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

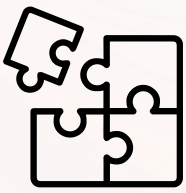
We covered the indexing pipeline in its entirety. A quick recap -

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

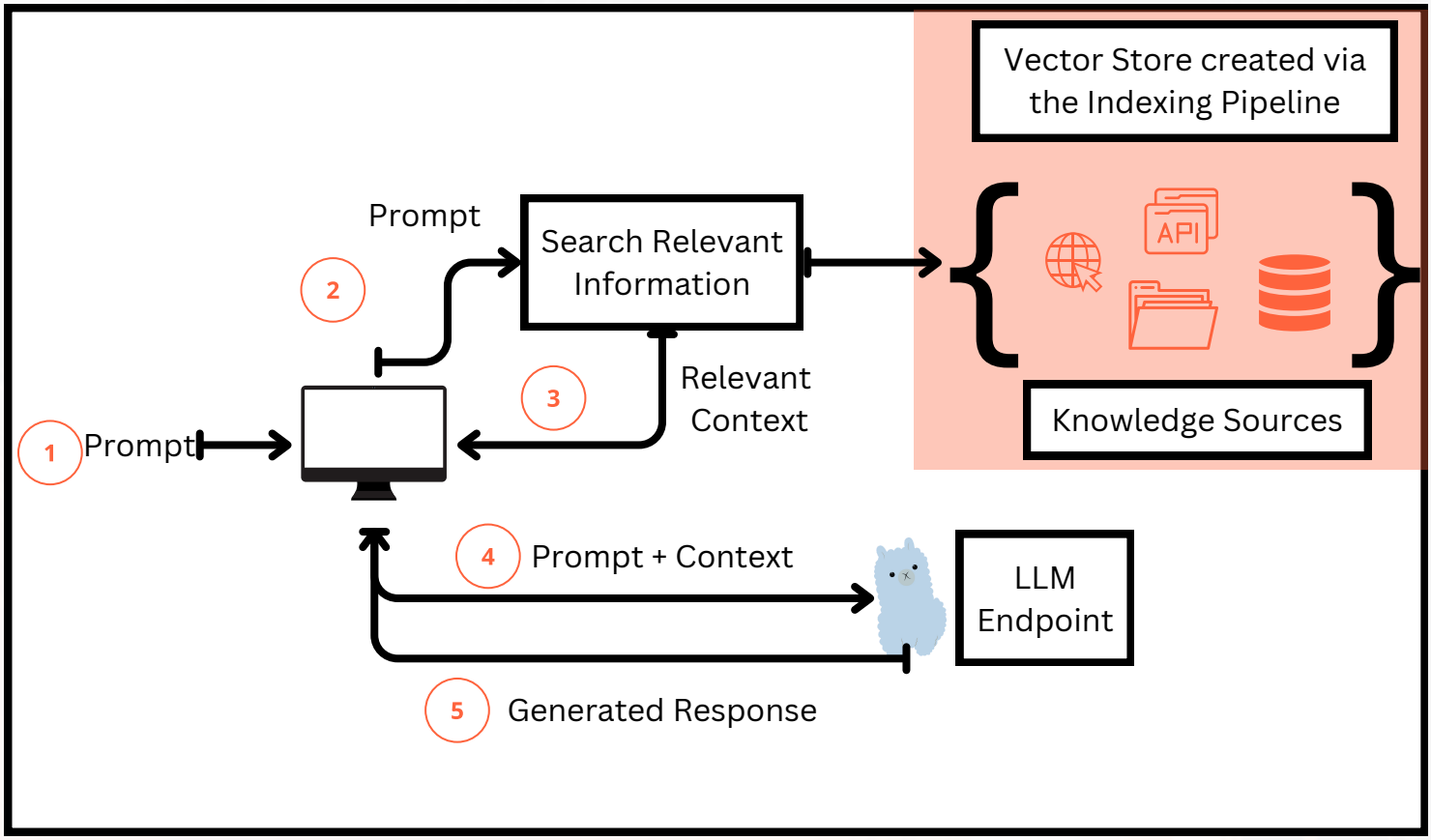
Storing

- 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

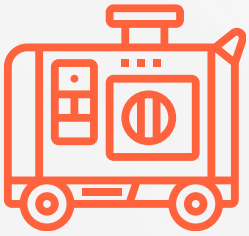
The three main steps in a RAG pipeline are

