Python code snippets using LangChain, LlamaIndex, HuggingFace, OpenAI and others

Retrieval Augmented Generation

A Simple Introduction

Abhinav Kimothi

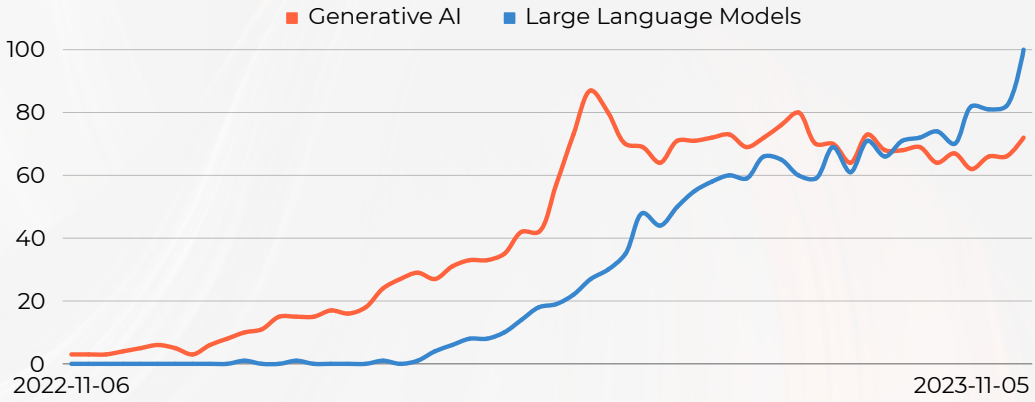
Swipe to view pages

Table of Contents

What is RAG?

Retrieval Augmented Generation

30th November, 2022 will be remembered as the watershed moment in artificial intelligence. OpenAI released ChatGPT and the world was mesmerised. Interest in previously obscure terms like Generative AI and Large Language Models (LLMs), was unstoppable over the following 12 months.



Google Trends - Interest Over Time (Nov'22 to Nov'23)

The Curse Of The LLMs

As usage exploded, so did the expectations. Many users started using ChatGPT as a source of information, like an alternative to Google. As a result, they also started encountering prominent weaknesses of the system. Concerns around copyright, privacy, security, ability to do mathematical calculations etc. aside, people realised that there are two major limitations of Large Language Models.

A Knowledge Cut-off date

Training an LLM is an expensive and time-consuming process. LLMs are trained on massive amount of data. The data that LLMs are trained on is therefore historical (or dated).
e.g. The latest GPT4 model by OpenAI has knowledge only till April 2023 and any event that happened post that date, the information is not available to the model.

Hallucinations

Often, it was observed that LLMs provided responses that were factually incorrect. Despite being factually incorrect, the LLM responses "sounded" extremely confident and legitimate. This characteristic of "lying with confidence" proved to be one of the biggest criticisms of ChatGPT and LLM techniques, in general.

> Users look at LLMs for knowledge and wisdom, yet LLMs are sophisticated predictors of what word comes next.

The Hunger For More

While the weaknesses of LLMs were being discussed, a parallel discourse around providing context to the models started. In essence, it meant creating a ChatGPT on proprietary data.

The Challenge

- Make LLMs respond with up-to-date information
- Make LLMs not respond with factually inaccurate information
- Make LLMs aware of proprietary information

  Providing LLMs with information not in their memory

Providing Context

While model re-training/fine-tuning/reinforcement learning are options that solve the aforementioned challenges, these approaches are time-consuming and costly. In majority of the use-case, these costs are prohibitive.

In May 2020, researchers in their paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" explored models which combine pre-trained parametric and non-parametric memory for language generation.

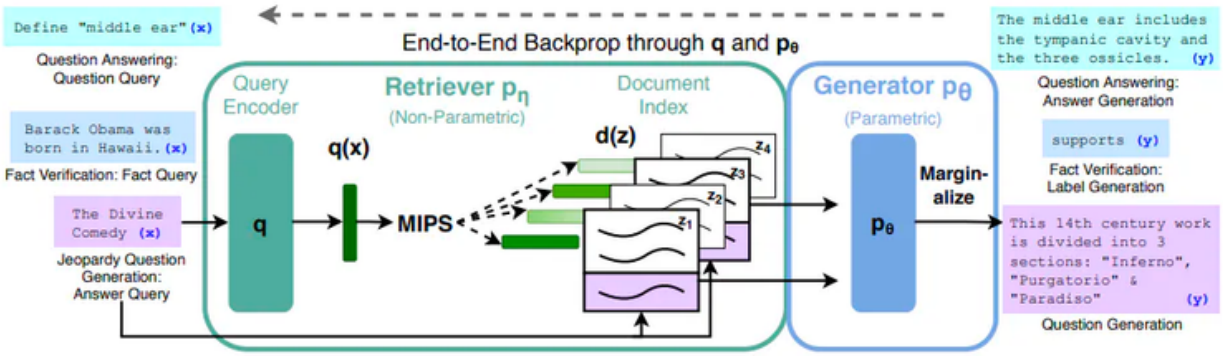
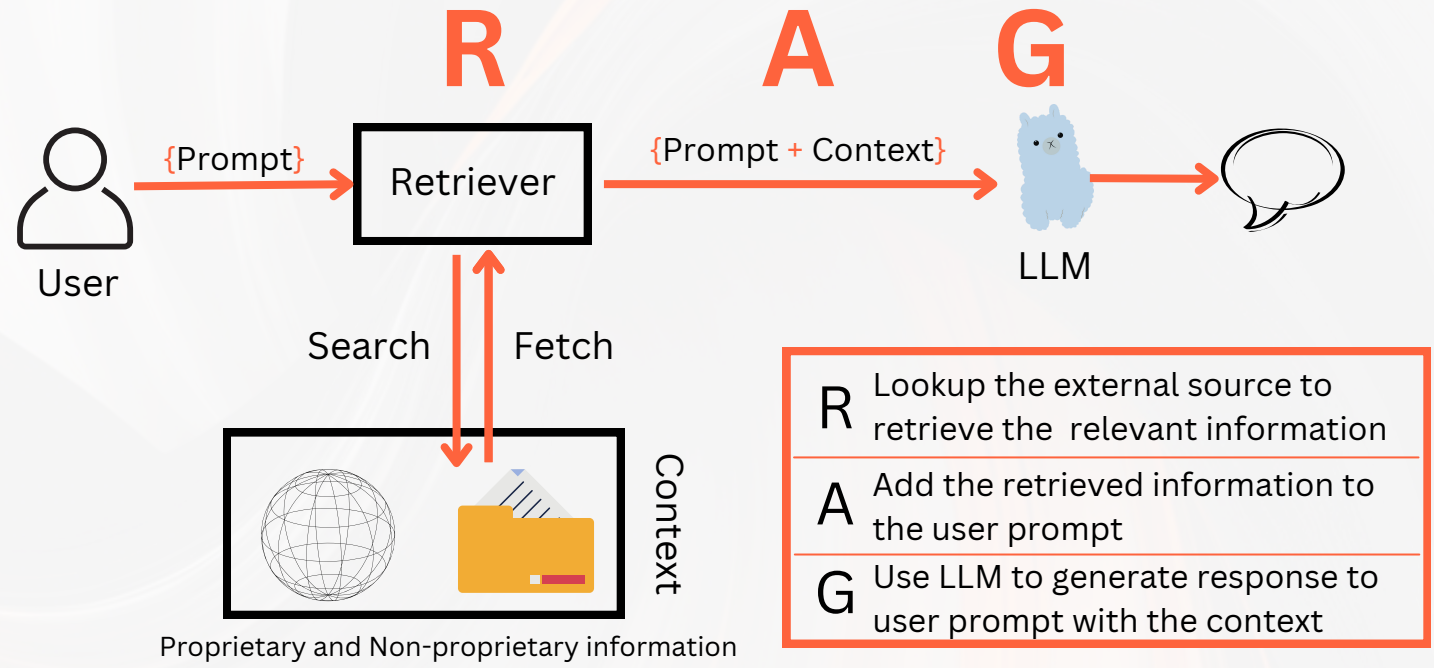


Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder + Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x, we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i. For final prediction y, we treat z as a latent variable and marginalize over seq2seq predictions given different documents.

So, What is RAG?

In 2023, RAG has become one of the most used technique in the domain of Large Language Models.



What is RAG?

User enters a prompt/query

↓

Retriever searches and fetches information relevant to the prompt (e.g. from the internet or internet data warehouse)

↓

Retrieved relevant information is augmented to the prompt as context

↓

LLM is asked to generate response to the prompt in the context (augmented information)

↓

User receives the response

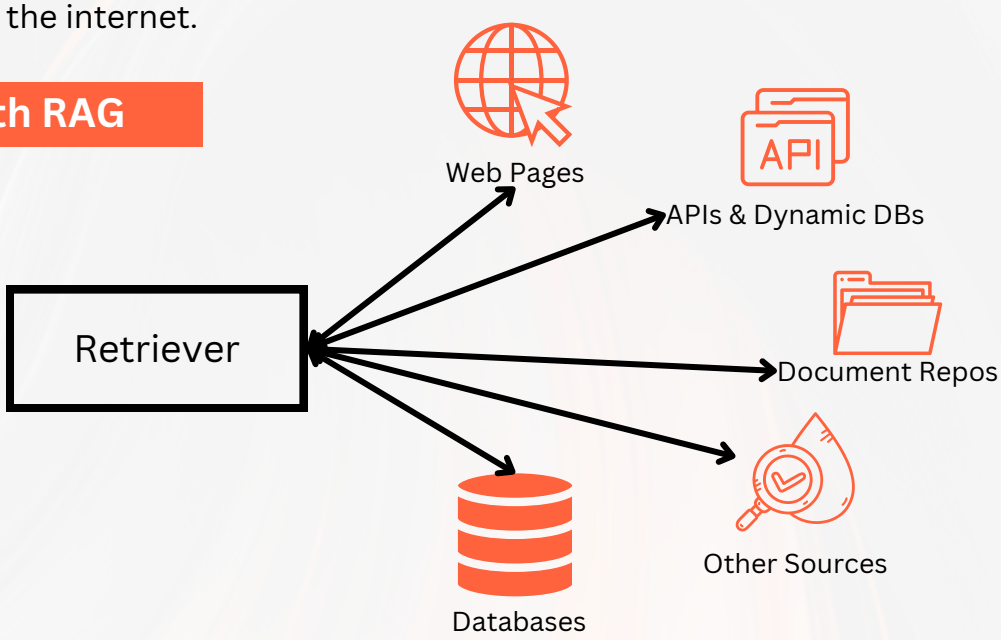
A Naive RAG workflow

How does RAG help?

Unlimited Knowledge

The Retriever of an RAG system can have access to external sources of information. Therefore, the LLM is not limited to its internal knowledge. The external sources can be proprietary documents and data or even the internet.

Without RAG | **With RAG**



An LLM has knowledge only of the data it has been trained on
Also called **Parametric Memory** (information stored in the model parameters)

Retriever searches and fetches information that the LLM has not necessarily been trained on. This adds to the LLM memory and is passed as context in the prompts. Also called **Non-Parametric Memory** (information available outside the model parameters)

- Expandable to all sources
- Easier to update/maintain
- Much cheaper than retraining/fine-tuning

The effort lies in creation of the knowledge base

Confidence in Responses

With the context (extra information that is retrieved) made available to the LLM, the confidence in LLM responses is increased.

Context Awareness

Added information assists LLMs in generating responses that are accurate and contextually appropriate

Source Citation

Access to sources of information improves the transparency of the LLM responses

Reduced Hallucinations

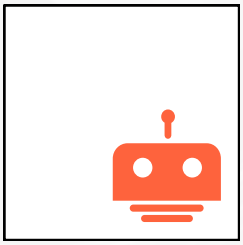
RAG enabled LLM systems are observed to be less prone to hallucinations than the ones without RAG

RAG Use Cases

The development of RAG technique is rooted in use cases that were limited by the inherent weaknesses of the LLMs. As of today some commercial applications of RAG are in -

Document Question Answering Systems

By providing access to proprietary enterprise document to an LLM, the responses are limited to what is provided within them. A retriever can search for the most relevant documents and provide the information to the LLM. Check out this blog for an example



Conversational agents

LLMs can be customised to product/service manuals, domain knowledge, guidelines, etc. using RAG. The agent can also route users to more specialised agents depending on their query. SearchUnify has an LLM+RAG powered conversational agent for their users.

Real-time Event Commentary

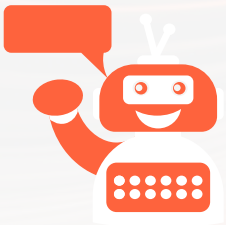
Imagine an event like a sports or a new event. A retriever can connect to real-time updates/data via APIs and pass this information to the LLM to create a virtual commentator. These can further be augmented with Text To Speech models.IBM leveraged the technology for commentary during the 2023 US Open

Content Generation

The widest use of LLMs has probably been in content generation. Using RAG, the generation can be personalised to readers, incorporate real-time trends and be contextually appropriate. Yarnit is an AI based content marketing platform that uses RAG for multiple tasks.

Personalised Recommendation

Recommendation engines have been a game changes in the digital economy. LLMs are capable of powering the next evolution in content recommendations. Check out Aman's blog on the utility of LLMs in recommendation systems.

