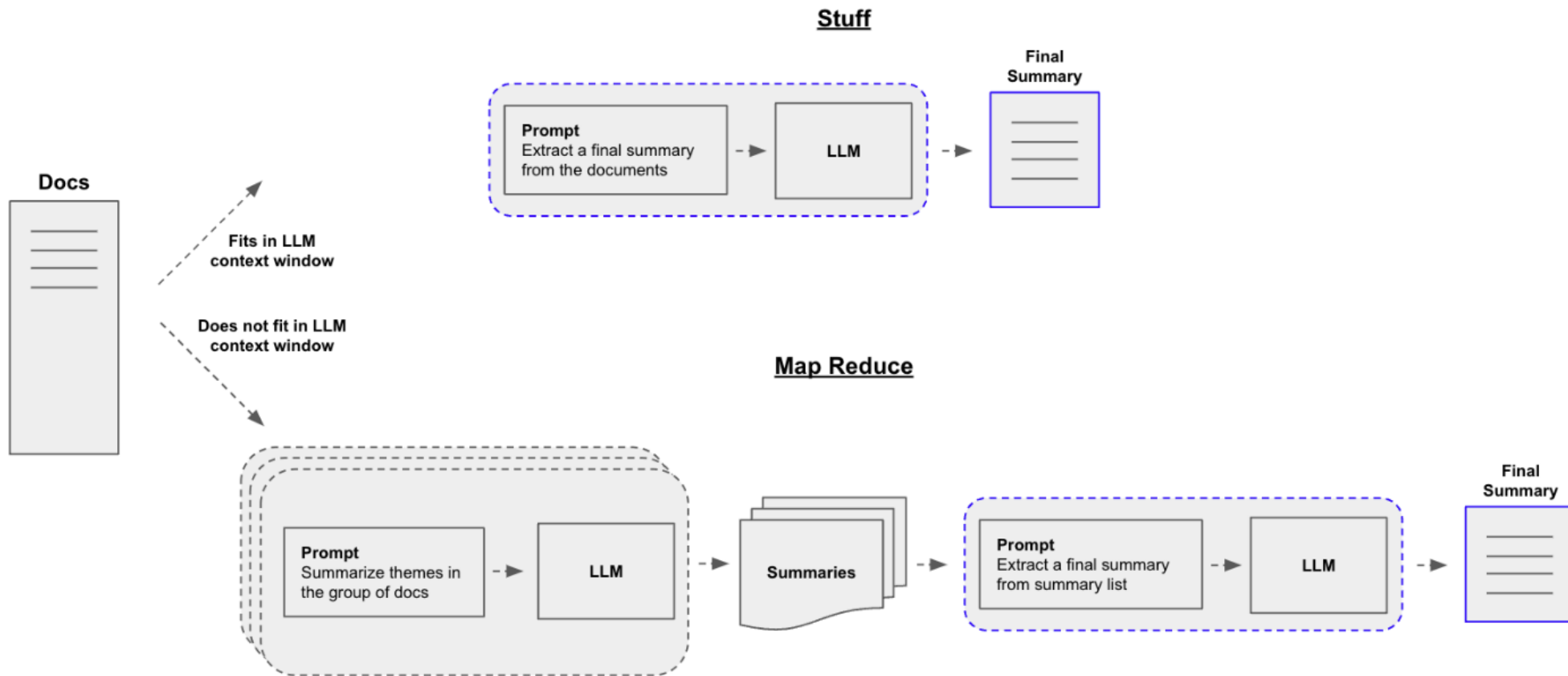
RAG Summarization



RAG Summarization

- RAG techniques ingest multiple documents from a variety of sources
- If the relevant retrieved chunks are too many, it may be a good idea to summarize their content before sending them to the LLM
- Three ways to summarize:
 - Stuff the content in the context window of LLM (if contents are less than context size)
 - Map-reduce: create low level summaries, summarize the summaries
 - Refine: updates a rolling summary by iterating over the documents in a sequence

Summarizing



Summarizer Demo

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.

RAG Paradigms

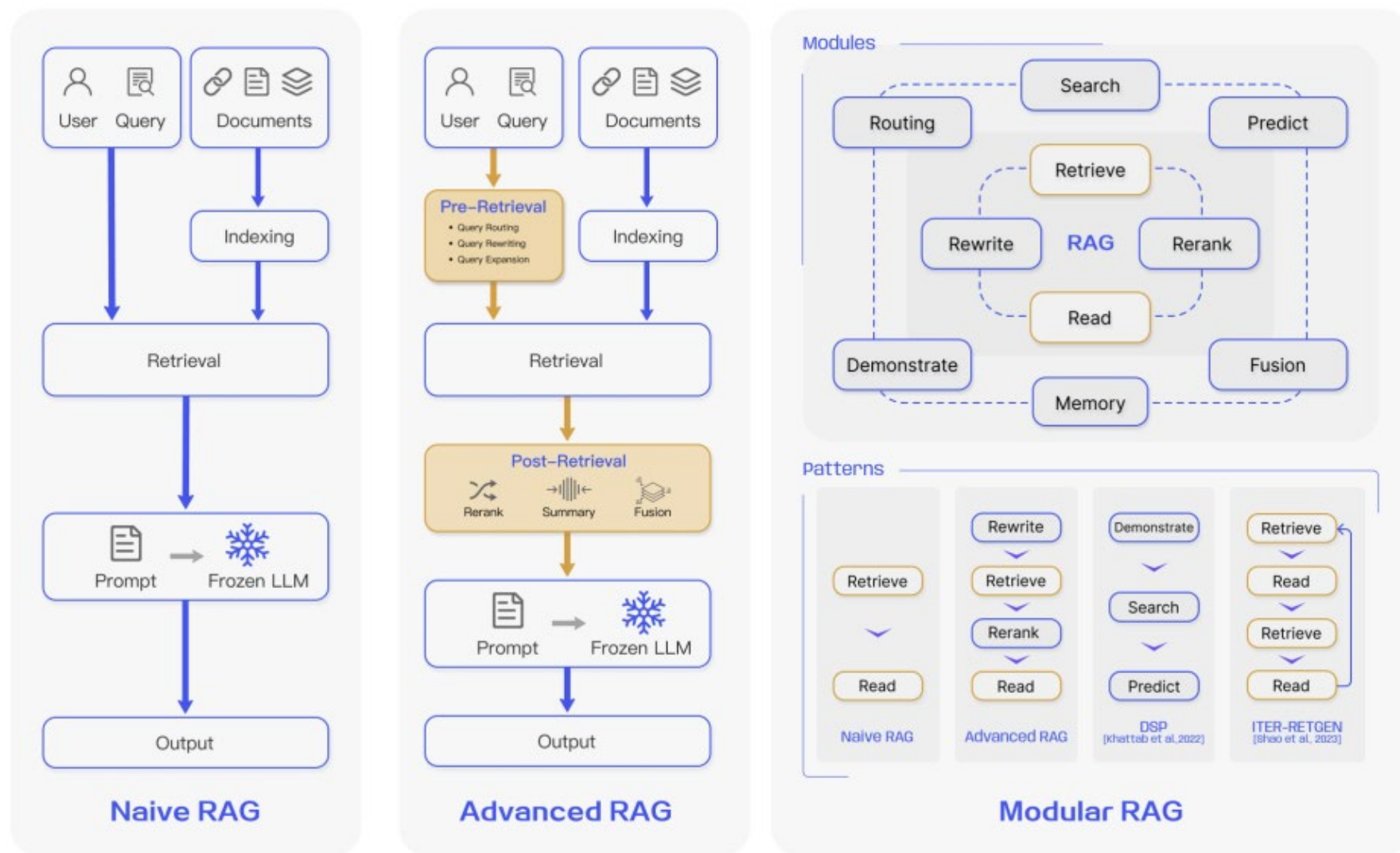


Fig Credits: RAG for LLM Models – A survey, Gao et al.