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

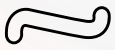
On the fly

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

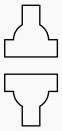
Maximum Marginal Relevance

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

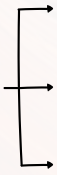


Contextual compression

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

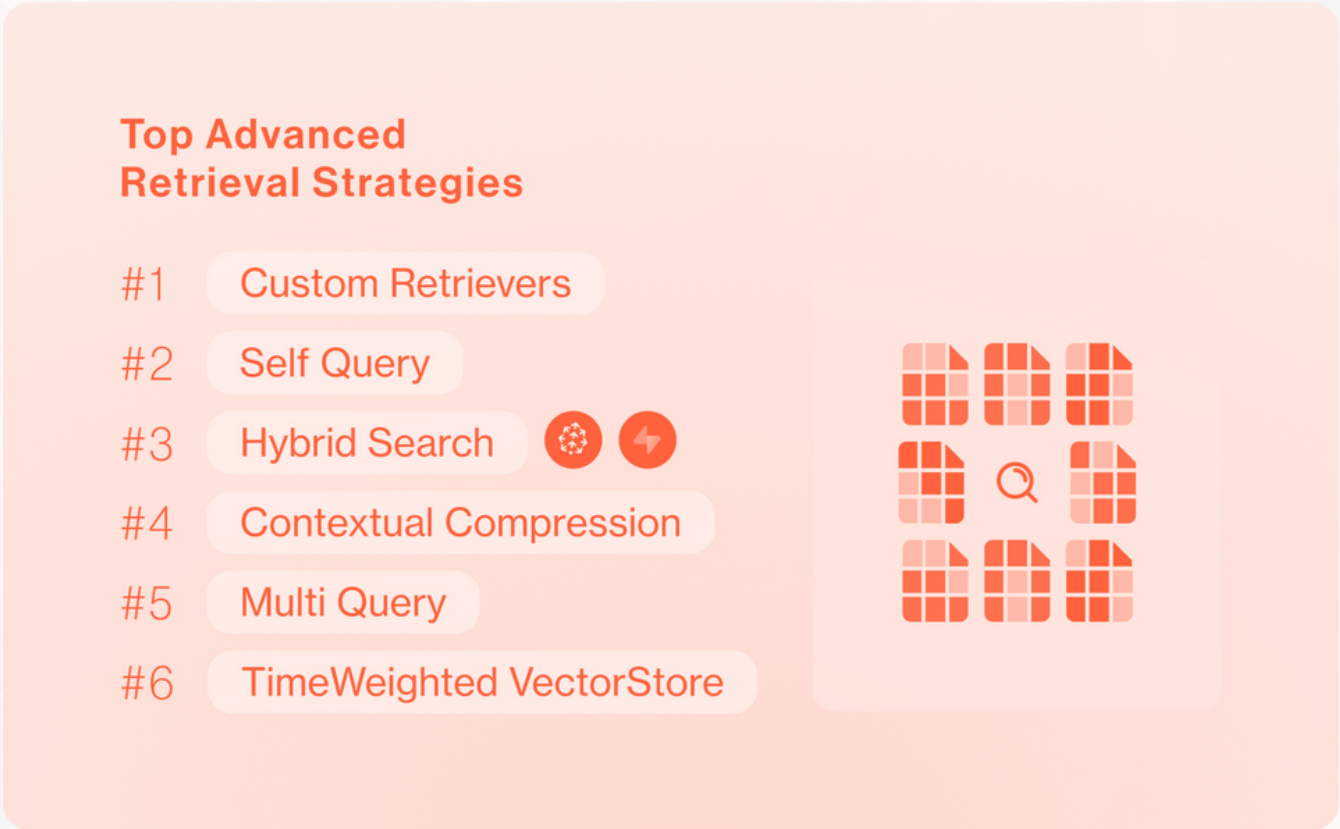
Time-weighted Retrieval

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")
```

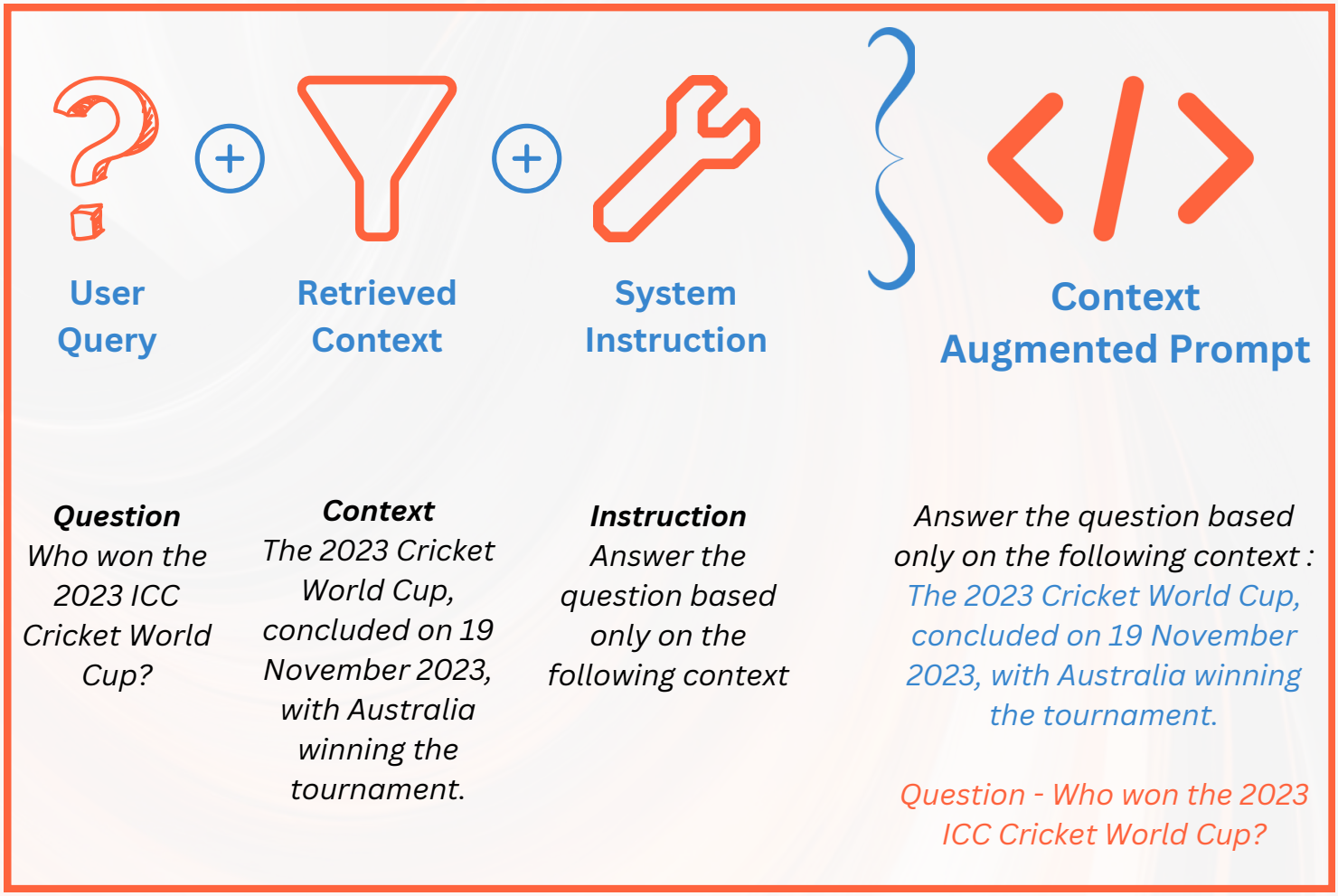
Example : Contextual compression

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

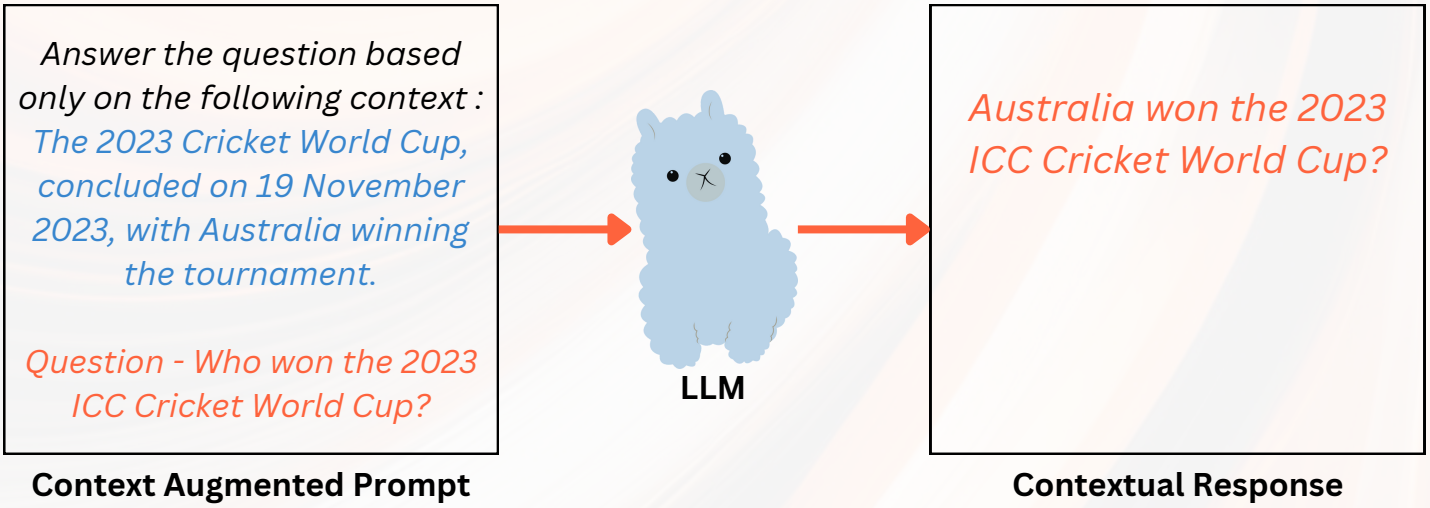
```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.llms import OpenAI
from langchain.retrievers import ContextualCompressionRetriever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

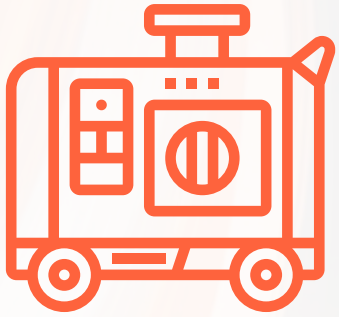
This can be because -

- 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
- Corresponding **Response** or **Answer** from LLM
- **Retrieved Context** for each prompt
- **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**?

Evaluation Metrics

1 Faithfulness

Faithfulness is the measure of the extent to which the response is factually grounded in the retrieved context

Problem addressed : The LLM, despite being provided the context, does not consider it

or

Is the response grounded in the provided context?