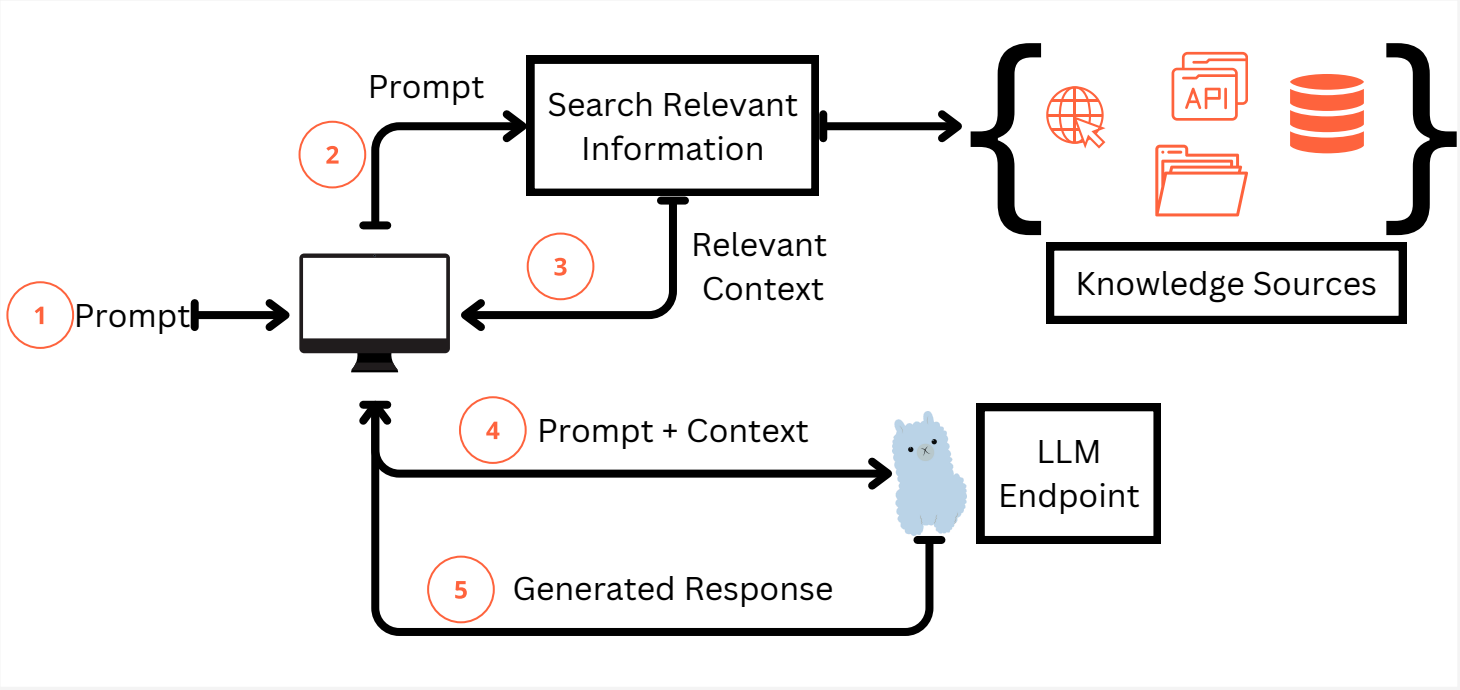


Virtual Assistants

Virtual personal assistants like Siri, Alexa and others are in plans to use LLMs to enhance the experience. Coupled with more context on user behaviour, these assistants can become highly personalised.

RAG Architecture

Let's revisit the five high level steps of an RAG enabled system



RAG System

1. User writes a prompt or a query that is passed to an orchestrator
2. Orchestrator sends a search query to the retriever
3. Retriever fetches the relevant information from the knowledge sources and sends back
4. Orchestrator augments the prompt with the context and sends to the LLM
5. LLM responds with the generated text which is displayed to the user via the orchestrator

Two pipelines become important in setting up the RAG system. The first one being setting up the knowledge sources for efficient search and retrieval and the second one being the five steps of the generation.

Indexing Pipeline

Data for the knowledge is ingested from the source and indexed. This involves steps like splitting, creation of embeddings and storage of data.

RAG Pipeline

This involves the actual RAG process which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model

Indexing Pipeline

The indexing pipeline sets up the knowledge source for the RAG system. It is generally considered an offline process. However, information can also be fetched in real time. It involves four primary steps.

Loading

This step involves extracting information from different knowledge sources a loading them into documents.

Splitting

This step involves splitting documents into smaller manageable chunks. Smaller chunks are easier to search and to use in LLM context windows.

Embedding

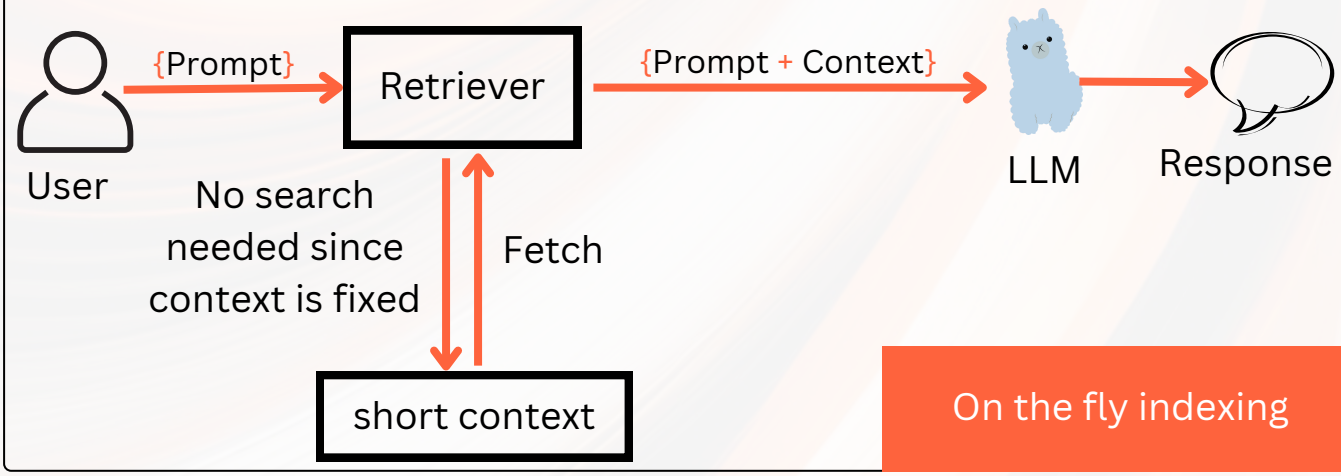
This step involves converting text documents into numerical vectors. ML models are mathematical models and therefore require numerical data.

Storing

This step involves storing the embeddings vectors. Vectors are typically stored in Vector Databases which are best suited for searching.

Offline Indexing pipelines are typically used when a knowledge base with large amount of data is being built for repeated usage e.g. a number of enterprise documents, manuals etc.

In cases where only a fixed small amount of one time data is required e.g. a 300 word blog, there is no need for storing the data. The blog text can either be directly passed in the LLM context window or a temporary vector index can be created.



Retrieval Augmented Generation

For Pages 10-28, Download Your Free Copy of Complete Notes from Gumroad

https://abhinavkimothi.gumroad.com/l/RAG

DOWNLOAD

Storing

We are at the last step of creating the indexing pipeline. We have loaded and split the data, and created the embeddings. Now, for us to be able to use the information repeatedly, we need to store it so that it can be accessed on demand. For this we use a special kind of database called the **Vector Database**.

What is a Vector Database?

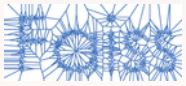
For those familiar with databases, **indexing** is a data structure technique that allows users to quickly retrieve data from a database. Vector databases specialise in indexing and storing embeddings for **fast retrieval** and **similarity search**.

A strip down variant of a Vector Database is a Vector Index like FAISS (Facebook AI Similarity Search). It is this vector indexing that improves the search and retrieval of vector embeddings. Vector Databases augment the indexing with typical database features like data management, metadata storage, scalability, integrations, security etc.

In short, Vector Databases provide -

- Scalable Embedding Storage.
- Precise Similarity Search.
- Faster Search Algorithm.

Popular Vector Databases



Facebook AI Similarity search is a vector index released with a library in 2017

Pinecone is one of the most popular managed Vector DB for large scale

Weaviate is an open source vector database that stores both objects and vectors

Chromadb is also an open source vector database.

With the growth in demand for vector storage, it can be anticipated that all major database players will add the vector indexing capabilities to their offerings.

How to choose a Vector Database?

All vector databases offer the same basic capabilities. Your choice should be influenced by the nuance of your use case matching with the value proposition of the database.

A few things to consider -

- Balance search accuracy and query speed based on application needs. Prioritize accuracy for precision applications or speed for real-time systems.
- Weigh increased flexibility vs potential performance impacts. More customization can add overhead and slow systems down.
- Evaluate data durability and integrity requirements vs the need for fast query performance. Additional persistence safeguards can reduce speed.
- Assess tradeoffs between local storage speed and access vs cloud storage benefits like security, redundancy and scalability.
- Determine if tight integration control via direct libraries is required or if ease-of-use abstractions like APIs better suit your use case.
- Compare advanced algorithm optimizations, query features, and indexing vs how much complexity your use case necessitates vs needs for simplicity.
- Cost considerations - while you many incur regular cost in a fully managed solution, a self hosted one might prove costlier if not managed well

User Friendly for PoCs

Chroma

Higher Performance

FAISS

Weaviate

Customization

Pinecone

There are many more Vector DBs. For a comprehensive understanding of the pros and cons of each, this blog is highly recommended

Storing Embeddings in Vector DBs

To store the embeddings, LangChain and LlamaIndex can be used for quick prototyping. The more nuanced implementation will depend on the choice of the DB, use case, volume etc.

Example : FAISS from langchain.vectorstores

In this example, we complete our indexing pipeline for one document.

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **OpenAIEmbeddings**
4. Storing the embeddings into **FAISS** vector index

```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS


loader=TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
document=loader.load()
text_splitter = RecursiveCharacterTextSplitter(chunk_size=10000,
                                               chunk_overlap=2000)
docs = text_splitter.split_documents(document)
num_emb=len(docs)
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)
db = FAISS.from_documents(docs, embeddings)
```

You'll have to address the following dependencies.

1. Install openai, tiktoken and faiss-cpu or faiss-gpu
2. Get an OpenAI API key

Now that our knowledge base is ready, let's quickly see it in action. Let's performa a search on the FAISS index we've just created.

Similarity search

In the YouTube video, for which we have indexed the transcript, Andrej Karpathy talks about the idea of LLM as an operating system. Let's perform a search on this.

Query : What did Andrej say about LLM operating system?

```
query="What did Andrej say about LLM operating system?"
docs = db.similarity_search(query)
docs[0].page_content
```

```
"operating system and um basically this process
is coordinating a lot of resources be they memory
or computational tools for problem solving so let's
think through based on everything I've shown you
what an LLM might look like in a few years it can
read and generate text it has a lot more knowledge
any single human about all the subjects it can browse
the internet or reference local files uh through
retrieval augmented generation it can use existing
software infrastructure like calculator python Etc it
can see and generate images and videos it can hear and
speak and generate music it can think for a long time using
a system too it can maybe self-improve in some narrow domains
 that have a reward function available maybe it can be customized
and fine-tuned to many specific tasks maybe there's lots of
llm experts almost uh living in an App Store that can sort of
coordinate uh for problem solving and so I see a lot of equivalence
between this new llm OS operating system and operating"
```

We can see here that out of the entire text, we have been able to retrieve the specific chunk talking about the LLM OS. We'll look at it in detail again in the RAG pipeline

Example : Chroma from langchain.vectorstores

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **all-MiniLM-L6-v2**
4. Storing the embeddings into **Chromadb**

```python
from langchain.document_loaders import TextLoader
from langchain.embeddings.sentence_transformer import SentenceTransformerEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma

# load the document and split it into chunks
loader = TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
documents = loader.load()

# split it into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
                                               chunk_overlap=200)

docs = text_splitter.split_documents(documents)

# create the open-source embedding function
embedding_function = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")

# load it into Chroma
db = Chroma.from_documents(docs, embedding_function)

# query it
query = "What did Andrej say about LLM operating system?"
docs = db.similarity_search(query)

# print results
print(docs[0].page_content)
```

```
'''
llm trying to page relevant information in and out of its
 context window to perform your task um and so a lot of
 other I think connections also exist I think there's
 equivalence of um multi-threading multiprocessing speculative
 execution uh there's equivalent of in the random access memory
 in the context window there's equivalence of user space and
 kernel space and a lot of other equivalents to today's
 operating systems that I didn't fully cover but fundamentally
 the other reason that I really like this analogy of llms kind
 of becoming a bit of an operating system ecosystem is that
 there are also some equivalence I think between the current
 operating systems and the uh and what's emerging today so for
 example in the desktop operating system space we have a few
 proprietary operating systems like Windows and Mac OS but we
 also have this open source ecosystem of a large diversity of
 operating systems based on Linux in the same way here we have
 some proprietary operating systems like GPT
'''
```

LangChain

All LangChain VectorDB Integrations

Indexing Pipeline Recap

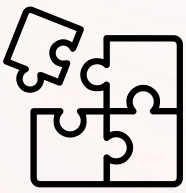
We covered the indexing pipeline in its entirety. A quick recap -

Loading

- A variety of data loaders from LangChain and LlamaIndex can be leveraged to load data from all sort of sources.
- Loading documents from a list of sources may turn out to be a complicated process. Make sure to plan for all the sources and loaders in advance.
- More often than naught, transformations/clean-ups to the loaded data will be required