Retrieval Augmentation Processes

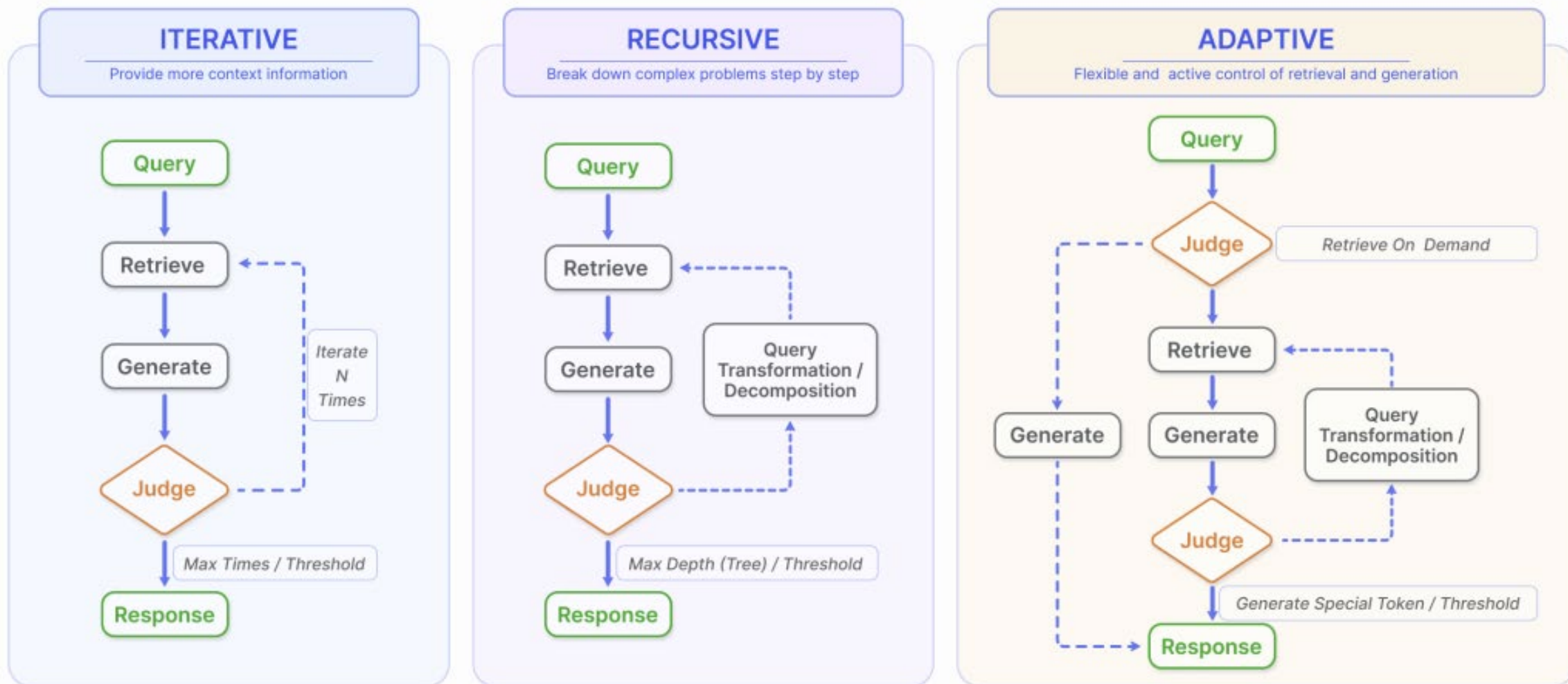
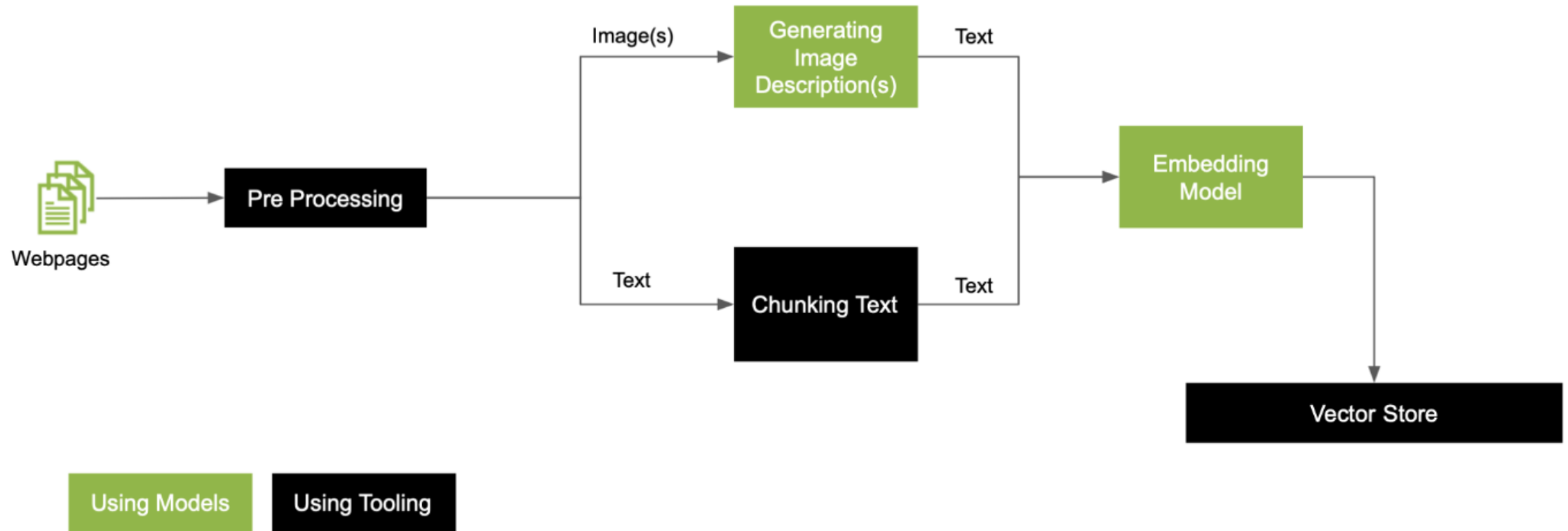


Fig Credits: RAG for LLM Models – A survey, Gao et al.