Evaluated Process : Generation

Any measure of retrieval accuracy is out of scope

Score Range : (0,1) Higher score is better

Methodology

Faithfulness identifies the number of "claims" made in the response and calculates the proportion of those "claims" present in the context.

$$\textbf{Faithfulness} = \frac{\textit{Number of generated claims present in the context}}{\textit{Total number of claims made in the generated response}}$$

Illustrative Example

Query : Who won the 2023 ODI Cricket World Cup and when?

Context : The 2023 ODI Cricket World Cup concluded on 19 November 2023, with Australia winning the tournament.

Response 1 : High Faithfulness	**Response 2 : Low Faithfulness**
[Australia] won on [19 November 2023]	*[Australia] won on [15 October 2023]*

Evaluation Metrics

2 Answer Relevance

Answer Relevance is the measure of the extent to which the response is relevant to the query or the prompt

Problem addressed :The LLM instead of answering the query responds with irrelevant information

or

Is the response relevant to the query?

Evaluated Process : Generation

Any measure of retrieval accuracy is out of scope

Score Range : (0,1) Higher score is better

Methodology

For this metric, a response is generated for the initial query or prompt
To compute the score, the LLM is then prompted to generate questions for the generated response several times. The mean cosine similarity between these questions and the original one is then calculated. The concept is that if the answer correctly addresses the initial question, the LLM should generate questions from it that match the original question.

$$\text{Answer Relevance} = \textit{Avg}\left(\textit{Sc}(\textit{Initial Query}, \textit{LLM generated Query}[i])\right)$$

Illustrative Example

Query : Who won the 2023 ODI Cricket World Cup and when?

Response 1 : High Answer Relevance	**Response 2 : Low Answer Relevance**
India won on 19 November 2023	*Cricket world cup is held once every four years*

Note

Answer Relevance is **not a measure of truthfulness** but only of relevance. The response may or may not be factually accurate but may be relevant.

Evaluation Metrics

3 Context Relevance

Context Relevance is the measure of the extent to which the retrieved context is relevant to the query or the prompt

Problem addressed :The retriever fails to retrieve relevant context

or

Is the retrieved context relevant to the query?

Evaluated Process : Retrieval

Indifferent to the final generated response

Score Range : (0,1) Higher score is better

Methodology

The retrieved context should contain information only relevant to the query or the prompt. For context relevance, a metric 'S' is estimated. 'S' is the number of sentences in the retrieved context that are relevant for responding to the query or the prompt.

$$\text{Context Relevance} = \frac{S}{\text{Total number of sentences in the retrieved context}}$$

(number of relevant sentences from the context)

Illustrative Example

Query : Who won the 2023 ODI Cricket World Cup and when?

Context 1 : High Context Relevance	Context 2 : Low Context Relevance
The 2023 Cricket World Cup, concluded on 19 November 2023, with Australia winning the tournament. The tournament took place in ten different stadiums, in ten cities across the country. The final took place between India and Australia at Narendra Modi Stadium	*The 2023 Cricket World Cup was the 13th edition of the Cricket World Cup. It was the first Cricket World Cup which India hosted solely. The tournament took place in ten different stadiums.In the first semi-final India beat New Zealand, and in the second semi-final Australia beat South Africa.*

Evaluation Metrics

Ground Truth

Ground truth is information that is known to be real or true. In RAG, or Generative AI domain in general, Ground Truth is a prepared set of **Prompt-Response examples**. It is akin to *labelled data* in Supervised Learning parlance.

Calculation of certain metrics necessitates the availability of Ground Truth data

4 Context Recall

Context recall measures the extent to which the retrieved context aligns with the "provided" answer or Ground Truth

Problem addressed :The retriever fails to retrieve accurate context

or

Is the retrieved context good enough to provide the response?

Evaluated Process : Retrieval

Indifferent to the final generated response

Score Range : (0,1) Higher score is better

Methodology

To estimate context recall from the ground truth answer, each sentence in the ground truth answer is analyzed to determine whether it can be attributed to the retrieved context or not. Ideally, all sentences in the ground truth answer should be attributable to the retrieved context.

$$\text{Context Recall} = \frac{\textit{Number of Ground Truth sentences in the context}}{\textit{Total number of sentences in the Ground Truth}}$$

Illustrative Example

Query : Who won the 2023 ODI Cricket World Cup and when?

Ground Truth : Australia won the world cup on 19 November, 2023.

Context 1 : High Context Recall	Context 2 : Low Context Recall
The 2023 Cricket World Cup, concluded on 19 November 2023, with Australia winning the tournament.	*The 2023 Cricket World Cup was the 13th edition of the Cricket World Cup. It was the first Cricket World Cup which India hosted solely.*

Evaluation Metrics

5 Context Precision

Context Precision is a metric that evaluates whether all of the ground-truth relevant items present in the contexts are ranked higher or not.

Problem addressed :The retriever fails to rank retrieve context correctly

or

Is the higher ranked retrieved context better to provide the response?