Splitting

- Documents need to be split for ease of search and limitations of the llm context windows
- Chunking strategies are dependent on the use case, nature of content, embeddings, query length & complexity
- Chunking methods determine how the text is split and how the chunks are measured



Embedding

- Embeddings are vector representations of data that capture meaningful relationships between entities
- Some embeddings work better for some use cases

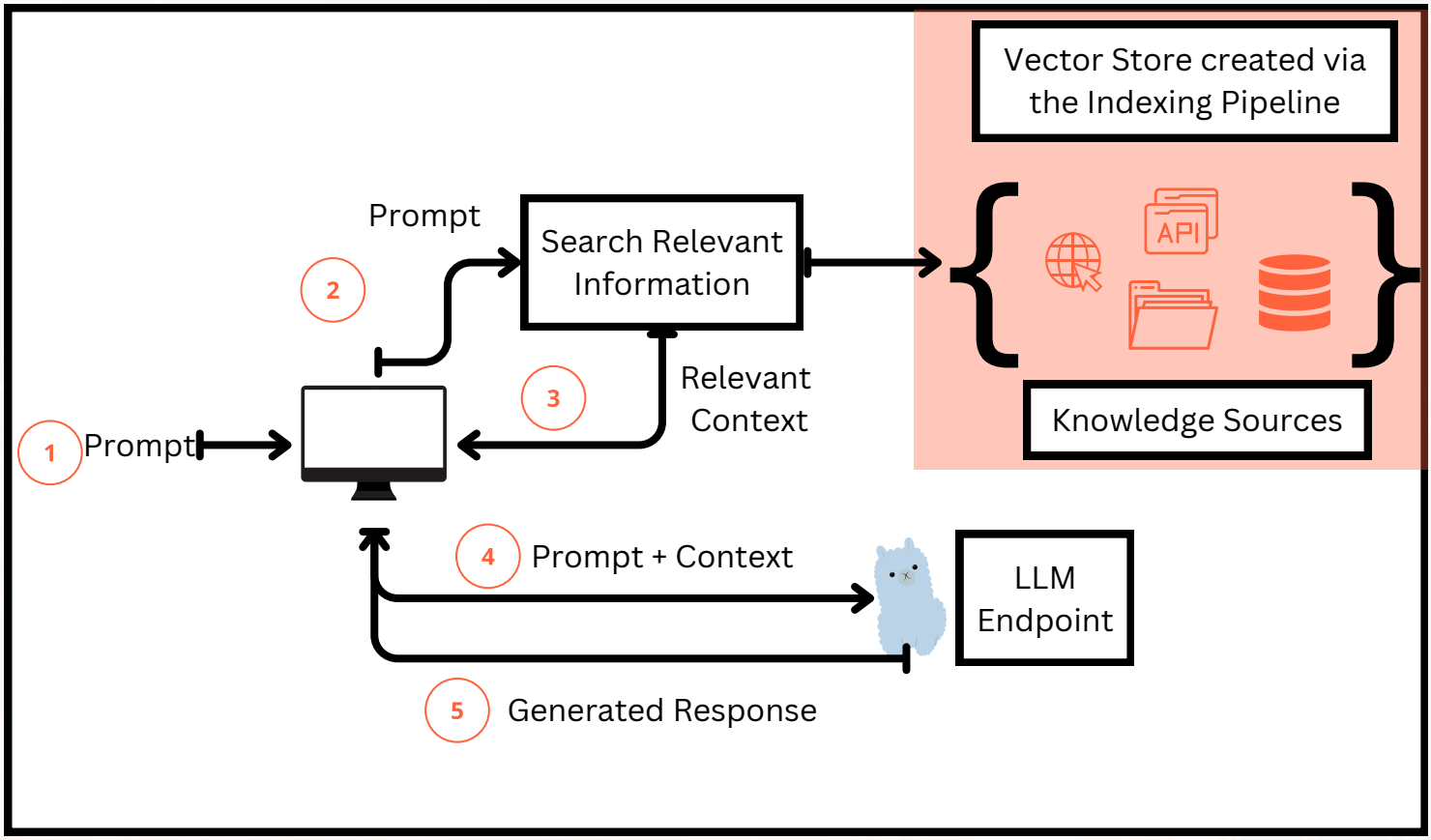
Storing

- Vector databases specialise in indexing and storing embeddings for fast retrieval and similarity search
- Different vector databases present different benefits and can be used in accordance with the use case

RAG Pipeline

Now that the knowledge base has been created in the indexing pipeline, the main generation or the **RAG pipeline** will have to be setup for receiving the input and generating the output.

Let's revisit our architecture diagram.



RAG System

Generation Steps

1. User writes a prompt or a query that is passed to an orchestrator
2. Orchestrator sends a search query to the retriever
3. Retriever fetches the relevant information from the **knowledge sources** and returns
4. Orchestrator augments the prompt with the context and sends to the LLM
5. LLM responds with the generated text which is displayed to the user via the orchestrator

The knowledge sources highlighted above have been set up using the indexing pipeline. These sources can be served using "on-the-fly" indexing also

RAG Pipeline Steps

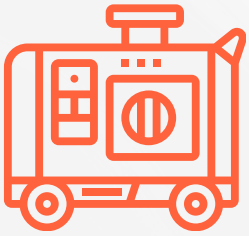
The three main steps in a RAG pipeline are

Search & Retrieval

This step involves searching for the context from the source (e.g. vector db)

Augmentation

This step involves adding the context to the prompt depending on the use case.



Generation

This step involves generating the final response from the large language model

An important consideration is how knowledge is stored and accessed. This has a bearing on the search & retrieval step.

Persistent Vector DBs

When a large volume of data is stored in vector databases, the retrieval and search needs to be quick. The relevance and accuracy of the search can be tested.

Temporary Vector Index

When data is temporarily stored in vector indices for one time use, the accuracy and relevance of the search needs to be ascertained

Small Data

Generally, when small amount of data is retrieved from pre-determined external sources, the augmentation of the data becomes more critical.

Indexing Pipeline

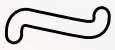
On the fly

Retrieval

Perhaps, the most critical step in the entire RAG value chain is searching and retrieving the relevant pieces of information (known as **documents**). When the user enters a query or a prompt, it is this system (**Retriever**) that is responsible for accurately fetching the correct snippet of information that is used in responding to the user query.

Retrievers accept a Query as input and return a list of Documents as output

Popular Retrieval Methods



Similarity Search

The similarity search functionality of vector databases forms the backbone of a Retriever. Similarity is calculated by calculating the distance between the embedding vectors of the input and the documents

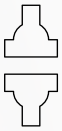
Maximum Marginal Relevance

MMR addresses redundancy in retrieval. MMR considers the relevance of each document only in terms of how much new information it brings given the previous results. MMR tries to reduce the redundancy of results while at the same time maintaining query relevance of results for already ranked documents/phrases

Multi-query Retrieval

Multi-query Retrieval automates prompt tuning using a language model to generate diverse queries for a user input, retrieving relevant documents from each query and combining them to overcome limitations and obtain a more comprehensive set of results. This approach aims to enhance retrieval performance by considering multiple perspectives on the same query.

Retrieval Methods

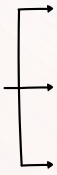


Contextual compression

Sometimes, relevant info is hidden in long documents with a lot of extra stuff. Contextual Compression helps with this by squeezing down the documents to only the important parts that match your search.

Multi Vector Retrieval

Sometimes it makes sense to store more than one vectors in a document. E.g A chapter, its summary and a few quotes. The retrieval becomes more efficient because it can match with all the different itypes of nformation that has been embedded.



Parent Document Retrieval

In breaking down documents for retrieval, there's a dilemma. Small pieces capture meaning better in embeddings, but if they're too short, context is lost. The Parent Document Retrieval finds a middle ground by storing small chunks. During retrieval, it fetches these bits, then gets the larger documents they came from using their parent IDs

Self Query

A self-querying retriever is a system that can ask itself questions. When you give it a question in normal language, it uses a special process to turn that question into a structured query. Then, it uses this structured query to search through its stored information. This way, it doesn't just compare your question with the documents; it also looks for specific details in the documents based on your question, making the search more efficient and accurate.

Retrieval Methods

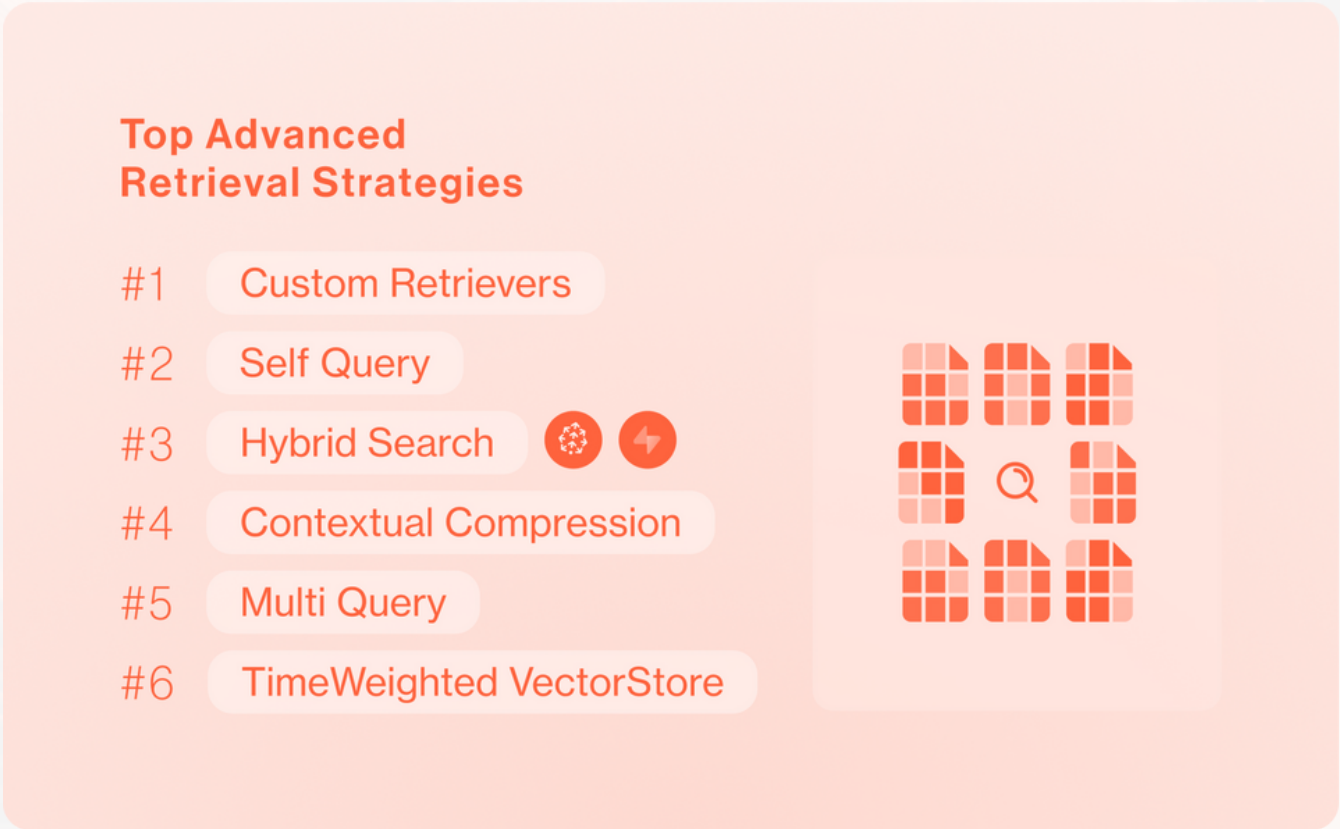
Time-weighted Retrieval

This method supplements the semantic similarity search with a time delay. It gives more weightage, then, to documents that are fresher or more used than the ones that are older

Ensemble Techniques

As the term suggests, multiple retrieval methods can be used in conjunction with each other. There are many ways of implementing ensemble techniques and use cases will define the structure of the retriever

Top Advanced Retrieval Strategies



Source : LangChain State of AI 2023

Example : Similarity Search using LangChain

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **all-MiniLM-L6-v2**
4. Storing the embeddings into **Chromadb**
5. Retrieving chunks using **similarity_search**

```python
from langchain.document_loaders import TextLoader
from langchain.embeddings.sentence_transformer import SentenceTransformerEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma

# load the document and split it into chunks
loader = TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
documents = loader.load()

# split it into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
                                               chunk_overlap=200)

docs = text_splitter.split_documents(documents)

# create the open-source embedding function
embedding_function = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")

# load it into Chroma
db = Chroma.from_documents(docs, embedding_function)

# query it
query = "What did Andrej say about LLM operating system?"
docs = db.similarity_search(query)

# print results
print(docs[0].page_content)
```

```
'''
llm trying to page relevant information in and out of its
 context window to perform your task um and so a lot of
 other I think connections also exist I think there's
 equivalence of um multi-threading multiprocessing speculative
 execution uh there's equivalent of in the random access memory
 in the context window there's equivalence of user space and
 kernel space and a lot of other equivalents to today's
 operating systems that I didn't fully cover but fundamentally
 the other reason that I really like this analogy of llms kind
 of becoming a bit of an operating system ecosystem is that
 there are also some equivalence I think between the current
 operating systems and the uh and what's emerging today so for
 example in the desktop operating system space we have a few
 proprietary operating systems like Windows and Mac OS but we
 also have this open source ecosystem of a large diversity of
 operating systems based on Linux in the same way here we have
 some proprietary operating systems like GPT
'''
```