Multimodal RAG: Preprocessing



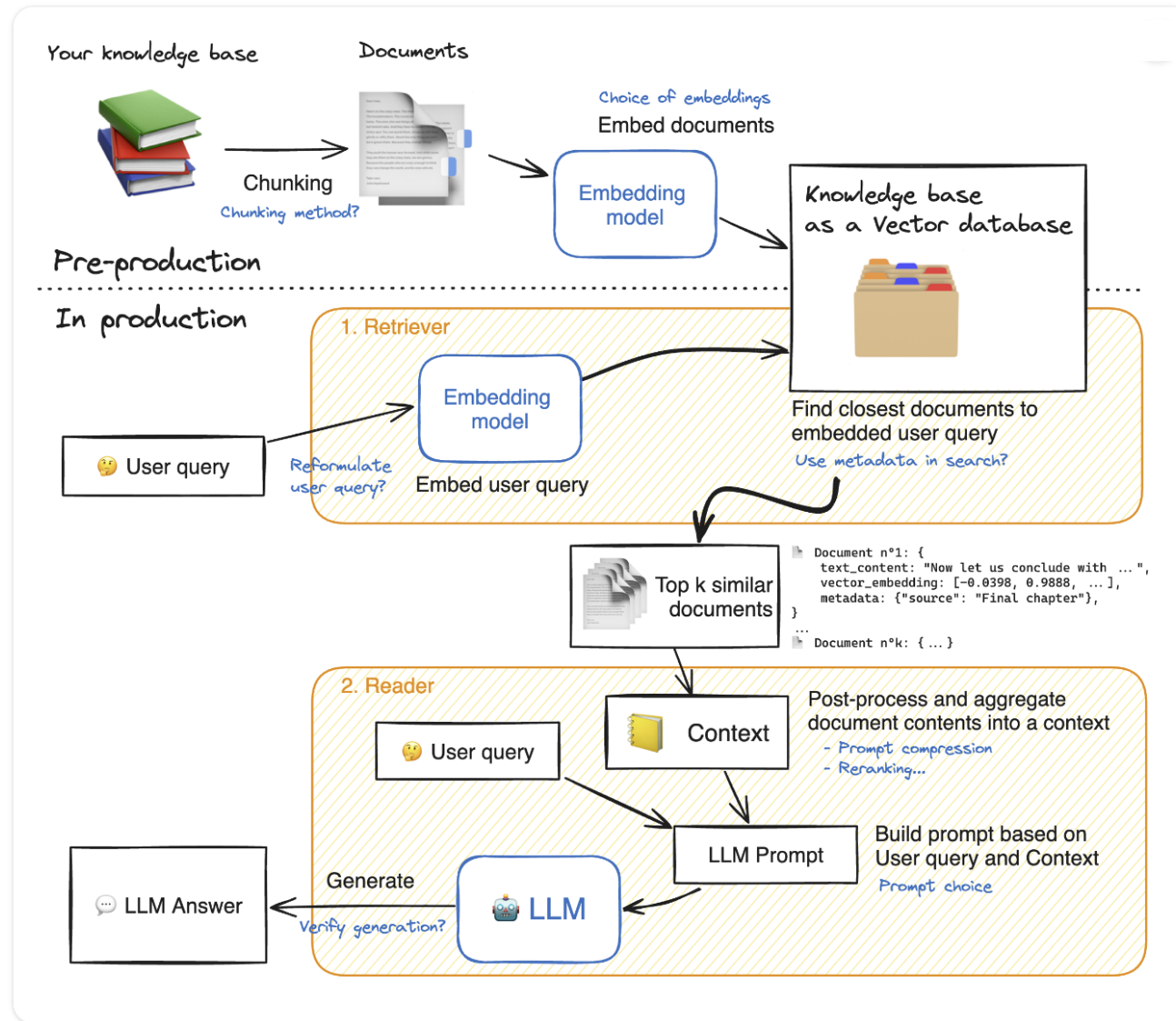
Part 2: Advanced Techniques

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

Enhancement Opportunities

Image from: [Advanced RAG on Hugging Face documentation using LangChain - Hugging Face Open-Source AI Cookbook](#)



RAG: Seven Failure Points

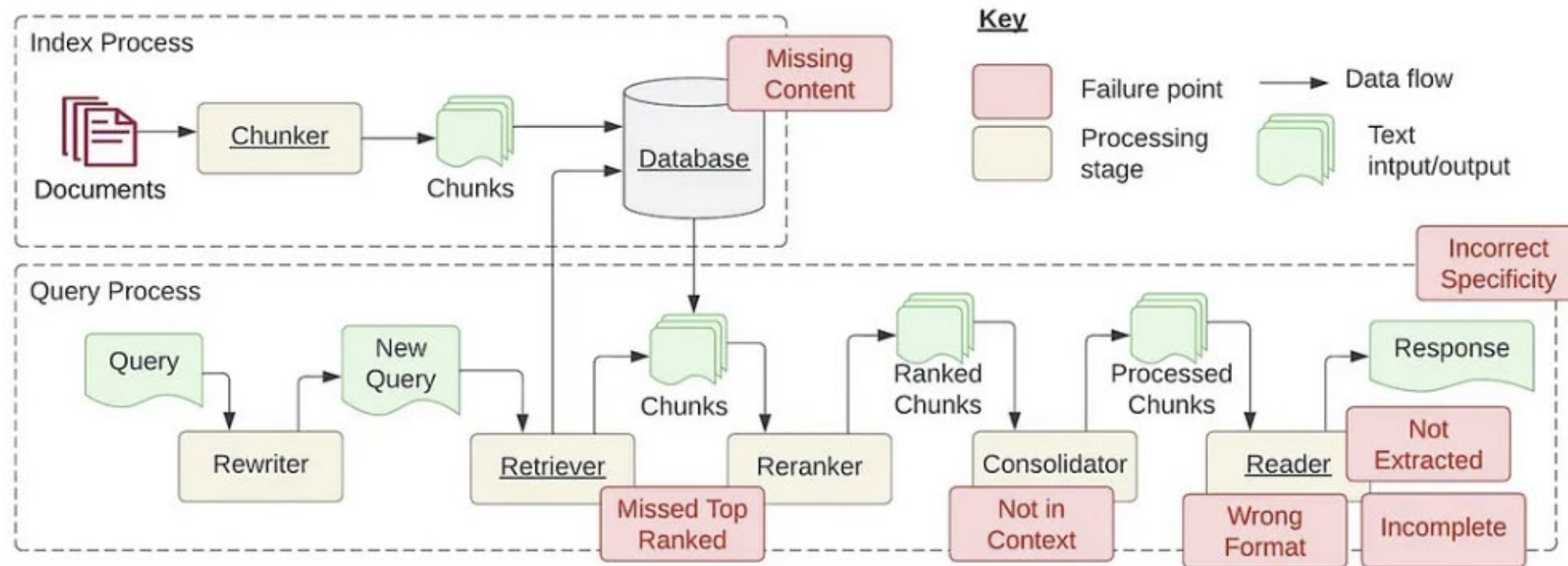


Figure 1: Indexing and Query processes required for creating a Retrieval Augmented Generation (RAG) system. The indexing process is typically done at development time and queries at runtime. Failure points identified in this study are shown in red boxes. All required stages are underlined. Figure expanded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