Evaluated Process : Retrieval

Indifferent to the final generated response

Score Range : (0,1) Higher score is better

Methodology

Context Precision is a metric that evaluates whether all of the ground-truth relevant items present in the all retrieved context documents are ranked higher or not. Ideally all the relevant chunks must appear at the top

$$\text{Context Precision @ k} = \frac{\text{Sum(Precision@k)}}{\text{Total number of relevant documents in the top results}}$$

$$\text{Precision @ k} = \frac{\text{True Positives @ k}}{\text{(True Positives @ k + False Positives @ k)}}$$

Precision @ k

Precision@k is a metric used in information retrieval and recommendation systems to evaluate the accuracy of the top k items retrieved or recommended. It measures the proportion of relevant items among the top k items.

Evaluation Metrics

6 Answer semantic similarity

Answer semantic similarity evaluates whether the generated response is similar to the “provided” response or Ground Truth.

Problem addressed : The generated response is incorrect

or

Does the pipeline generate the right response?

Evaluated Process : Retrieval & Generation

Score Range : (0,1) Higher score is better

Methodology

Answer semantic similarity score is calculated by measuring the semantic similarity between the generated response and the ground truth response.

Answer Semantic Similarity = *Similarity (Generated Response, Ground Truth Response)*

7 Answer Correctness

Answer correctness evaluates whether the generated response is semantically and factually similar to the “provided” response or Ground Truth.

Problem addressed : The generated response is incorrect

or

Does the pipeline generate the right response?

Evaluated Process : Retrieval & Generation

Score Range : (0,1) Higher score is better

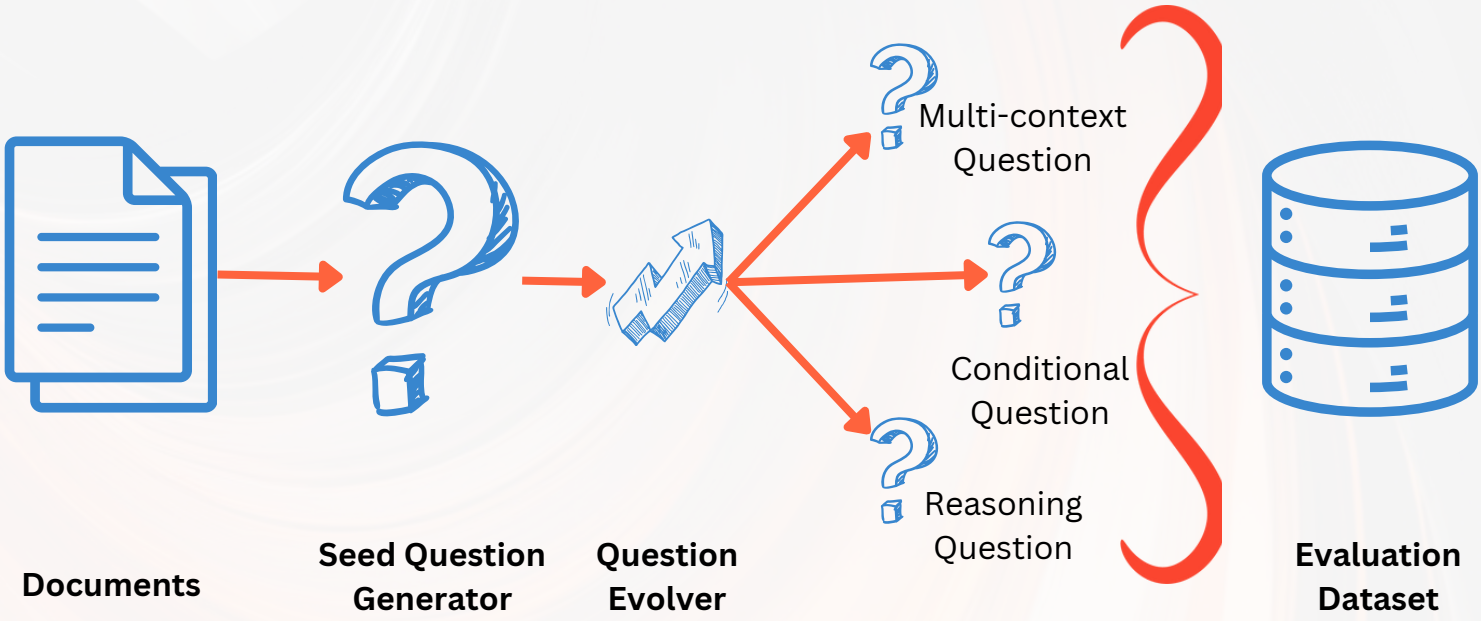
Methodology

Answer correctness score is calculated by measuring the semantic and the factual similarity between the generated response and the ground truth response.

Synthetic Test Data Generation

Generating hundreds of QA (Question-Context-Answer) samples from documents manually can be a time-consuming and labor-intensive task. Moreover, questions created by humans may face challenges in achieving the necessary level of complexity for a comprehensive evaluation, potentially affecting the overall quality of the assessment.

Synthetic Data Generation uses Large Language Models to generate a variety of Questions/Prompts and Responses/Answers from the Documents (Context). It can greatly reduce developer time.