Example : Similarity Vector Search

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **all-MiniLM-L6-v2**
4. Storing the embeddings into **Chromadb**
5. Converting input query into a **vector embedding**
6. Retrieving chunks using **similarity_search_by_vector**

```python
from langchain.document_loaders import TextLoader
from langchain.embeddings.sentence_transformer import SentenceTransformerEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma

# load the document and split it into chunks
loader = TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
documents = loader.load()

# split it into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
                                               chunk_overlap=200)

docs = text_splitter.split_documents(documents)

# create the open-source embedding function
embedding_function = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")

# load it into Chroma
db = Chroma.from_documents(docs, embedding_function)

# query it
query = "What did Andrej say about LLM operating system?"

# convert query to embedding
query_vector=embedding_function.embed_query(query)

# distance based search
docs = db.similarity_search_by_vector(query_vector)

# print results
print(docs[0].page_content)
```

```
'''llm trying to page relevant information in and out of its context window to perform your task um
and so a lot of other I think connections also exist I think there's equivalence of um multi-threading
multiprocessing speculative execution uh there's equivalent of in the random access memory in the context
window there's equivalence of user space and kernel space and a lot of other equivalents to today's
operating systems that I didn't fully cover but fundamentally the other reason that I really like
this analogy of llms kind of becoming a bit of an operating system ecosystem is that there are also
some equivalence I think between the current operating systems and the uh and what's emerging today
so for example in the desktop operating system space we have a few proprietary operating systems like
Windows and Mac OS but we also have this open source ecosystem of a large diversity of operating systems
based on Linux in the same way here we have some proprietary operating systems like GPT'''
```

How Similarity Vector Search is different from Similarity Search is that the query is also converted into a vector embedding from regular text

Example : Maximum Marginal Relevance

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **OpenAI Embeddings**
4. Storing the embeddings into **Qdrant**
5. Retrieving and ranking chunks using **max_marginal_relevance_search**

```python
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Qdrant

# load the document and split it into chunks
loader = TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
documents = loader.load()

# split it into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
                                               chunk_overlap=200)

docs = text_splitter.split_documents(documents)

# create the openai embedding function
embedding_function = OpenAIEmbeddings(openai_api_key=openai_api_key)

# load it into Qdrant
db = Qdrant.from_documents(docs, embedding_function, location=":memory:",
    collection_name="my_documents")

# query it
query = "What did Andrej say about LLM operating system?"

# max marginal relevance search
docs = db.max_marginal_relevance_search(query,k=2, fetch_k=10)

# print results
for i, doc in enumerate(docs):
    print(f"{i + 1}.", doc.page_content, "\n")
```

fetch_k = Number of documents in the initial retrieval
k = final number of reranked documents to output

Example : Multi-query Retrieval

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **OpenAI Embeddings**
4. Storing the embeddings into **Qdrant**
5. Set the LLM as **ChatOpenAI (gpt 3.5)**
6. Set up **logging** to see the query variations generated by the LLM
7. use **MultiQueryRetriever** & **get_relevant_documents** functions

```python
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Qdrant
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain.chat_models import ChatOpenAI

# load the document and split it into chunks
loader = TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
documents = loader.load()

# split it into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
docs = text_splitter.split_documents(documents)

# create the openai embedding function
embedding_function = OpenAIEmbeddings(openai_api_key=openai_api_key)

# load it into Qdrant
db = Qdrant.from_documents(docs, embedding_function, location=":memory:",collection_name="my_documents")

# query it
query = "What did Andrej say about LLM operating system?"

# set the LLM for multiquery
llm = ChatOpenAI(temperature=0, openai_api_key=openai_api_key)


# Multiquery retrieval using OpenAI
retriever_from_llm = MultiQueryRetriever.from_llm(retriever=db.as_retriever(), llm=llm)

# set up logging to see the queries generated
import logging
logging.basicConfig()
logging.getLogger("langchain.retrievers.multi_query").setLevel(logging.INFO)

# retrieved documents
unique_docs = retriever_from_llm.get_relevant_documents(query=query)

# print results
for i, doc in enumerate(unique_docs):
    print(f"{i + 1}.", doc.page_content, "\n")
```

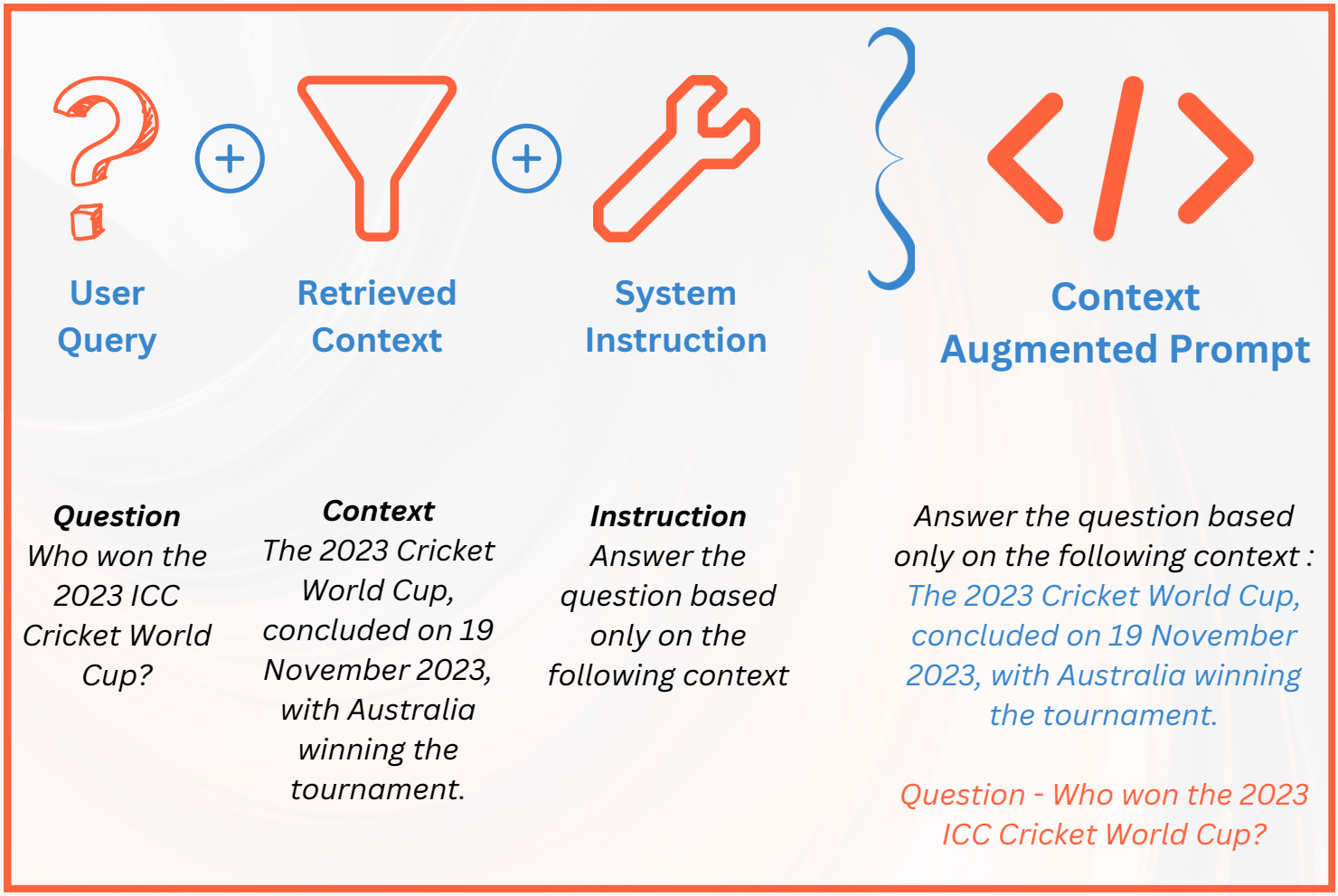
Example : Contextual compression

1. Loading our text file using **TextLoader**,
2. Splitting the text into chunks using **RecursiveCharacterTextSplitter**,
3. Creating embeddings using **OpenAI Embeddings**
4. Set up retriever as **FAISS**
5. Set the LLM as **ChatOpenAI (gpt 3.5)**
6. Use **LLMChainExtractor** as the compressor
7. use **ContextualCompressionRetriever** & **get_relevant_documents** functions

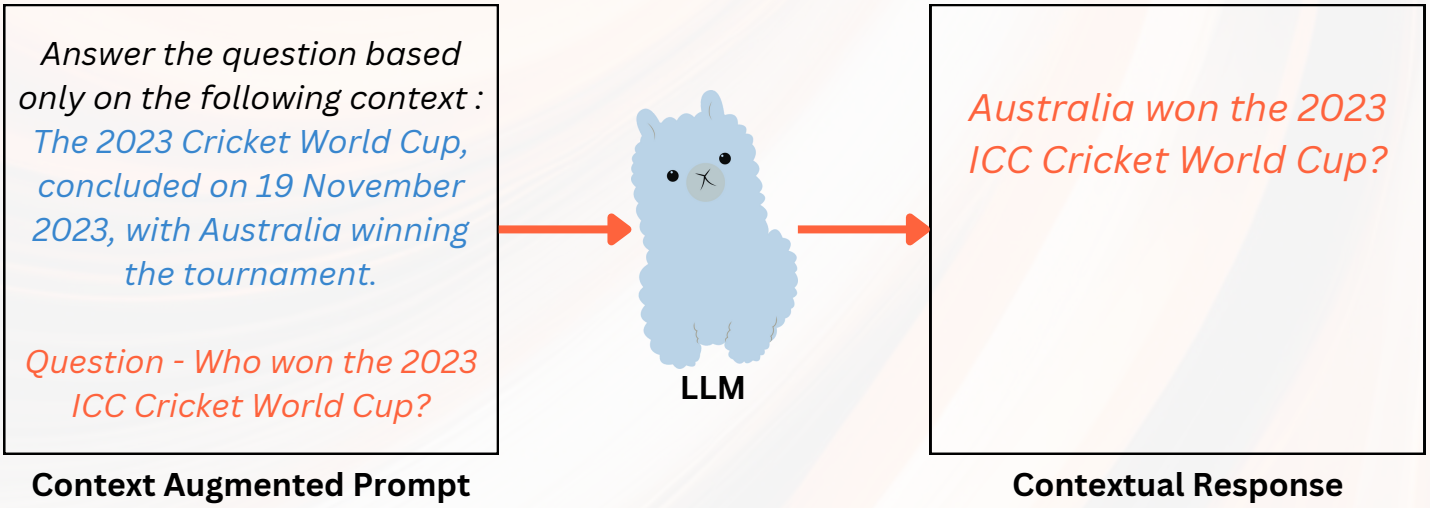
```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.llms import OpenAI
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

# load and split text
loader = TextLoader('../Data/AK_BusyPersonIntroLLM.txt')
documents = loader.load()
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
                                               chunk_overlap=200)
docs = text_splitter.split_documents(documents)
# save as vector embeddings
retriever = FAISS.from_documents(docs,
                OpenAIEmbeddings(
                    openai_api_key=openai_api_key)).as_retriever()

# use a compressor
llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
compressor = LLMChainExtractor.from_llm(llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=retriever)

query = "What did Andrej say about LLM operating system?"

# retrieve docs
compressed_docs = compression_retriever.get_relevant_documents(query)

# print docs
for i, doc in enumerate(unique_docs):
    print(f"{i + 1}.", doc.page_content, "\n")
```

Augmentation & Generation

Post-retrieval, the next set of steps include merging the user query and the retrieved context (**Augmentation**) and passing this merged prompt as an instruction to an LLM (**Generation**)



Augmentation with an Illustrative Example



Generation with an Illustrative Example

Evaluation

Building a PoC RAG pipeline is not overtly complex. LangChain and LlamaIndex have made it quite simple. Developing highly impressive Large Language Model (LLM) applications is achievable through brief training and verification on a limited set of examples. However, to enhance its robustness, thorough testing on a dataset that accurately mirrors the production distribution is imperative.

RAG is a great tool to address hallucinations in LLMs but...
even RAGs can suffer from hallucinations

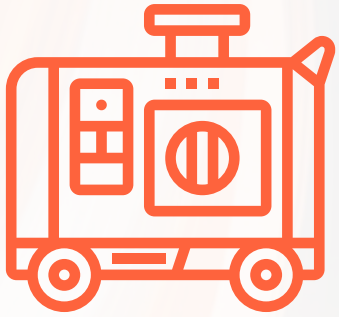
This can be because -

- The retriever fails to retrieve relevant context or retrieves irrelevant context
- The LLM, despite being provided the context, does not consider it
- The LLM instead of answering the query picks irrelevant information from the context

Two processes, therefore, to focus on from an evaluation perspective -

Search & Retrieval

- How good is the retrieval of the context from the Vector Database?
- Is it relevant to the query?
- How much noise (irrelevant information) is present?



Generation

- How good is the generated response?
- Is the response grounded in the provided context?
- Is the response relevant to the query?

Ragas (RAG Assessment)

Jithin James and Shahul ES from Exploding Gradients, in 2023, developed the Ragas framework to address these questions.

https://github.com/explodinggradients/ragas

Evaluation Data

To evaluate RAG pipelines, the following four data points are recommended

- A set of **Queries** or **Prompts** for evaluation
- **Retrieved Context** for each prompt
- Corresponding **Response** or **Answer** from LLM
- **Ground Truth** or known correct response

Evaluation Metrics

Evaluating Generation

Faithfulness Is the **Response** faithful to the **Retrieved Context**?