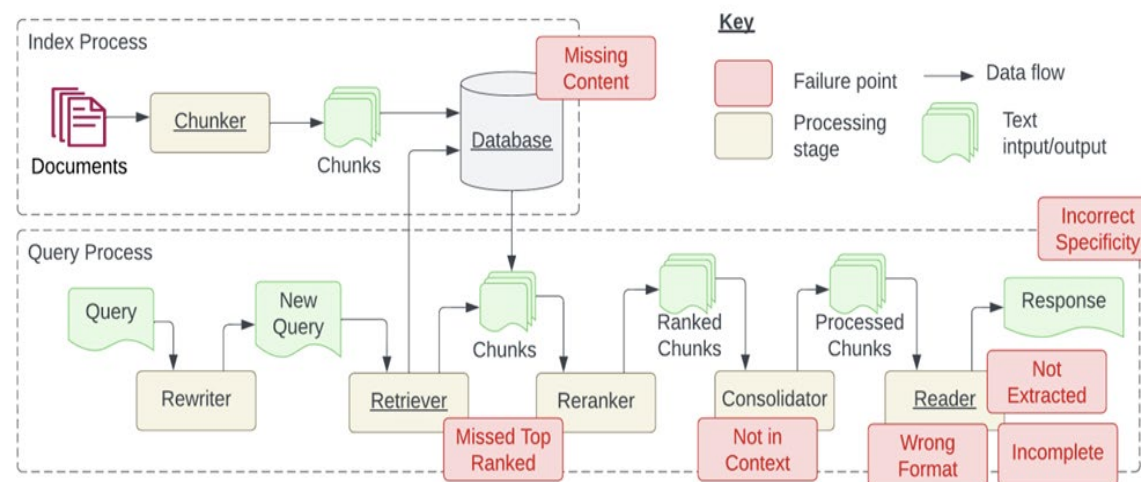
FP2 Missed the Top Ranked Documents The answer to the question is in the document but did not rank highly enough to be returned to the user. In theory, all documents are ranked and used in the next steps. However, in practice the top K documents are returned where K is a value selected based on performance.

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.



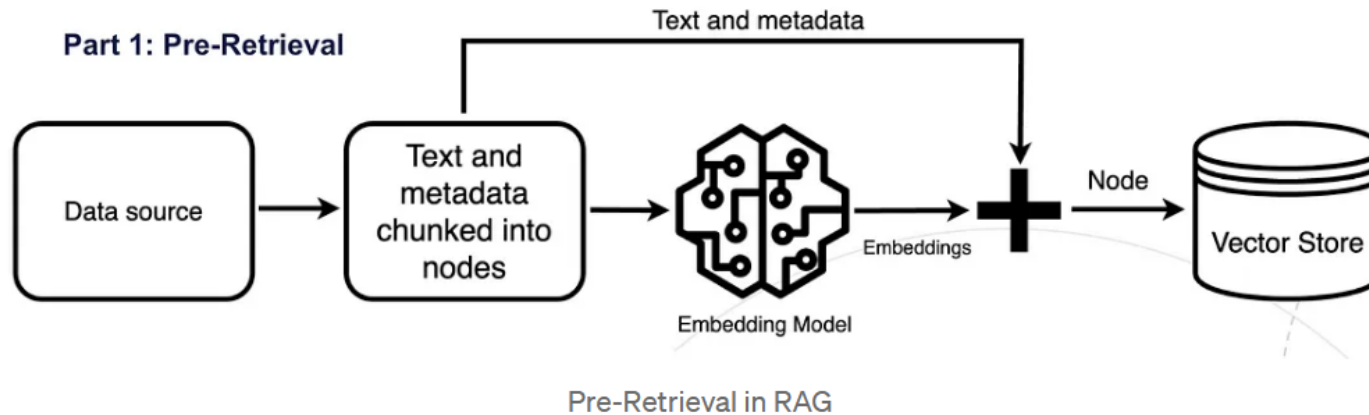
FP7 Incomplete Incomplete answers are not incorrect but miss some of the information even though that information was in the context and available for extraction. An example question such as "What are the key points covered in documents A, B and C?" A better approach is to ask these questions separately.

Advanced RAG: Approach

- The failure points can be associated with 3 broad steps of the RAG workflow:
 - Pre retrieval
 - Retrieval
 - Post Retrieval
- The main approach of advanced RAG techniques is to look at each stage and find ways to improve the quality of each of the components.
- In this section, we discuss some of the key techniques.

Quick Recap

RAG Stages: Pre retrieval

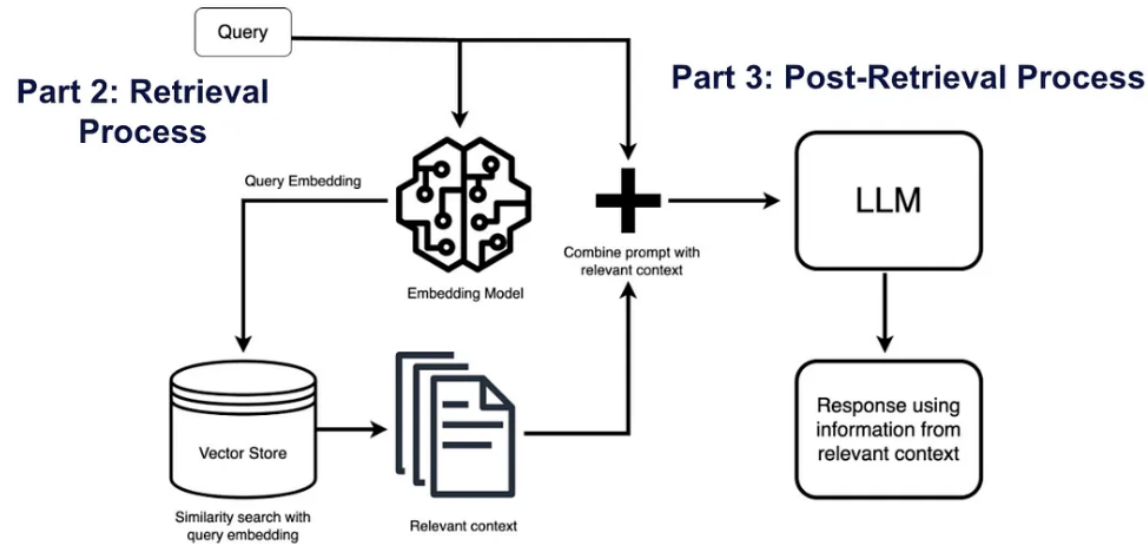


- Prepare external data, split into chunks and index the chunk data using embedding models
- Store the embeddings and chunks in the vector database
- This process creates a knowledge library that the LLM can understand.

RAG Stages: Retrieval

- In this step, convert the user query to a vector representation called embedding
- Obtain relevant chunks using cosine similarity from the vector database.
- Find highly-relevant document chunks (top-k) from vector store.