Synthetic Data Generation Pipeline

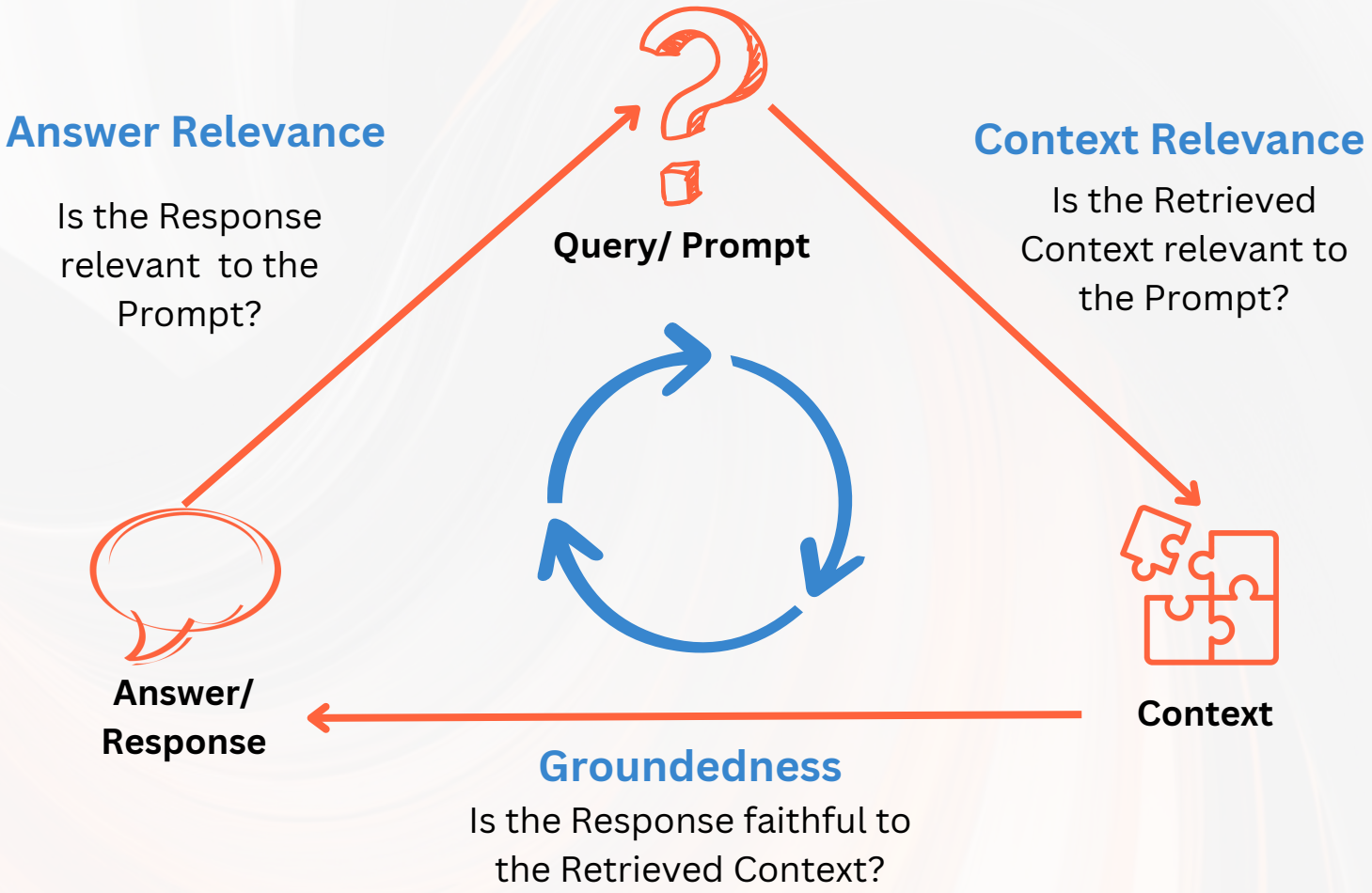
	question	context	answer	question_type	episode_done
0	What technique improves the performance of lar...	- "We explore how generating a chain of though...	The technique that improves the performance of...	simple	True
1	What phenomenon is discussed in the paper rega...	- This paper instead discusses an unpredictabl...	The phenomenon discussed in the paper is the e...	reasoning	True
2	What is the purpose of chain-of-thought (CoT) ...	- Providing these steps for prompting demonstr...	The purpose of chain-of-thought (CoT) promptin...	simple	True
3	What is the performance of the largest fine-tu...	On the MathQA-Python dataset, the largest fine...	The performance of the largest fine-tuned mode...	simple	True
4	What is the accuracy increase of Zero-shot-CoT...	Experimental results demonstrate that our Zero...	The accuracy increase of Zero-shot-CoT on Mult...	reasoning	True

