What is Retrieval Augmented Generation(RAG) and how does it work?

Kerem Aydın

·

Subscribe

7 min read

·

Jul 25, 2024

36

2

In this article, we will delve into what retrieval augmented generation is, how does it work, why do we need it and how do we use it.

Retrieval Augmented Generation(created by Bing AI)

With the development of ChatGPT, the attention for large language models(LLMs) skyrocketed. The trained model was able to generate coherent answers to any question you could ask and chat with you on any given topic. At first, it seemed like a silver bullet to any natural language task there is. However, in a short time people realized some of the weaknesses of LLMs:

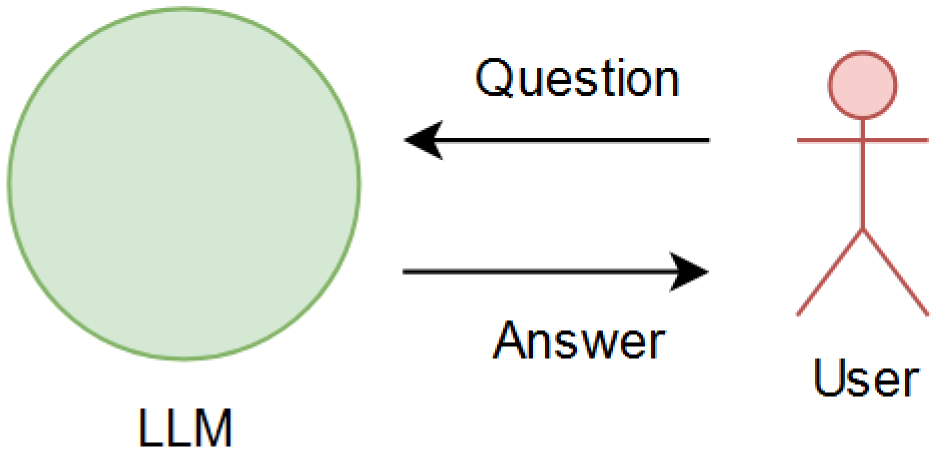
- LLMs do not really know what they know or do not know and are keen to answer questions with coherent text that is not even true

- LLMs may present out of date information since they only know what they were trained on
- LLMs can create a response from non-reliable sources since they do not give any references to the original text

Retrieval Augmented Generation (RAG) is an effective strategy for addressing the limitations of large language models. Retrieval-augmented generation (RAG) is a technique for enhancing the accuracy and reliability of generative AI models with facts fetched from external sources.

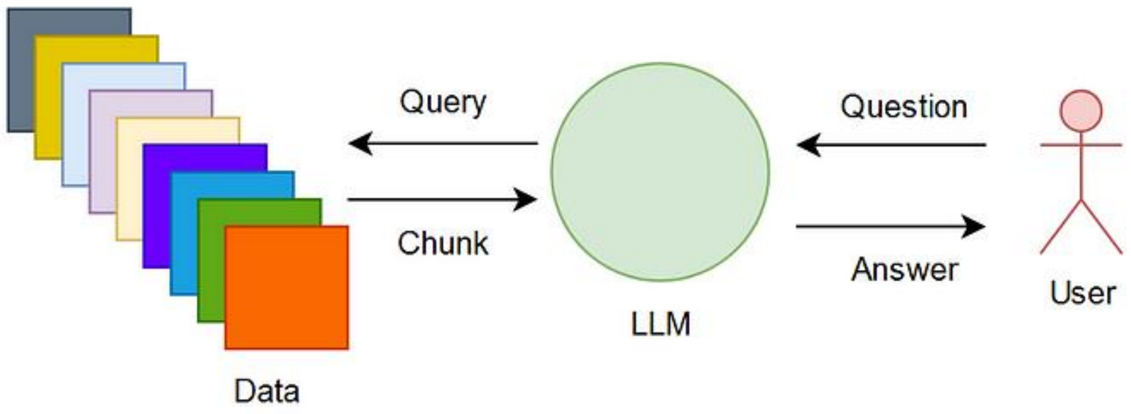
How does it work?

Without Retrieval Augmented Generation (RAG), large language models (LLMs) generate answers to user questions solely based on their trained neural networks. Therefore the model can hallucinate and generate coherent inaccurate results even though it does not know the answer to the question since it can only generate an accurate response if the information had been thought to the model in the training phase.