RAG Stages: Post retrieval



Retrieval and Post-Retrieval in RAG

- RAG augments the user input by adding the relevant retrieved data in context (query + context).
- Use prompt engineering techniques to communicate effectively with the LLM.
- Given this augmented prompt, the LLM generates an answer to the user query using given context.

Enhancing Data Quality

- Data cleaning on the candidate documents: Remove unnecessary or irrelevant content from the documents before ingesting
- Replace pronouns with names to improve possibility of providing required context.
- Example Text split as 2 chunks (colored Red and Black outlines):

Mayank Yadav's IPL debut has been remarkable, with the young pacer from Lucknow Super Giants securing consecutive Player of the Match awards. The 21-year-old's exceptional pace has captivated the cricketing world, as he surpassed his own record of 155.8 kph for the fastest ball of the season, unleashing a blistering 156.7 kph delivery against RCB.
- User Query: Who bowled the fastest delivery in the ongoing IPL?

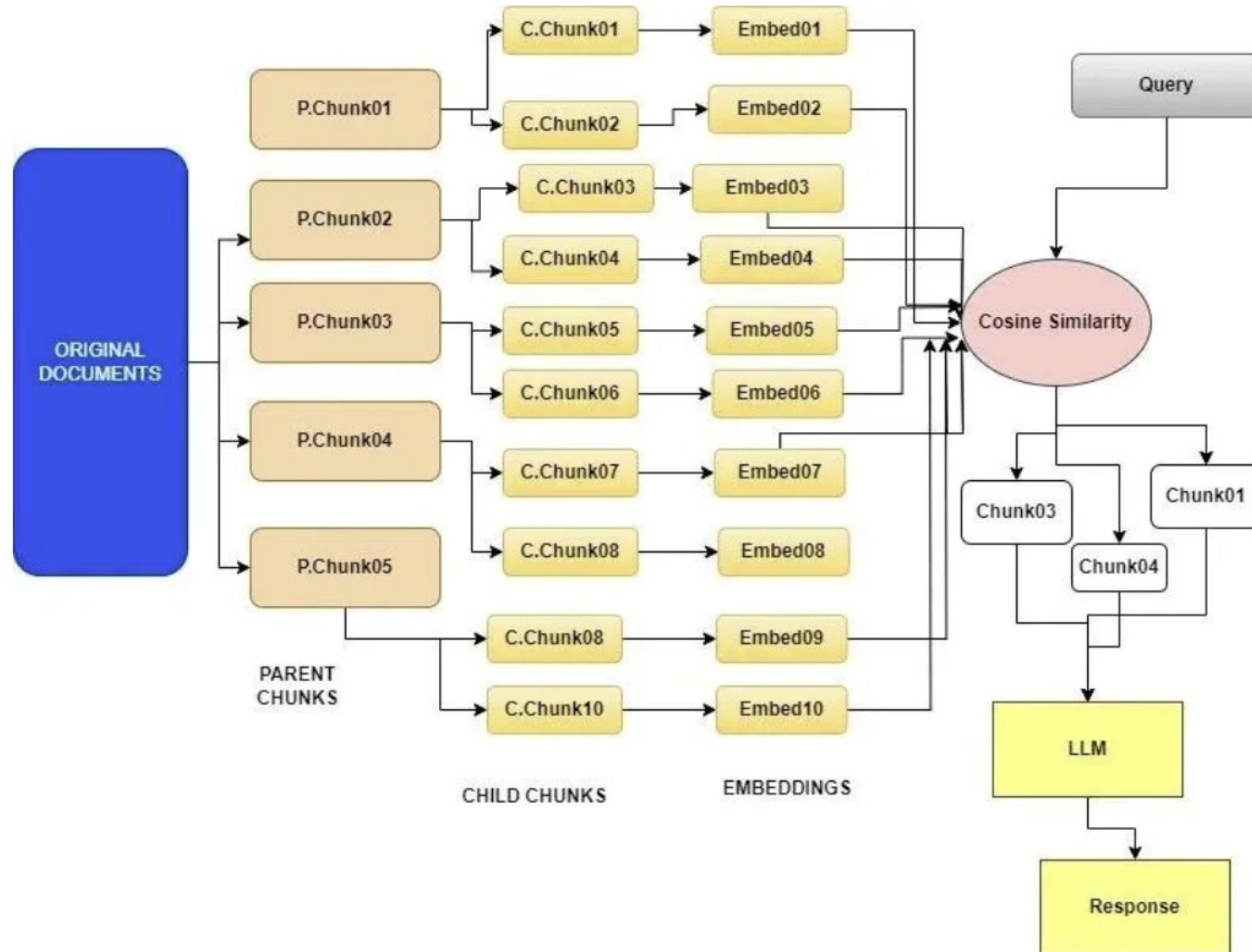
Metadata to enhance RAG quality

- 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 ongoing 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



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.