Synthetic Data Generated Using Ragas

Ragas Documentation

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) ✗

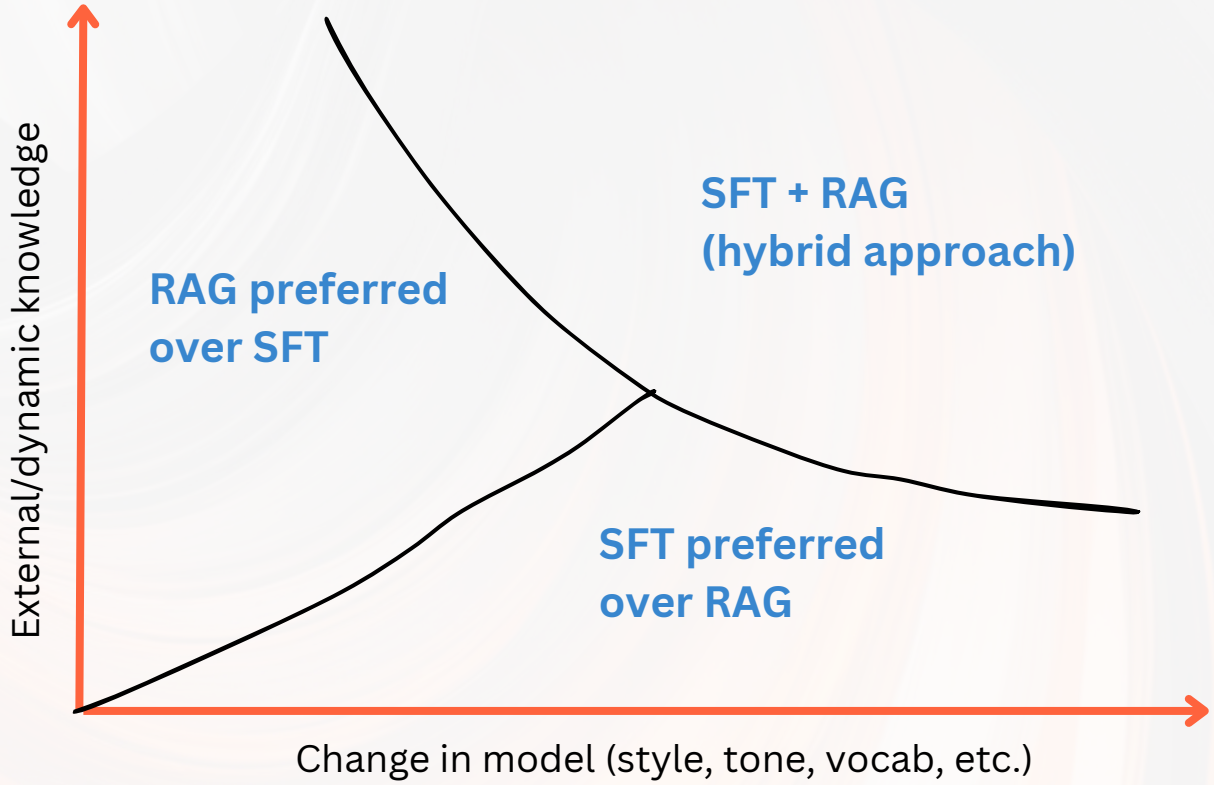
Important Use Case Considerations

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

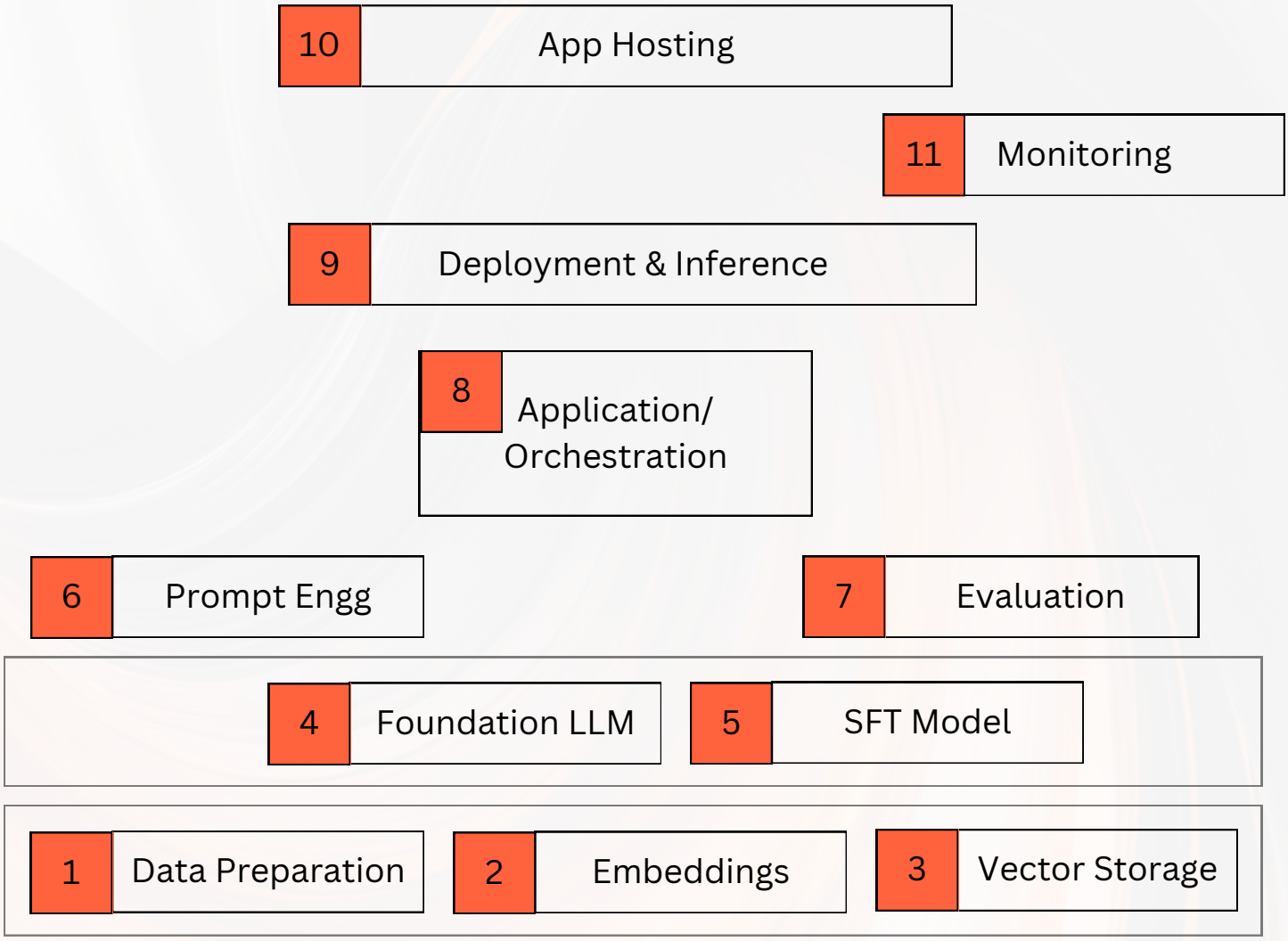
Both RAG and SFT warrant upfront investment. Training cost for SFT can vary depending on the technique and the choice of foundation model. Setting up the knowledge base and integration can be costly for RAG