LLMs without RAG

On the other hand, RAG retrieves the most relevant chunks to the user question from a stored database and LLM generates an answer from the combination of query and the most relevant chunks.



LLMs with RAG

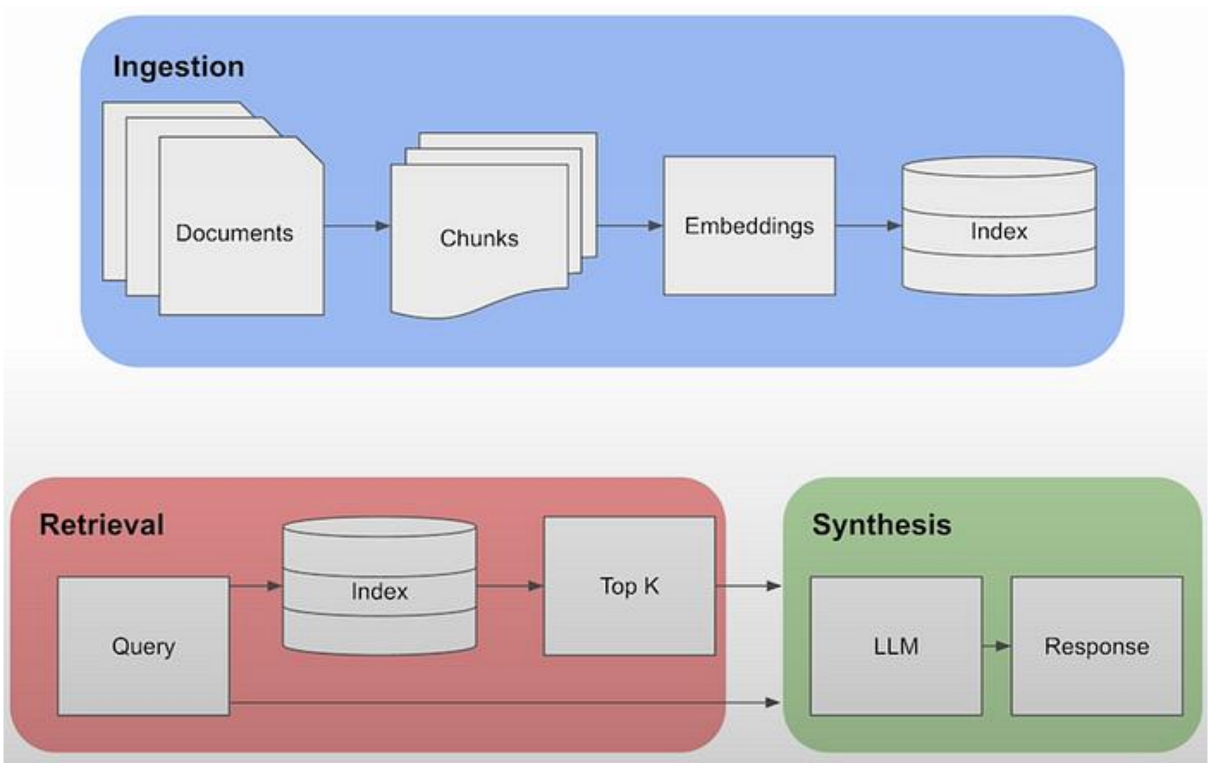
With this technique, we can store relevant data and the LLM can both generate an accurate result and give reference to the original source since it is given in the prompt with the question.

Without RAG, fine-tuning is the only solution to our problem. Fine-tuning is the further training of the model by only updating the last layers of a model with hundred or thousands of training samples. RAG has many superior properties to the fine-tuning:

- RAG does not need any training, therefore it is more computationally and timely efficient
- RAG does not need any preprocessing unlike the training data needed for fine-tuning
- RAG can make the model reference to the source data while generating the answer

A basic RAG pipeline

A basic RAG pipeline consists of 3 components: ingestion, retrieval and synthesis.