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

# 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()

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

retriever.add_documents(docs)
sub_docs = vectorstore.similarity_search("justice breyer")
```

Sentence Window Retrieval

[Ref: Advance RAG- Improve RAG performance | Medium](#)

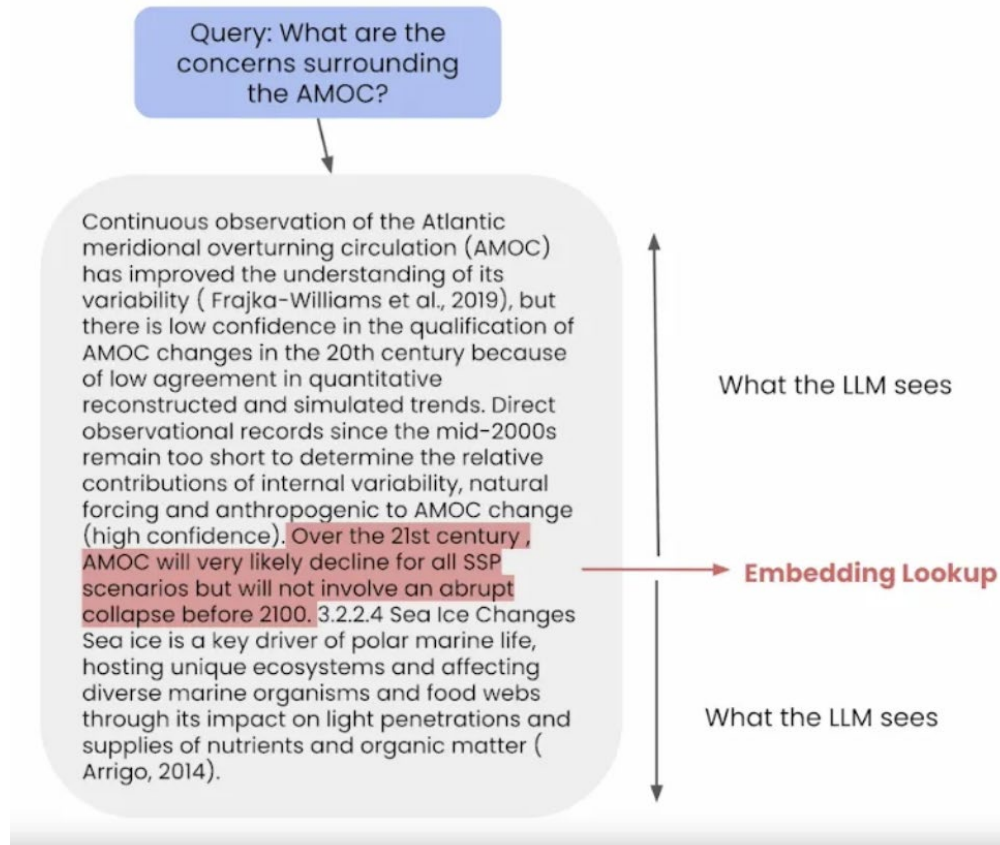
Sentence Window Retrieval

This chunking technique is very similar to above. The core idea behind Sentence Window Retrieval is to selectively fetch context from a custom knowledge base based on the query and then utilize a broader version of this context for more robust text generation.

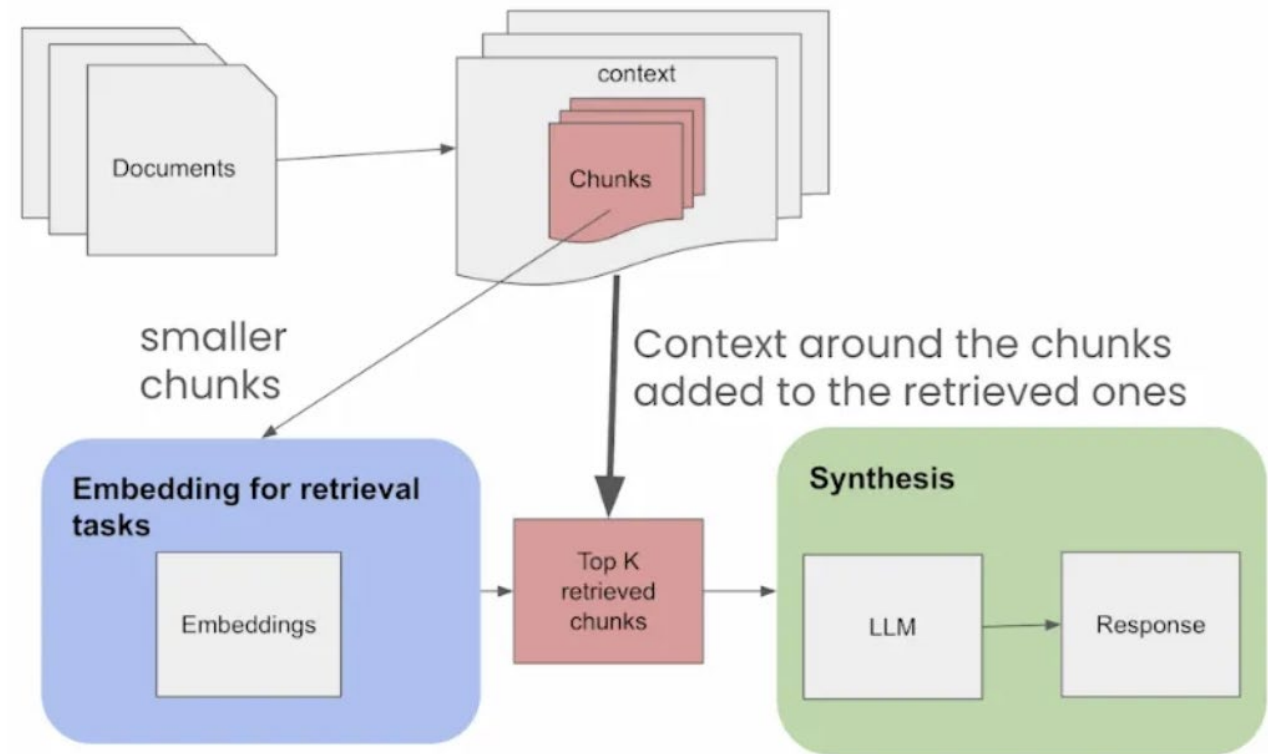
This process involves embedding a limited set of sentences for retrieval, with the additional context surrounding these sentences, referred to as “window context,” stored separately and linked to them. Once the top similar sentences are identified, this context is reintegrated just before these sentences are sent to the Large Language Model (LLM) for generation, thereby enriching overall contextual comprehension.