Answer Relevance Is the **Response** relevant to the **Prompt**?

Retrieval Evaluation

Context Relevance Is the **Retrieved Context** relevant to the **Prompt**?

Context Recall Is the **Retrieved Context** aligned to the **Ground Truth**?

Context Precision is the **Retrieved Context** ordered correctly?

Overall Evaluation

Answer Semantic Similarity

is the **Response** semantically similar to the **Ground Truth**?

Answer Correctness

is the **Response** semantically and factually similar to the **Ground Truth**?

Retrieval Augmented Generation

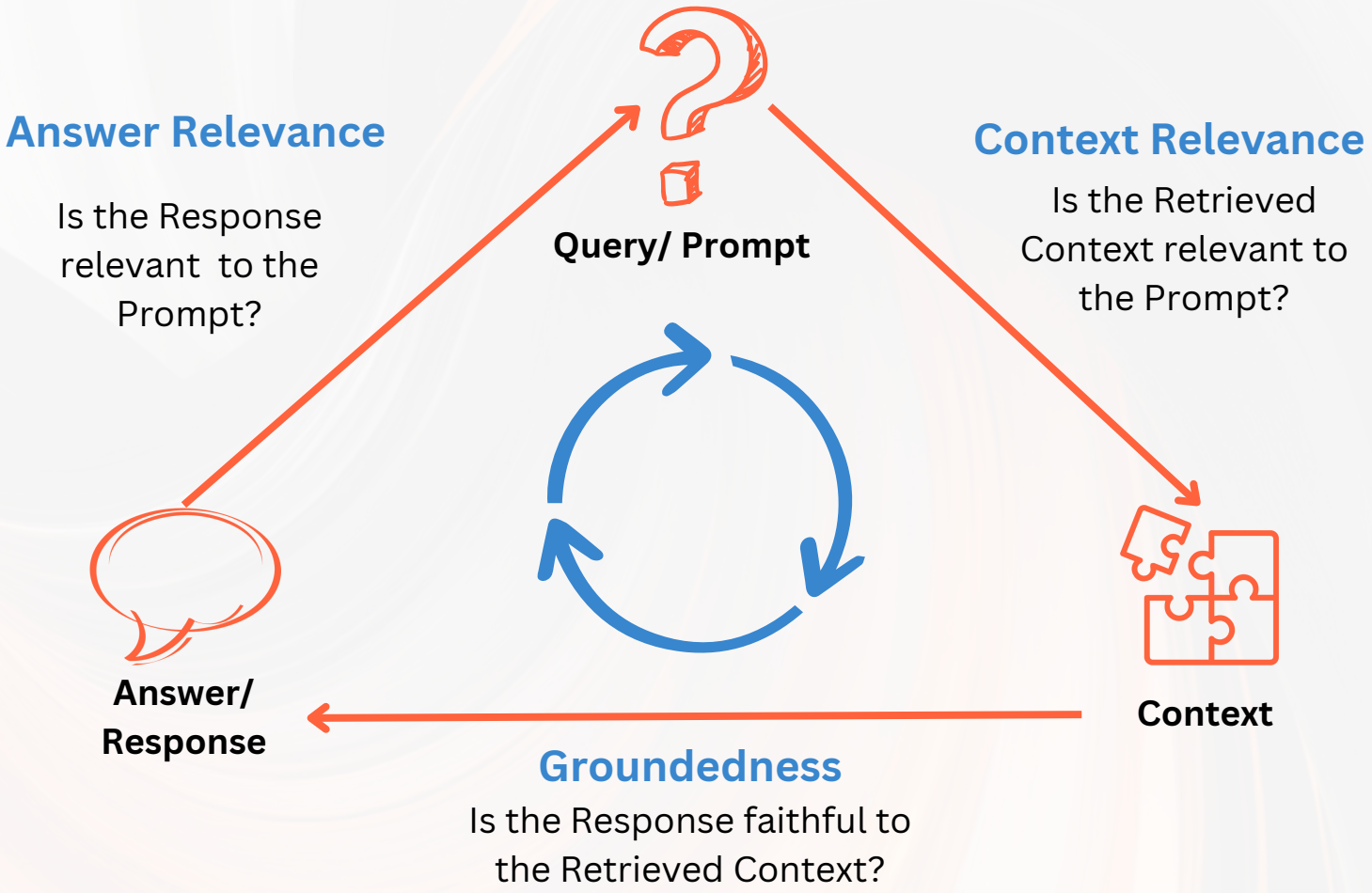
For Pages 48-54, Download Your Free Copy of Complete Notes from Gumroad

https://abhinavkimothi.gumroad.com/l/RAG

DOWNLOAD

The RAG Triad (TruLens)

The RAG triad is a framework proposed by TruLens to evaluate hallucinations along each edge of the RAG architecture.



Context Relevance:

- Verify quality by ensuring each context chunk is relevant to the input query

Groundedness:

- Verify groundedness by breaking down the response into individual claims.
- Independently search for evidence supporting each claim in the retrieved context.

Answer Relevance:

- Ensure the response effectively addresses the original question.
- Verify by evaluating the relevance of the final response to user input.

Trulens Documentation

RAG vs Finetuning vs Both

Supervised Finetuning (SFT) has fast become a popular method to customise and adapt foundation models for specific objectives. There has been a growing debate in the applied AI community around the application of fine-tuning or RAG to accomplish tasks.

RAG & SFT should considered as complementary, rather than competing, techniques.

RAG enhances the non-parametric memory of a foundation model without changing the parameters

SFT changes the parameters of a foundation model and therefore impacting the parametric memory

If the requirement dictates changes to the parametric memory and an increase in the non-parametric memory, then RAG and SFT can be used in conjunction

RAG Features

Connect to dynamic external data sources ✓

Reduce hallucinations ✓

Increase transparency (in terms of source of information) ✓

Works well only with very large foundation models ✗

Does not impact the style, tone, vocabulary of the foundation model ✗

SFT Features

Change the style, vocabulary, tone of the foundation model ✓

Can reduce model size ✓

Useful for deep domain expertise ✓

May not address the problem of hallucinations ✗

No improvement in transparency (as black box as foundation models) ✗

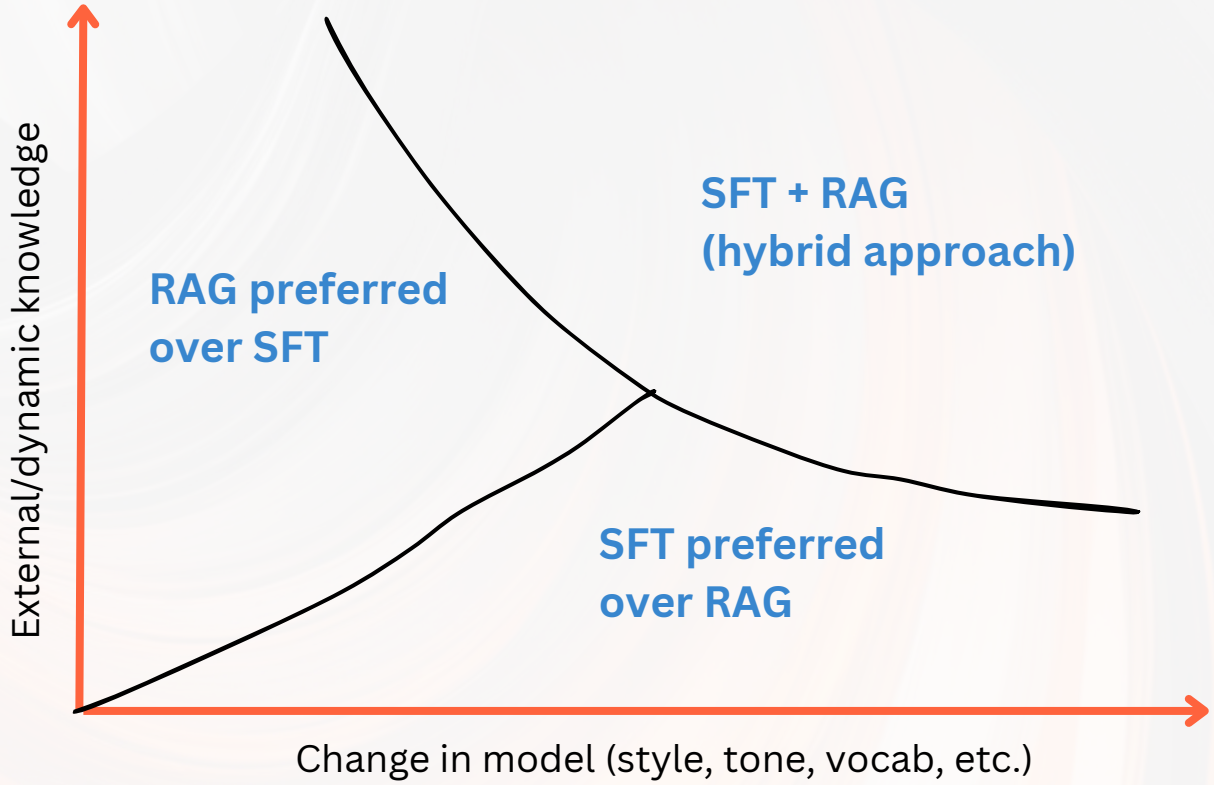
Important Use Case Considerations

Do you require usage of dynamic external data?

RAG preferred over SFT

Do you require changing the writing style, tonality, vocabulary of the model?

SFT preferred over RAG



RAG should be implemented (with or without SFT) if the use case requires

- Access to an external data source, especially, if the data is dynamic
- Resolving Hallucinations
- Transparency in terms of the source of information

For SFT, you'll need to have access to labelled training data

Other Considerations

Latency

RAG pipelines require an additional step of searching and retrieving context which introduces an inherent latency in the system

Scalability

RAG pipelines are modular and therefore can be scaled relatively easily when compared to SFT. SFT will require retraining the model with each additional data source

Cost

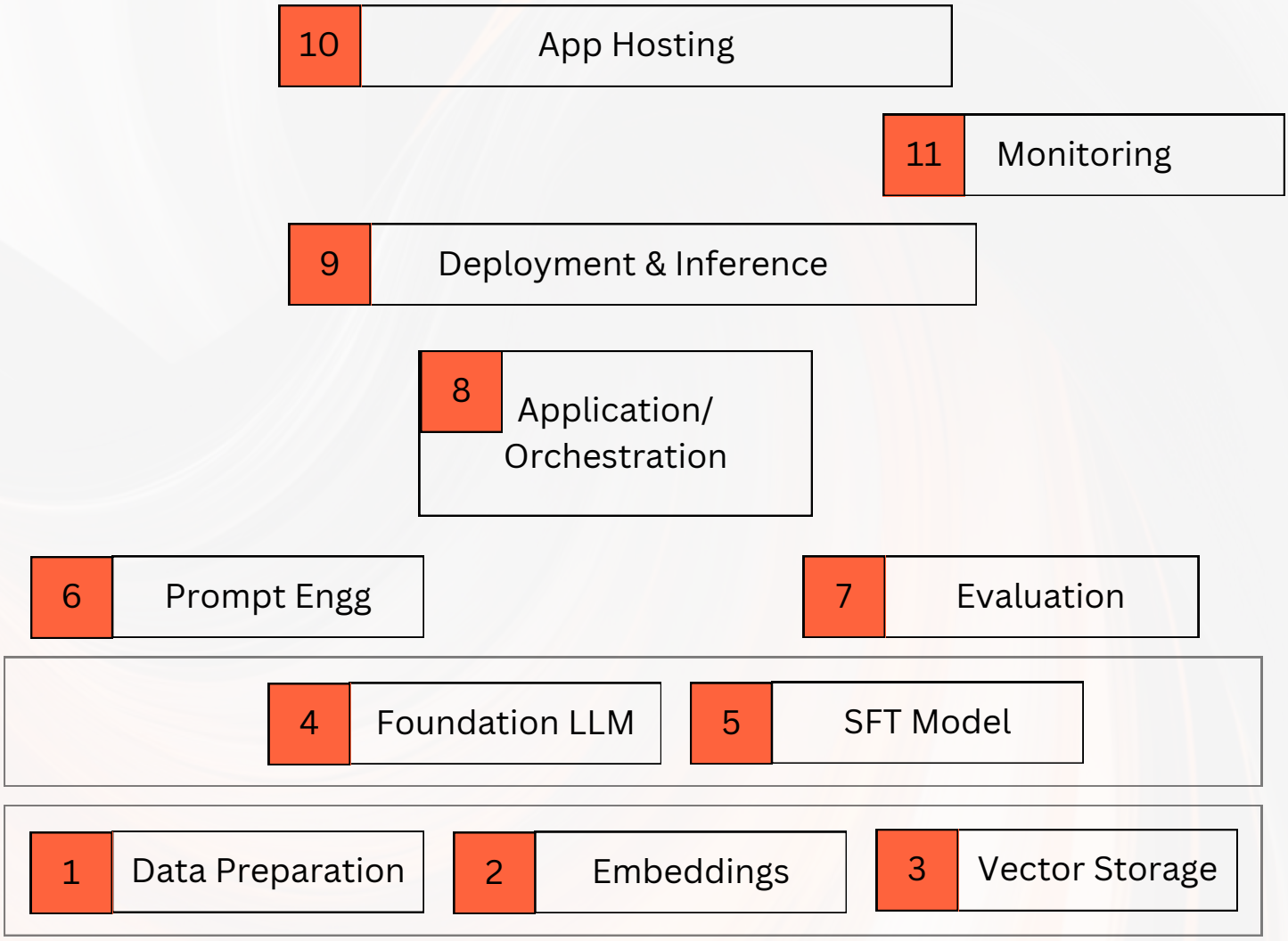
Both RAG and SFT warrant upfront investment. Training cost for SFT can vary depending on the technique and the choice of foundation model. Setting up the knowledge base and integration can be costly for RAG

Expertise

Creating RAG pipelines has become moderately simple with frameworks like LangChain and LlamaIndex. Fine-tuning on the other hand requires deep understanding of the techniques and creation of training data

Evolving RAG LLMOps Stack

The production ecosystem for RAG and LLM applications is still evolving. Early tooling and design patterns have emerged.