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**

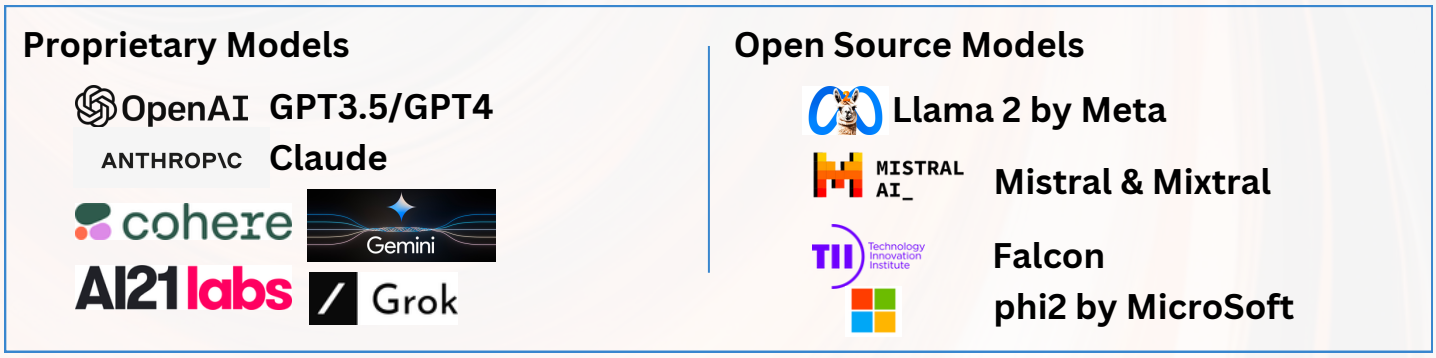
Popular Data Layer Vendors (Non Exhaustive)

Model Layer

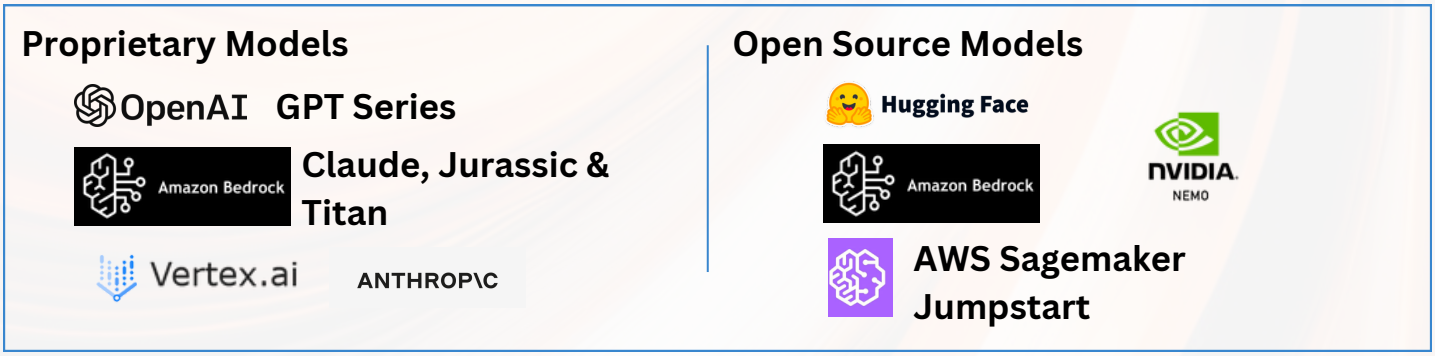
2023 can be considered a year of LLM wars. Almost every other week in the second half of the year a new model was released. Like there is no RAG without data, there is no RAG without an LLM. There are four broad categories of LLMs that can be a part of a RAG application

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