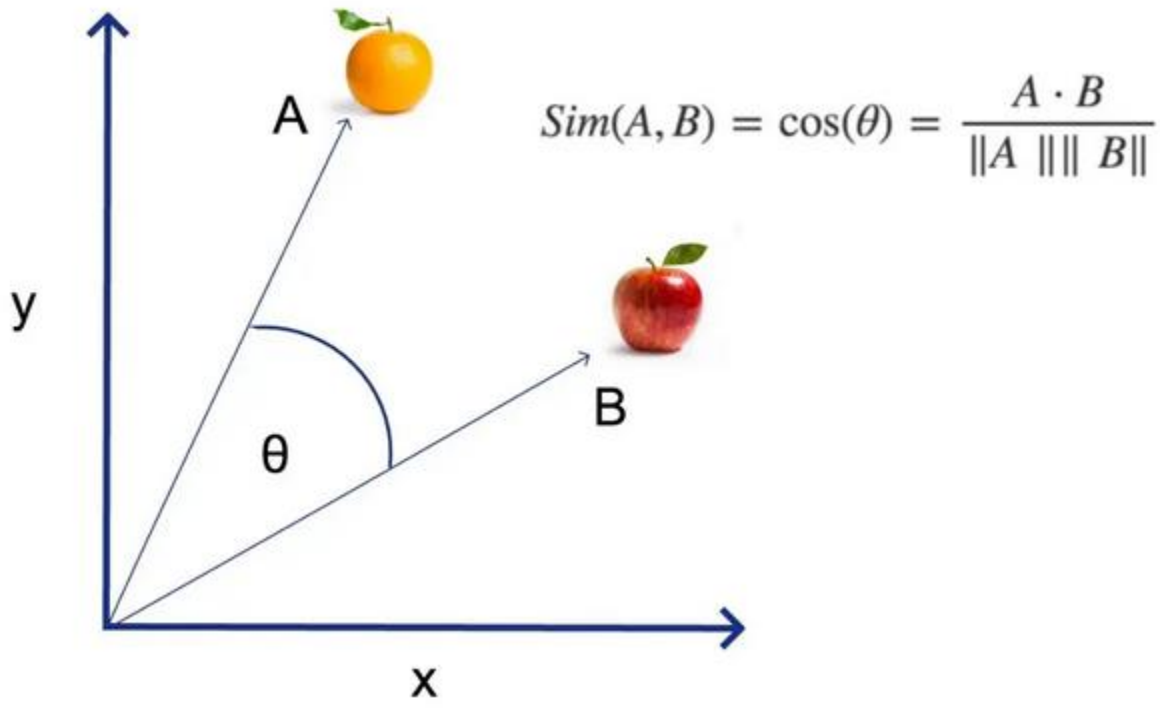
Basic RAG Pipeline[1]

1. Ingestion

In ingestion period, simply we store the embeddings of documents in an index. First we load the documents and then split the documents into chunks which are small sections of text. And then we use an embedding model to convert these chunks into embeddings to represent the meaning of the chunk in a numerical form. After all these, we store the embeddings in an index.

2. Retrieval

In retrieval phase, the query is passed from user to the model and then the query is converted to a vector by the same embedding model in the ingestion period since both query and the chunks can be in the same dimensions. Because after this we use similarity search on index to retrieve most similar chunks to the query. The most common similarity calculation method is the cosine similarity. It is the cosine of the angle between vectors which is calculated through the dot product of the embeddings and divided by the product of their lengths. If the angle between two vectors turns out to be small, that means the two vectors are similar to each other and vice versa.