Data Layer

The foundation of RAG applications is the data layer. This involves -

- Data preparation - Sourcing, Cleaning, Loading & Chunking
- Creation of Embeddings
- Storing the embeddings in a vector store

We've seen this process in the creation of the **indexing pipeline**

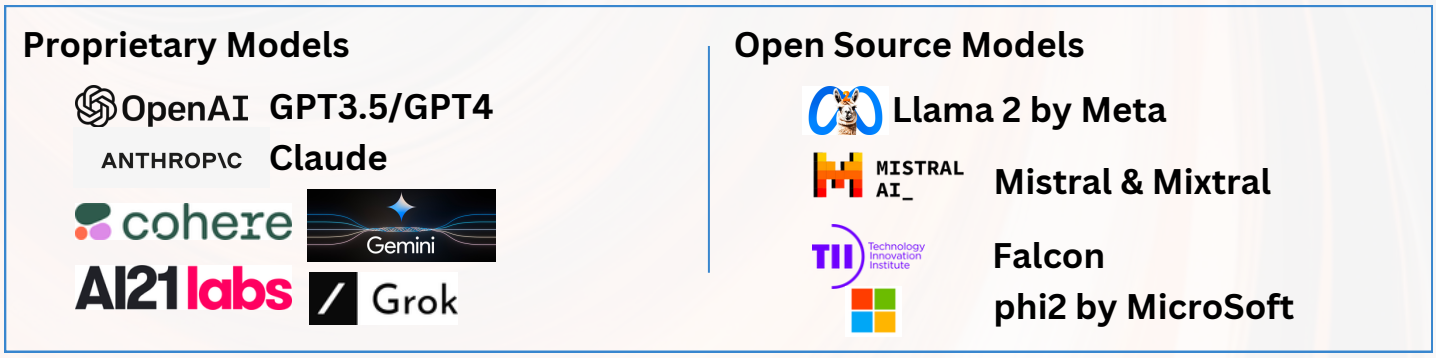
Popular Data Layer Vendors (Non Exhaustive)

Model Layer

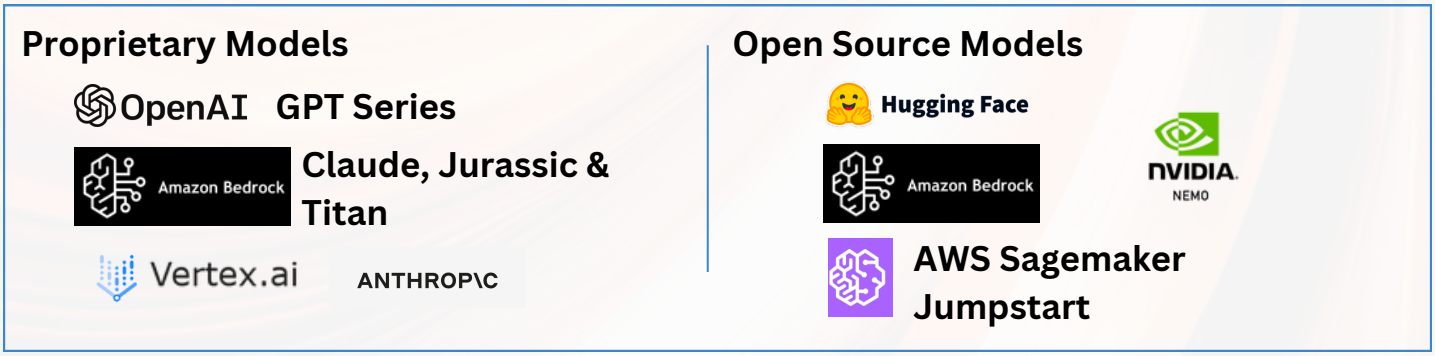
2023 can be considered a year of LLM wars. Almost every other week in the second half of the year a new model was released. Like there is no RAG without data, there is no RAG without an LLM. There are four broad categories of LLMs that can be a part of a RAG application

1. **A Proprietary Foundation Model** - Developed and maintained by providers (like OpenAI, Anthropic, Google) and is generally available via an API
2. **Open Source Foundation Model** - Available in public domain (like Falcon, Llama, Mistral) and has to be hosted and maintained by you.
3. **A Supervised Fine-Tuned Proprietary Model** - Providers enable fine-tuning of their proprietary models with your data. The fine-tuned models are still hosted and maintained by the providers and are available via an API
4. **A Supervised Fine-Tuned Open Source Model** - All Open Source models can be fine-tuned by you on your data using full fine-tuning or PEFT methods.

There are a lot of vendors that have enabled access to open source models and also facilitate easy fine tuning of these models



Popular proprietary and open source LLMs (Non Exhaustive)

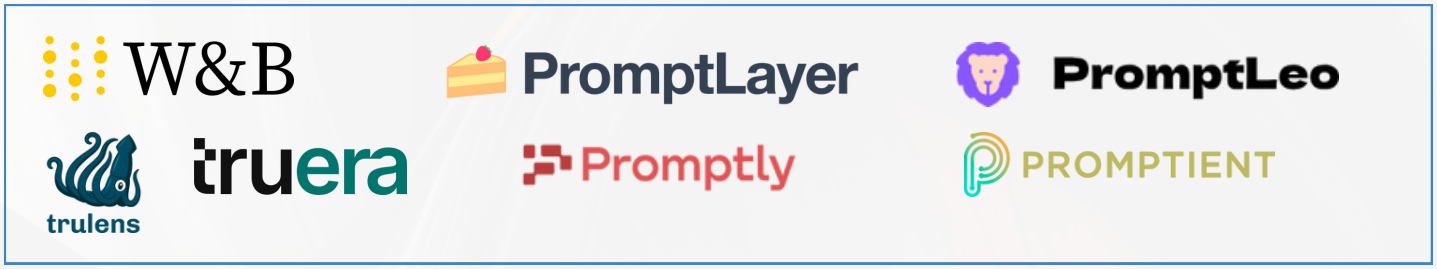


Popular vendors providing access to LLMs (Non Exhaustive)

Note : For Open Source models it is important to check the license type. Some open source models are not available for commercial use

Prompt Layer

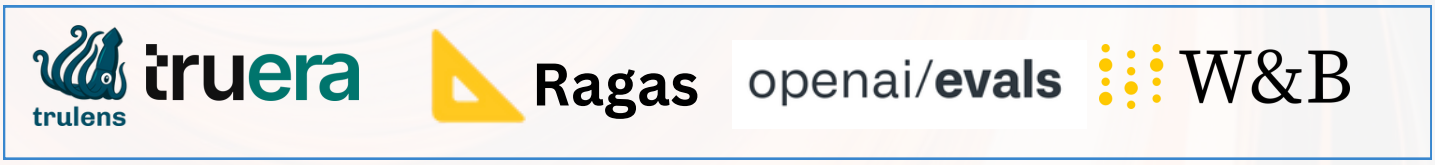
Prompt Engineering is more than writing questions in natural language. There are several prompting techniques and developers need to create prompts tailored to the use cases. This process often involves experimentation: the developer creates a prompt, observes the results and then iterates on the prompts to improve the effectiveness of the app. This requires tracking and collaboration



Popular prompt engineering platforms (Non Exhaustive)

Evaluation

It is easy to build a RAG pipeline but to get it ready for production involves robust evaluation of the performance of the pipeline. For checking hallucinations, relevance and accuracy there are several frameworks and tools that have come up.



Popular RAG evaluation frameworks and tools (Non Exhaustive)

App Orchestration

An RAG application involves interaction of multiple tools and services. To run the RAG pipeline, a solid orchestration framework is required that invokes these different processes.

Popular App orchestration frameworks (Non Exhaustive)

Deployment Layer

Deployment of the RAG application can be done on any of the available cloud providers and platforms. Some important factors to consider while deployment are also -

- Security and Governance
- Logging
- Inference costs and latency

Popular cloud providers and LLMOps platforms (Non Exhaustive)

Application Layer

The application finally needs to be hosted for the intended users or systems to interact with it. You can create your own application layer or use the available platforms.

Popular app hosting platforms (Non Exhaustive)

Monitoring

Deployed application needs to be continuously monitored for both accuracy and relevance as well as cost and latency.

Popular monitoring platforms (Non Exhaustive)

Other Considerations

LLM Cache - To reduce costs by saving responses for popular queries
LLM Guardrails - To add additional layer of scrutiny on generations

Multimodal RAG

Up until now, most AI models have been limited to a single modality (a single type of data like text or images or video). Recently, there has been significant progress in AI models being able to handle multiple modalities (majorly text and images). With the emergence of these Large Multimodal Models (LMMs) a multimodal RAG system becomes possible.

"Generate any type of output from any type of input providing any type of context"

The high-level features of multimodal RAG are -

1. Ability to **query/prompt in one or more modalities** like sending both text and image as input.
2. Ability to **search and retrieve not only text** but also images, tables, audio files related to the query
3. Ability to **generate text, image, video etc.** irrespective of the mode(s) in which the input is provided.