Sentence Window Retrieval

Sentence-window retrieval



In Sentence-window retrieval pipeline



Query Rewriting

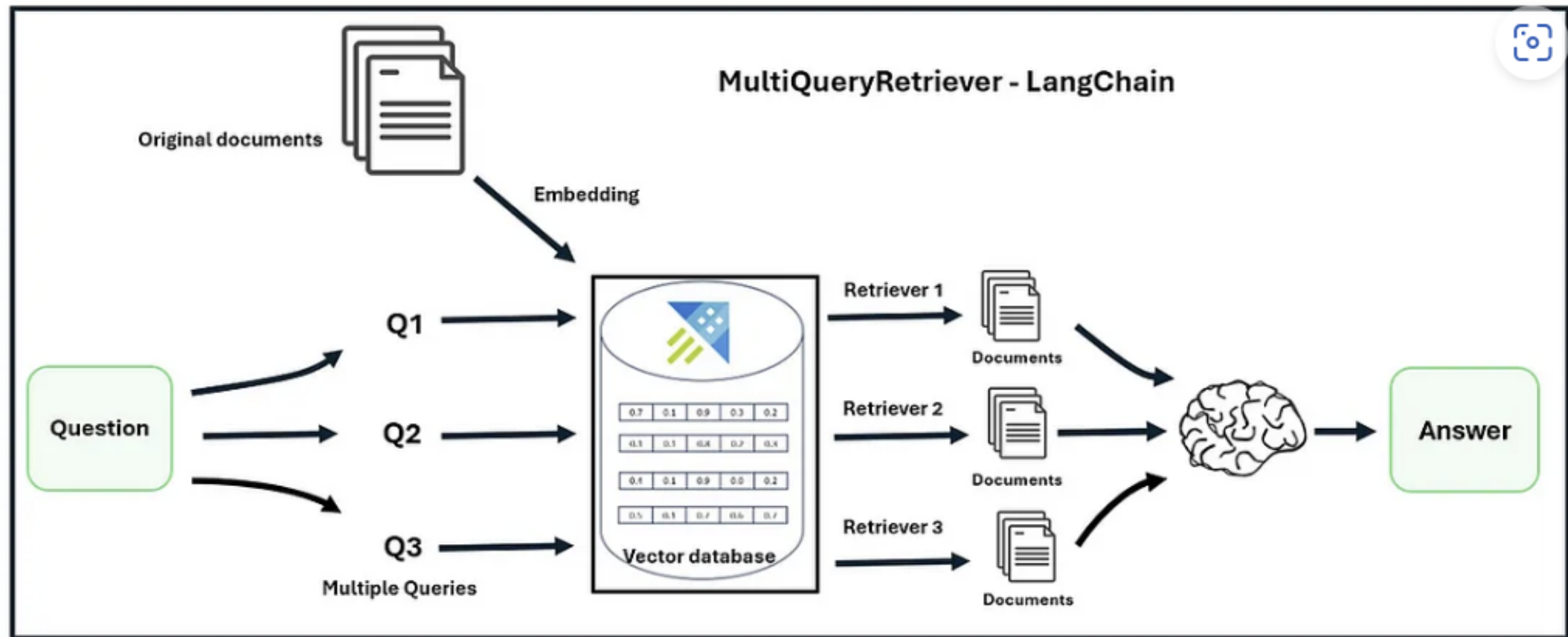
- Query rewriting aligns the semantics of a query and a document.
- An LLM is used to rephrase the user's query and give it another shot. It's important to note that two questions that might look the same to a human may not appear similar in the embedding space.
- The Multi-query Retrieval method utilizes 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.

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.

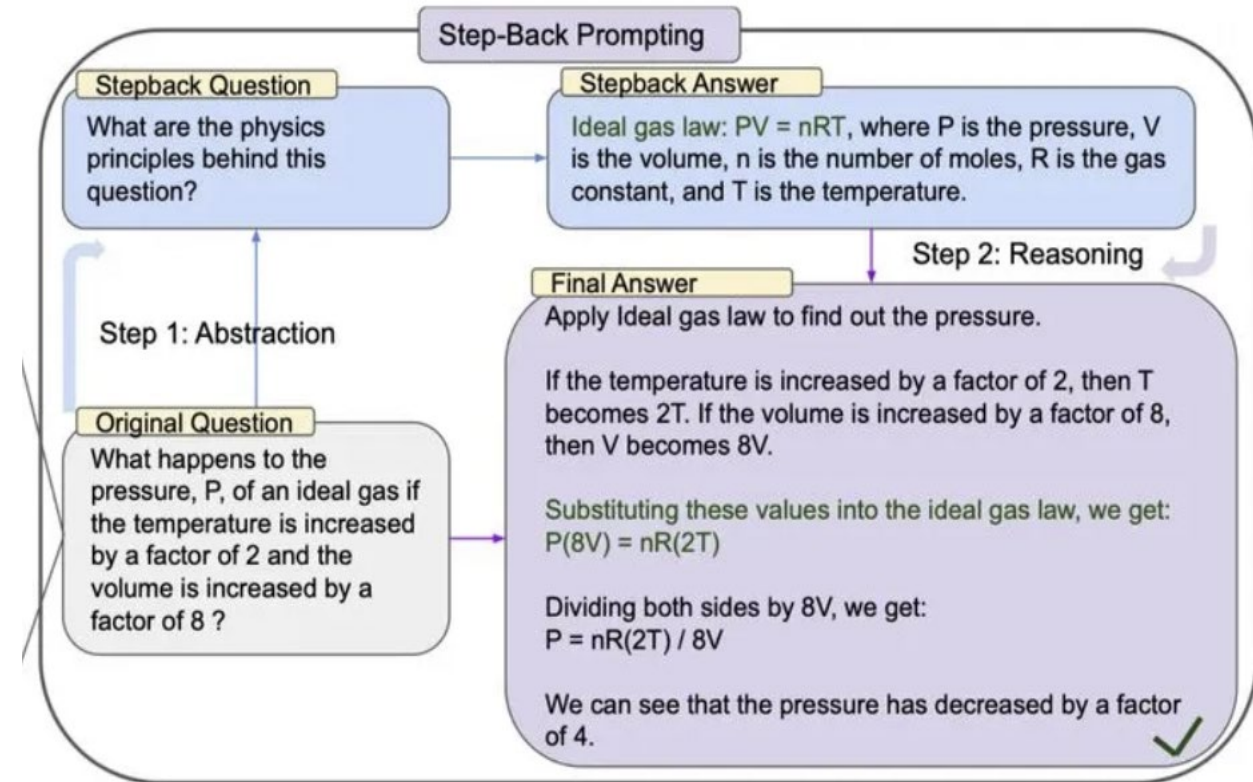
Original question: I am interested to know more about quantum mechanics, such as, is it related to quantum computing or nano computing and also will I get a job if I study it?

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?

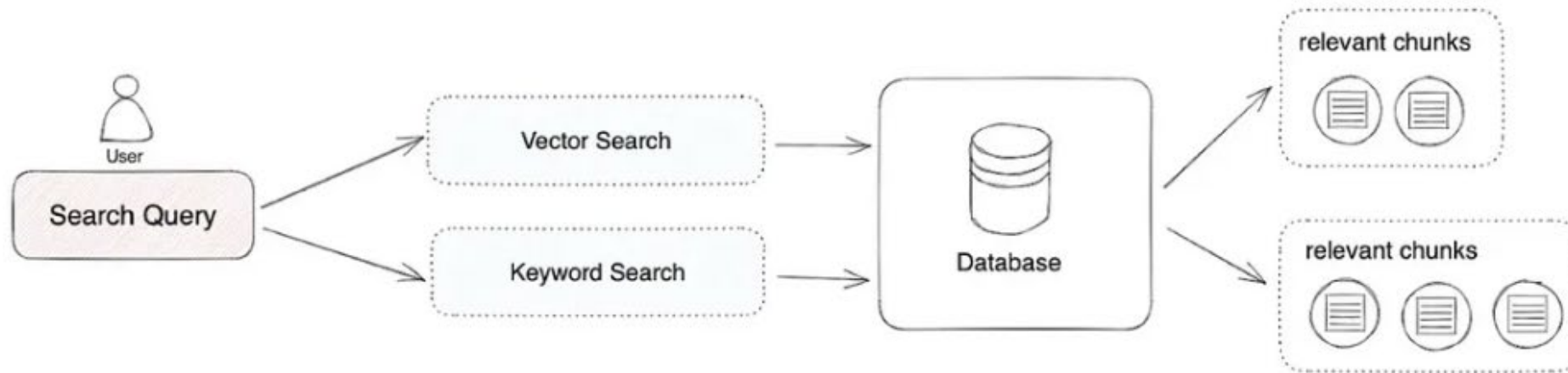
Step Back Prompt