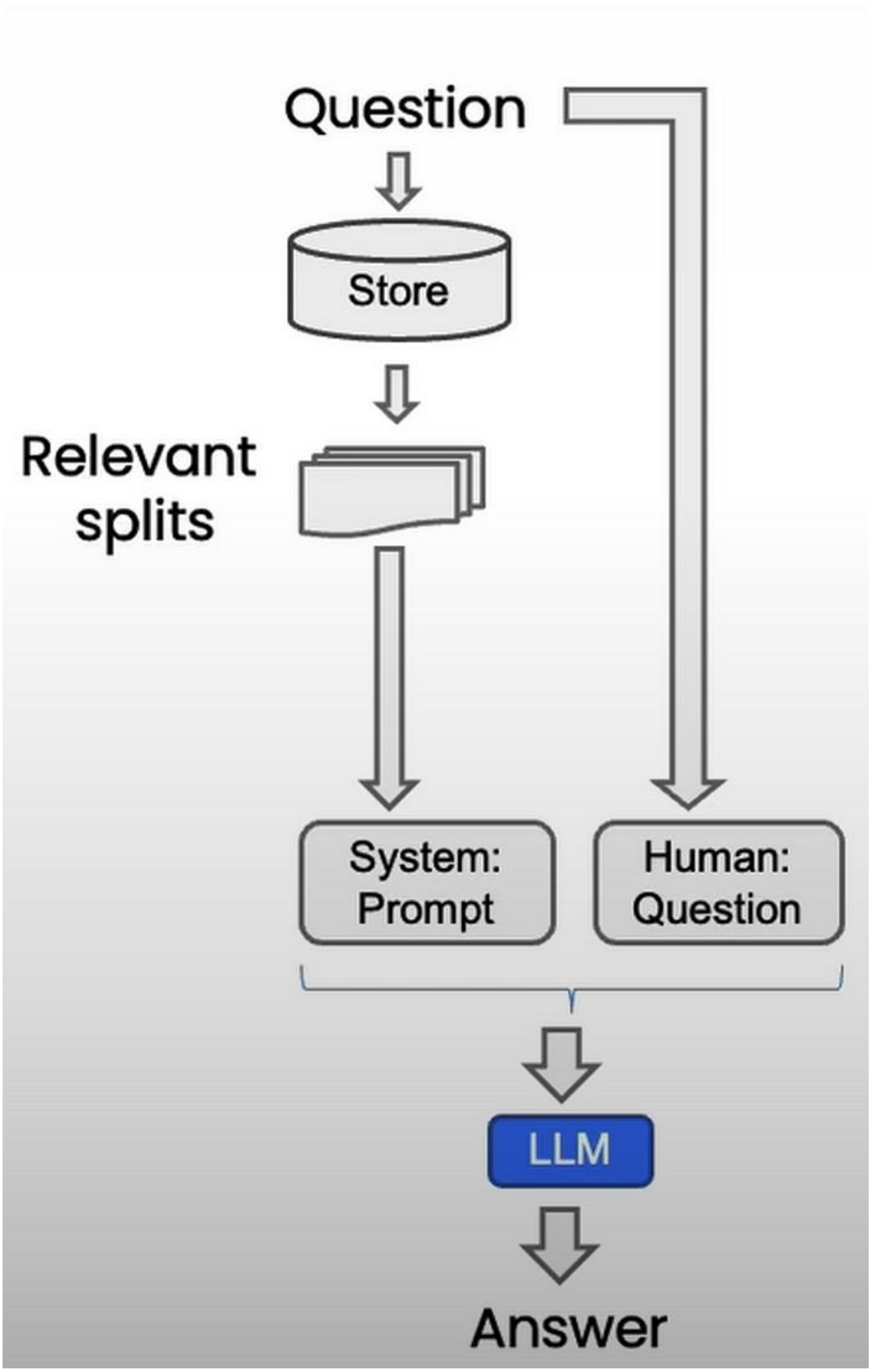


cosine similarity of two vectors called A and B[2]

In theory the most similar chunk in terms of meaning to the query will contain the answer to the question.