Approaches

Using MultiModal Embeddings

Using LMMs Only

Large MultiModel Models

Google DeepMind Flamingo

salesforce BLIP

Microsoft KOSMOS-1

Tencent Macaw-LLM

OpenAI GPT4

Google Gemini

LlaVA

LAVIN

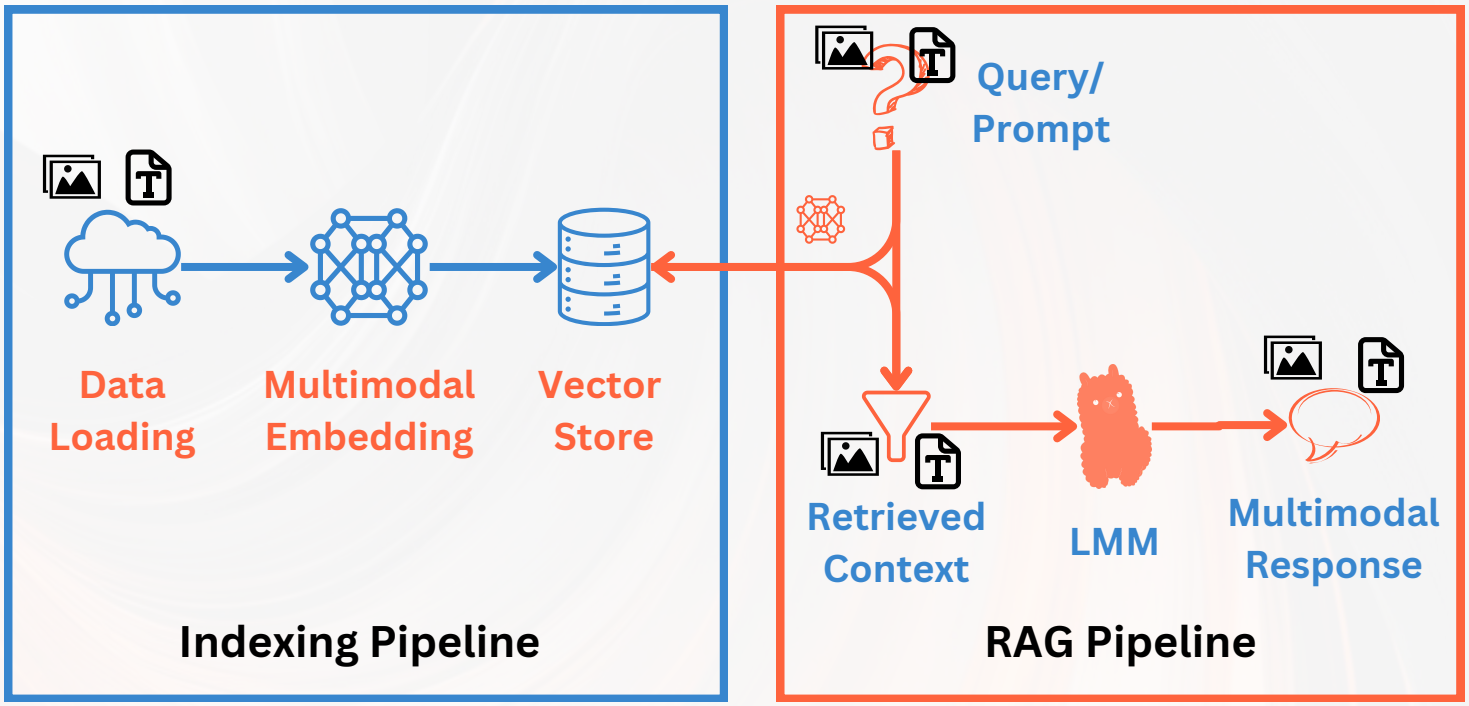
LLaMA - Adapter

FUYU

Multimodal RAG Approaches

Using MultiModal Embeddings

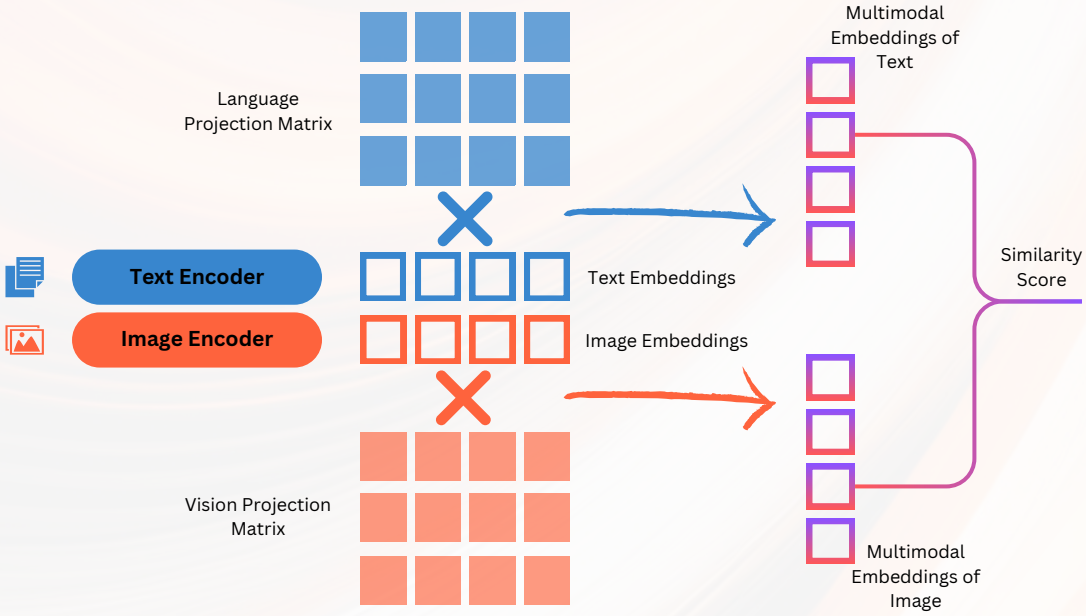
- Multimodal embeddings (like **CLIP**) are used to embed images and text
- User Query is used to retrieve context which can be image and/or text
- The image and/or text context is passed to an LMM with the prompt.
- The LMM generates the final response based on the prompt



Multimodal RAG using Multimodal Embeddings

CLIP : Contrastive Language-Image Pre-training

Mapping data of different modalities into a shared embedding space



OpenAI's CLIP (Contrastive Language-Image Pre-training), maps both images and text into the same semantic embedding space. This allows CLIP to "understand" the relationship between texts and images for powerful applications

CLIP is an example of training multimodal embeddings

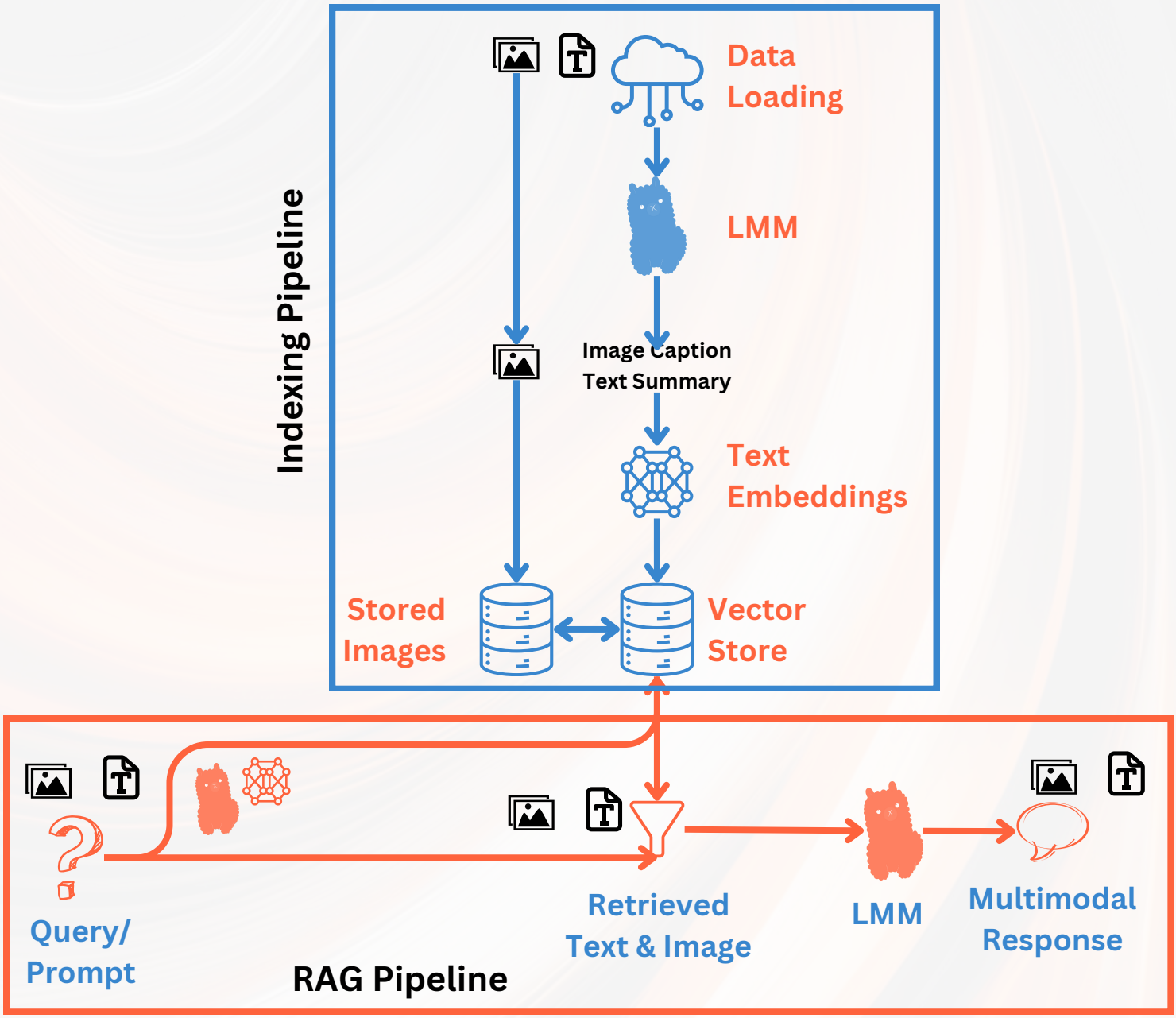
Using LMMs to produce text summaries from images

Indexing

- An LLM is used to generate captions for images in the data
- The image captions and text summaries are stored as text embeddings in a vector database
- A mapping is maintained from the image captions to the image files

Generation

- User enters a query (with text and image)
- Image captions are generated using an LLM and embeddings are generated
- Text summaries and image captions are searched. Images are retrieved based on the relevant image captions.
- Retrieved text summaries, captions and images are passed to the LMM with the prompt. The LMM generates a multimodal response



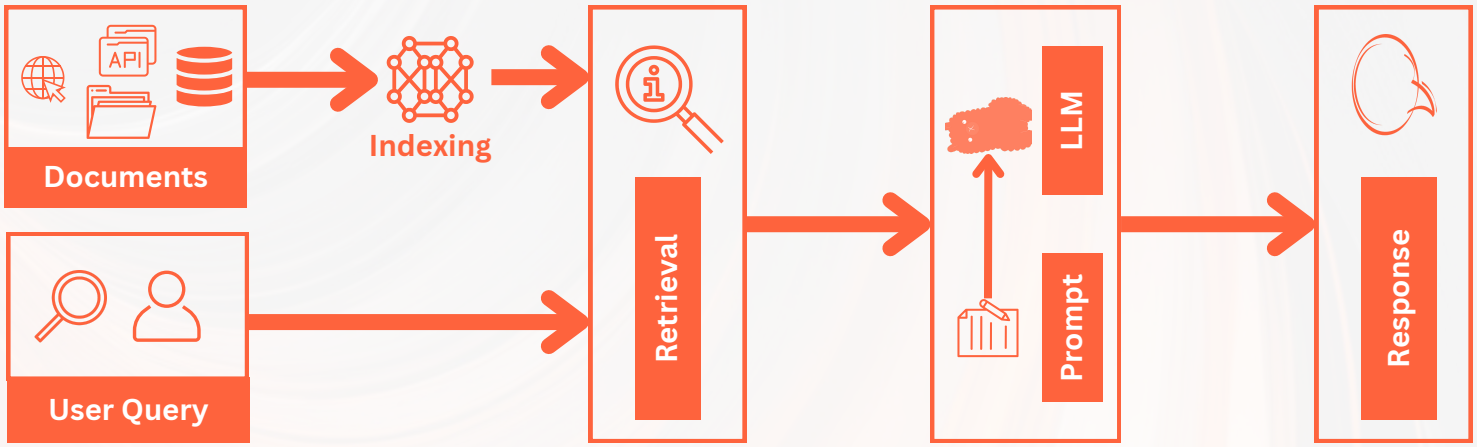
Progression of RAG Systems

Ever since its introduction in mid-2020, RAG approaches have followed a progression aiming to achieve the redressal of the hallucination problem in LLMs

Naive RAG

At its most basic, Retrieval Augmented Generation can be summarized in three steps -

1. **Indexing** of the **documents**
2. **Retrieval** of the context with respect to an input query
3. **Generation** of the **response** using the input query and retrieved context



This basic RAG approach can also be termed "Naive RAG"

Challenges in Naive RAG

Retrieval Quality

- **Low Precision** leading to Hallucinations/Mid-air drops
- **Low Recall** resulting in missing relevant info
- **Outdated information**

Augmentation

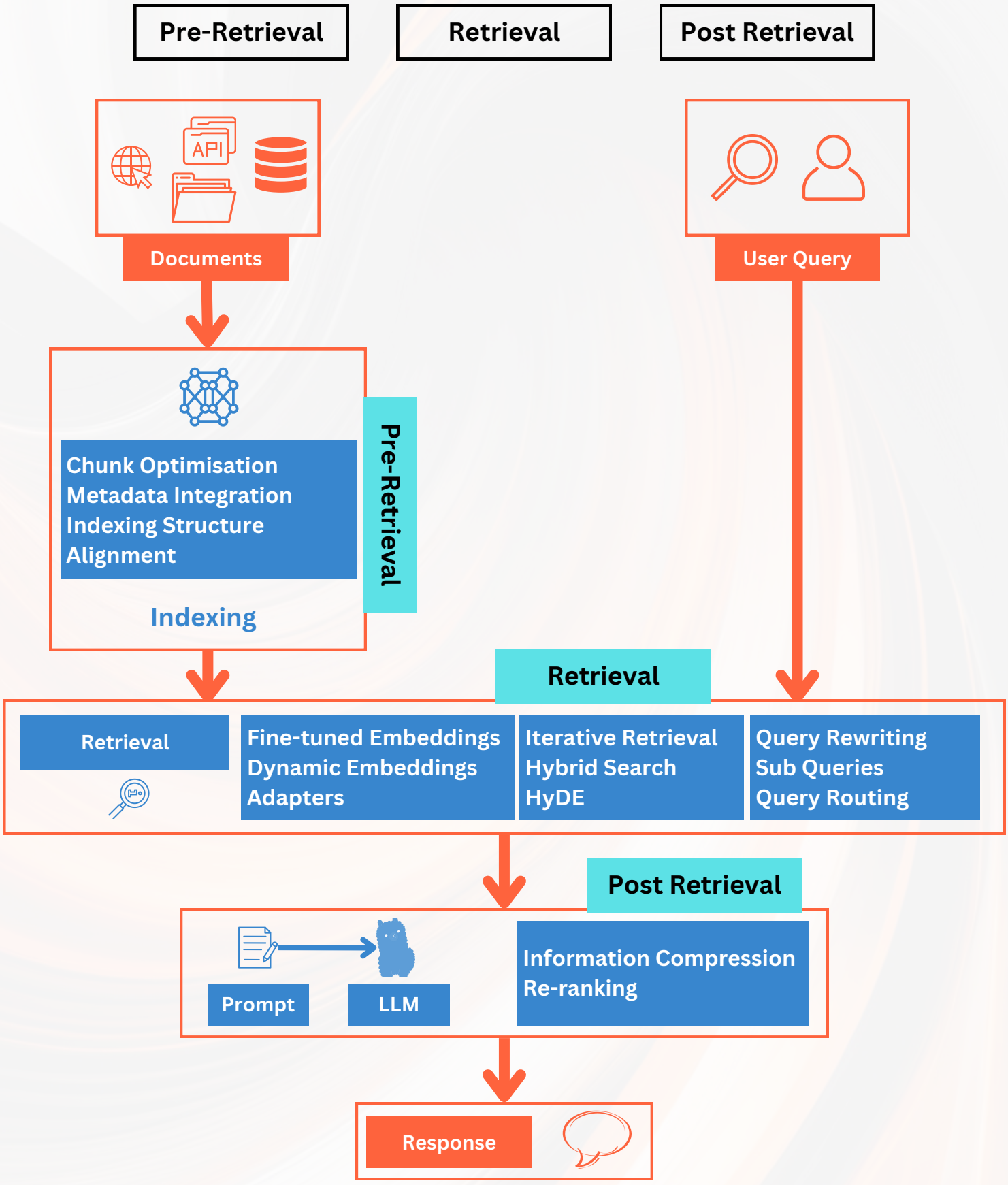
- **Redundancy and Repetition** when multiple retrieved documents have similar information
- **Context Length** challenges

Generation Quality

- Generations are **not grounded** in the context
- Potential of **toxicity and bias** in the response
- **Excessive dependence** on augmented context

Advanced RAG

To address the inefficiencies of the Naive RAG approach, Advanced RAG approaches implement strategies focussed on three processes -



** Indicative, non-exhaustive list*

Advanced RAG Concepts

Pre-retrieval/Retrieval Stage

Chunk Optimization

When managing external documents, it's important to break them into the right-sized chunks for accurate results. The choice of how to do this depends on factors like content type, user queries, and application needs. No one-size-fits-all strategy exists, so flexibility is crucial. Current research explores techniques like sliding windows and "small2big" methods

Metadata Integration

Information like dates, purpose, chapter summaries, etc. can be embedded into chunks. This improves the retriever efficiency by not only searching the documents but also by assessing the similarity to the metadata.