- From the given query, develop broader concepts and principles
- Retrieve Information for both original and step-back questions



Hybrid Search Retrievers

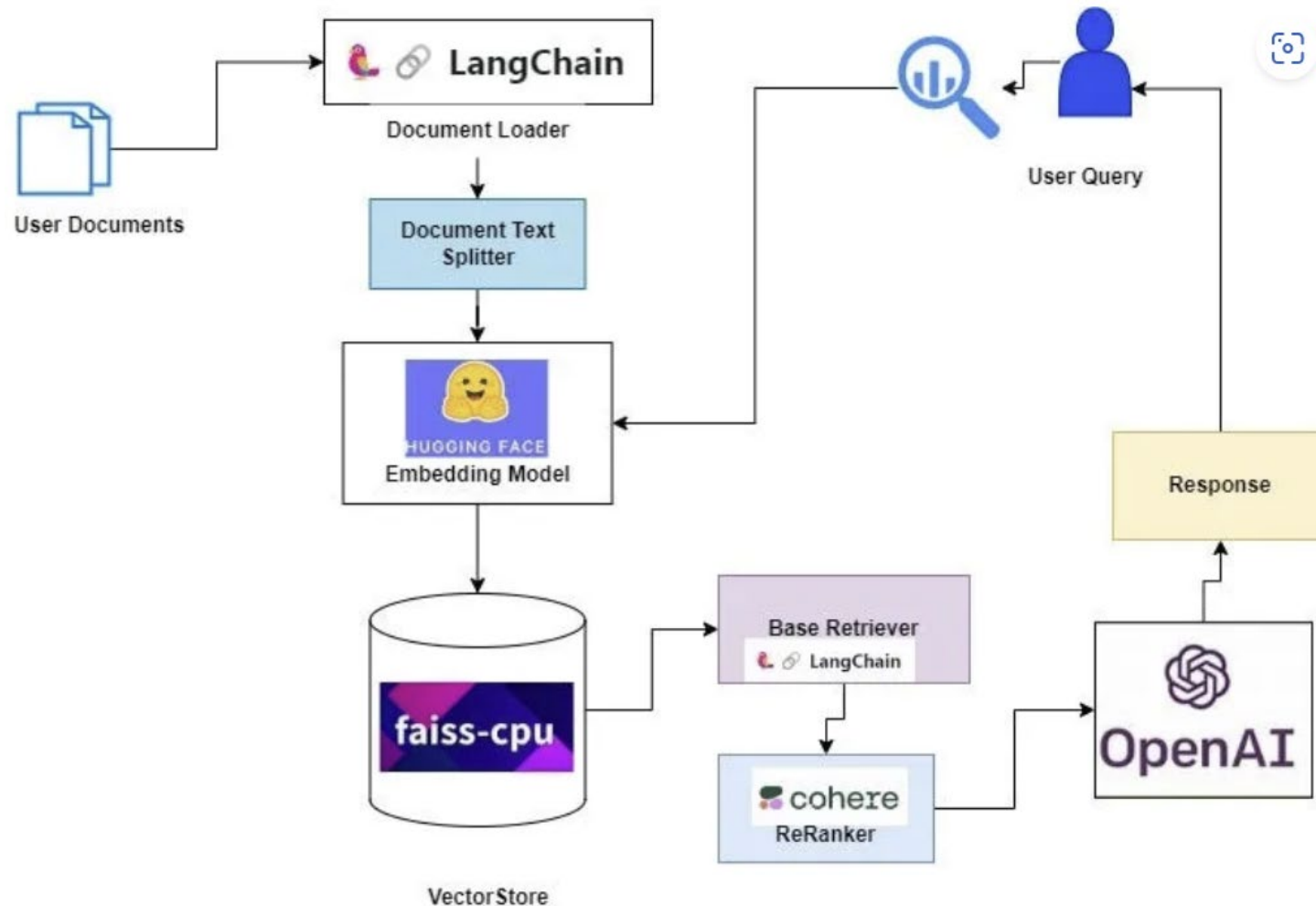
Hybrid Search



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

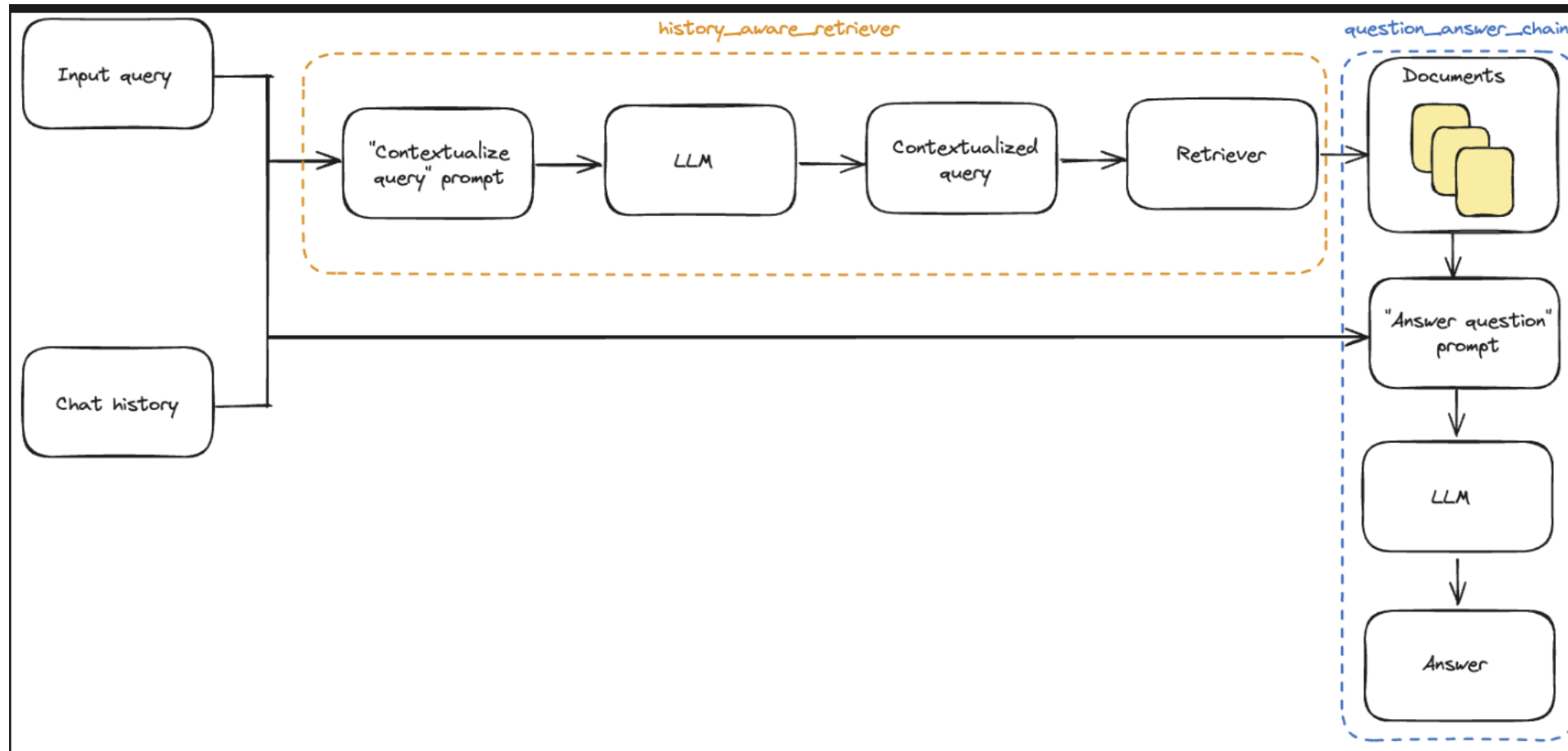


Re-Ranking the retrieved document using Cohere Reranker

Chat History

- 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

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)
```

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)
```

RAG on Tables: 3 strategies

- Here, we consider a case where there is a document that has both text and tables
- Strategies are:
 1. Use a large context size (e.g. 128k) LLM, feed the entire document with table
 2. Extract the table as a preprocessing step and use the RAG on table
 3. Chunk the document the usual way, make some assumption like a given table may not cross page boundaries.

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?