Indexing Structure

Introduction of graph structures can greatly enhance retrieval by leveraging nodes and their relationships. Multi-index paths can be created aimed at increasing efficiency.

Alignment

Understanding complex data, like tables, can be tricky for RAG. One way to improve the indexing is by using counterfactual training, where we create hypothetical (what-if) questions. This increases the alignment and reduces disparity between documents.

Query Rewriting

To bring better alignment between the user query and documents, several rewriting approaches exists. LLMs are sometimes used to create pseudo documents from the query for better matching with existing documents. Sometimes, LLMs perform abstract reasoning. Multi-querying is employed to solve complex user queries.

Hybrid Search Exploration

The RAG system employs different types of searches like keyword, semantic and vector search, depending upon the user query and the type of data available.

Sub Queries

Sub querying involves breaking down a complex query into sub questions for each relevant data source, then gather all the intermediate responses and synthesize a final response.

Query Routing

A query router identifies a downstream task and decides the subsequent action that the RAG system should take. During retrieval, the query router also identifies the most appropriate data source for resolving the query.

Iterative Retrieval

Documents are collected repeatedly based on the query and the generated response to create a more comprehensive knowledge base.

Recursive Retrieval

Recursive retrieval also iteratively retrieves documents. However, it also refines the search queries depending on the results obtained from the previous retrieval. It is like a continuous learning process.

Adaptive Retrieval

Enhance the RAG framework by empowering Language Models (LLMs) to proactively identify the most suitable moments and content for retrieval. This refinement aims to improve the efficiency and relevance of the information obtained, allowing the models to dynamically choose when and what to retrieve, leading to more precise and effective results

Hypothetical Document Embeddings (HyDE)

Using the Language Model (LLM), HyDE forms a hypothetical document (answer) in response to a query, embeds it, and then retrieves real documents similar to this hypothetical one. Instead of relying on embedding similarity based on the query, it emphasizes the similarity between embeddings of different answers.

Fine-tuned Embeddings

This process involves tailoring embedding models to improve retrieval accuracy, particularly in specialized domains dealing with uncommon or evolving terms. The fine-tuning process utilizes training data generated with language models where questions grounded in document chunks are generated.

Post Retrieval Stage

Information Compression

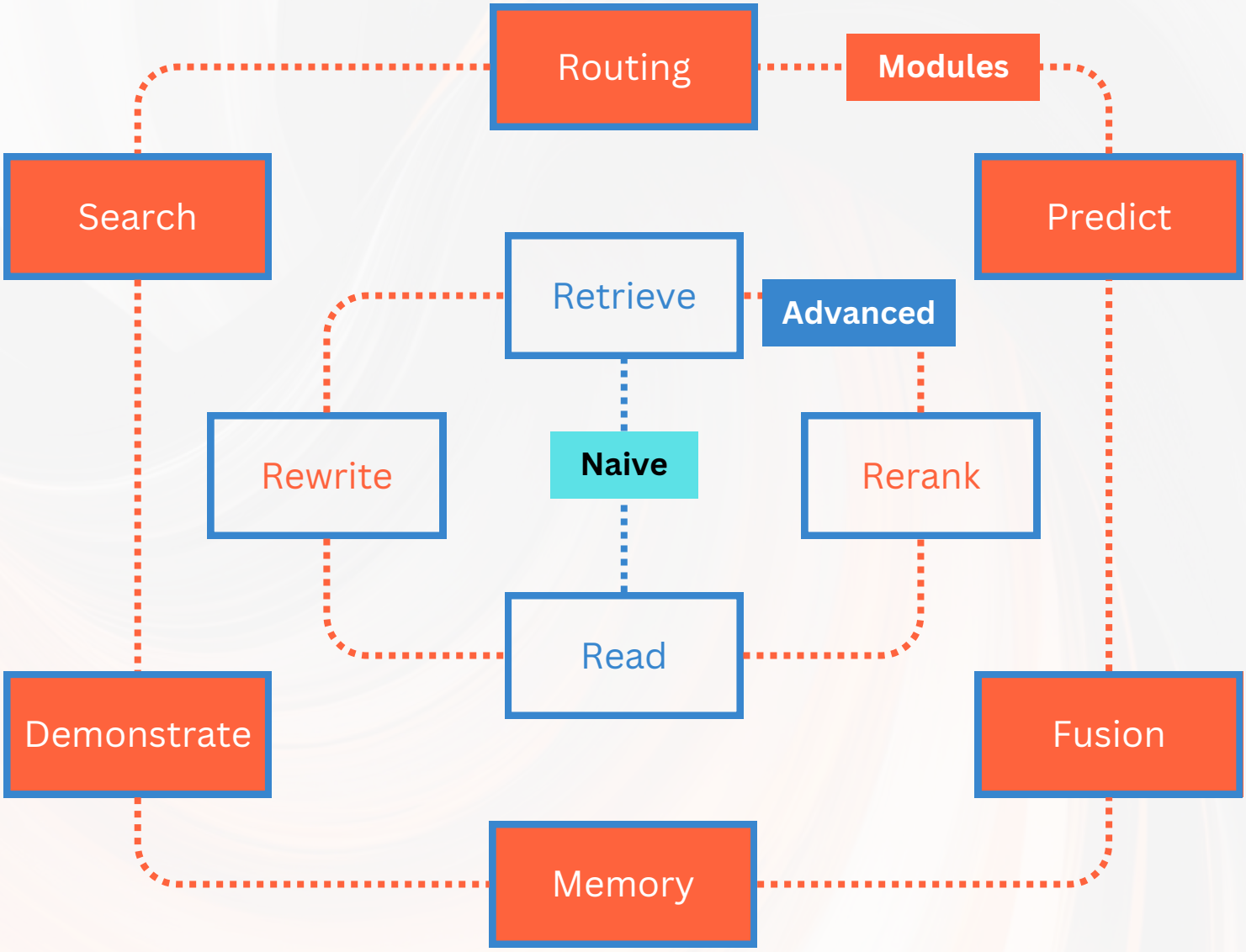
While the retriever is proficient in extracting relevant information from extensive knowledge bases, managing the vast amount of information within retrieval documents poses a challenge. The retrieved information is compressed to extract the most relevant points before passing it to the LLM.

Reranking

The re-ranking model plays a crucial role in optimizing the document set retrieved by the retriever. The main idea is to rearrange document records to prioritize the most relevant ones at the top, effectively managing the total number of documents. This not only resolves challenges related to context window expansion during retrieval but also improves efficiency and responsiveness.

Modular RAG

The SOTA in Retrieval Augmented Generation is a modular approach which allows components like search, memory, and reranking modules to be configured



Naive RAG is essentially a **Retrieve -> Read** approach which focusses on retrieving information and comprehending it.

Advanced RAG is adds to the **Retrieve -> Read** approach by adding it into a **Rewrite** and **Rerank** components to improve relevance and groundedness.

Modular RAG takes everything a notch ahead by providing **flexibility** and adding modules like **Search**, **Routing**, **etc**.

Naive, Advanced & Modular RAGs are not exclusive approaches but a progression. Naive RAG is a special case of Advanced which, in turn, is a special case of Modular RAG

Some RAG Modules

Search

The search module is aimed at performing search on different data sources. It is customised to different data sources and aimed at increasing the source data for better response generation

Memory

This module leverages the parametric memory capabilities of the Language Model (LLM) to guide retrieval. The module may use a retrieval-enhanced generator to create an unbounded memory pool iteratively, combining the "original question" and "dual question." By employing a retrieval-enhanced generative model that improves itself using its own outputs, the text becomes more aligned with the data distribution during the reasoning process.

Fusion

RAG-Fusion improves traditional search systems by overcoming their limitations through a multi-query approach. It expands user queries into multiple diverse perspectives using a Language Model (LLM). This strategy goes beyond capturing explicit information and delves into uncovering deeper, transformative knowledge. The fusion process involves conducting parallel vector searches for both the original and expanded queries, intelligently re-ranking to optimize results, and pairing the best outcomes with new queries.

Extra Generation

Rather than directly fetching information from a data source, this module employs the Language Model (LLM) to generate the required context. The content produced by the LLM is more likely to contain pertinent information, addressing issues related to repetition and irrelevant details in the retrieved content.

Task Adaptable Module

This module makes RAG adaptable to various downstream tasks allowing the development of task-specific end-to-end retrievers with minimal examples, demonstrating flexibility in handling different tasks.

Acknowledgements

Retrieval Augmented Generation continues to be a pivotal approach for any Generative AI led application and it is only going to grow. There are several individuals and organisations that have provided learning resources and made understanding RAG fun.

I'd like to thank -

- My team at **Yarnit.app** for taking a bet on RAG and helping me explore and execute RAG pipelines for content generation
- **Andrew Ng** and the good folks at **deeplearning.ai** for their short courses allowing everyone access to generative AI
- **OpenAI** and **HuggingFace** for all that they do
- **Harrison Chase** and all the folks at **LangChain** for not only building the framework but also making it easy to execute
- **Jerry Liu** and others at **LlamaIndex** for their perspectives and tutorials on RAG
- **TruEra** for demystifying observability and the tech stack for LLMOps
- **PineCone** for their amazing documentation and the learning center
- The team at **Exploding Gradients** for creating **Ragas** and explaining RAG evaluation in detail
- **TruLens** for their triad of RAG evaluations
- **Aman Chadha** for his curation of all thing AI, ML and Data Science
- Above all, to my **colleagues and friends**, who endeavour to learn, discover and apply technology everyday in their effort to make the world a better place.

With lots of love,

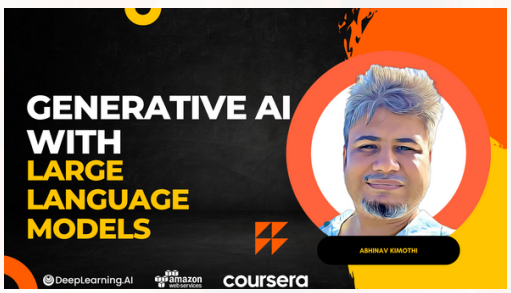
Abhinav

I talk about :

#AI #MachineLearning #DataScience #GenerativeAI #Analytics #LLMs #Technology #RAG #EthicalAI

If you like'd what you read

Detailed Notes from **Generative AI with Large Language Models** Course by **Deeplearning.ai** and **AWS**.



Resources

Official Documentations

LangChain
Python Documentation

LlamaIndex
Python Documentation

Pinecone
Learning Center

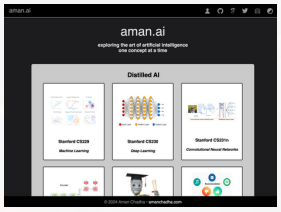
trulens
Documentation

Ragas
Documentation

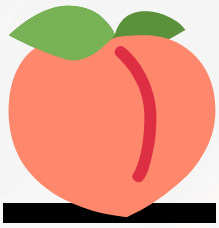
Hugging Face
Documentation

OpenAI
Documentation

Thought Leaders and Influencers



Aman Chadha's Blog



Lillian Weng's Log

Leonie Monigatti's Blogroll

Chip Huyen Blogs

Research Papers

Retrieval-Augmented Generation for Large Language Models: A Survey (Gao, et al, 2023)

Retrieval-Augmented Multimodal Language Modeling (Yasunaga, et al, 2023)

KG-Augmented Language Models for Knowledge-Grounded Dialogue (Kang, et al, 2023)

Learning Resources and Tutorials

DeepLearning.AI
Short 1-hour Courses

LangChain
Python Cookbook

LlamaIndex
Tutorials & Webinars

Hello!
I'm Abhinav...
A data science and AI professional with over 15 years in the industry. Passionate about AI advancements, I constantly explore emerging technologies to push the boundaries and create positive impacts in the world. Let's build the future, together!

Please share your feedback on these notes with me

LinkedIn Github Medium Insta email X Linktree Gumroad

Talk to me

Checkout Yarnit Magic

Newsletter

$$ Contribute $$

Buy me a coffee

Subscribe

Don't miss a post!
Get an email notification whenever I publish!

Follow on LinkedIn

Retrieval Augmented Generation

Download Your Free Copy of Complete Notes from Gumroad

https://abhinavkimothi.gumroad.com/l/RAG

- What is Retrieval Augmented Generation?
- How does RAG help?
- What are some popular RAG use cases?
- What does the RAG Architecture look like?
- What are Embeddings?
- What are Vector Stores?
- What are the best retrieval strategies?
- How to Evaluate RAG outputs?
- RAG vs Finetuning - What is better?
- How does the evolving LLMOps Stack look like?
- What is Multimodal RAG?
- What is Naive, Advanced and Modular RAG?

DOWNLOAD