3. Synthesis

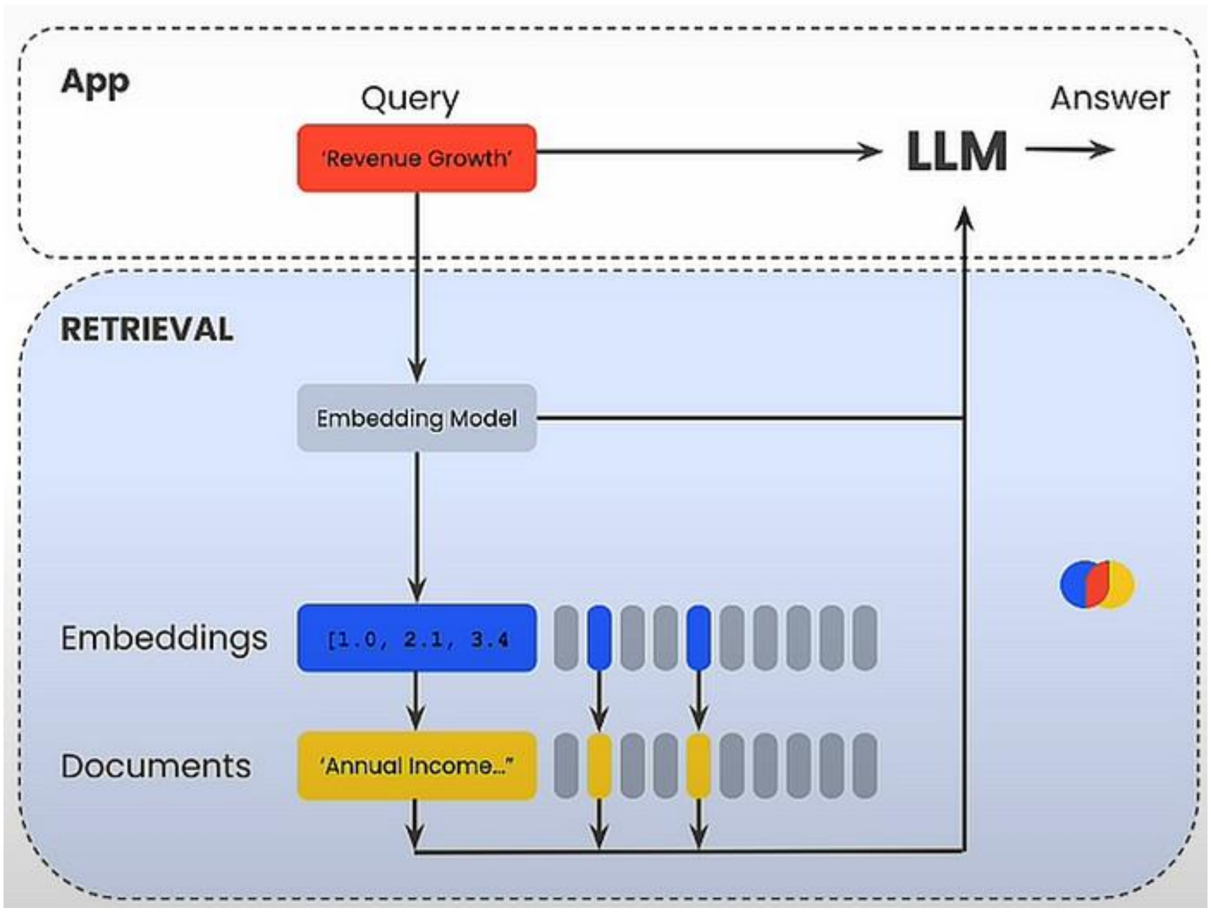
At last, we combine the retrieved chunks and query and then lead it to the LLM. The LLM generates an answer using both the chunks and the query.


Question
Store
Relevant splits
System: Prompt
Human: Question
LLM
Answer

RAG pipeline[3]

How do we use it?

There are many libraries that are designed to implement RAG in a question-answering model pipeline. Chroma[4] is one of the most popular open-source AI application database used for embeddings, vector search, document storage and retrieval. A simple example consist of these stages:



RAG pipeline using Chroma[5]

1. Loading the documents

In this case, we are going to load pdf documents as data in our RAG application.

from pypdf import PdfReader

```
reader = PdfReader("microsoft_annual_report_2022.pdf")
pdf_texts = [p.extract_text().strip() for p in reader.pages]
```

2. Splitting the documents into chunks

Langchain has built-in splitter functions that can separate huge text into smaller chunks in a logical way. Just like in the example below, we can indicate the separator beforehand.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter,
SentenceTransformersTokenTextSplitter

character_splitter = RecursiveCharacterTextSplitter(
   separators=["\n\n", "\n", ". ", " ", ""],
   chunk_size=1000,
   chunk_overlap=0
)
character_split_texts =
character_splitter.split_text('\n\n'.join(pdf_texts))
```

3. Token splitter

If we prefer splitting the text according to tokens rather than characters and we usually do since it is much more effective regarding to understand the meaning behind the text, we can use the method of langchain called SentenceTransformersTokenTextSplitter.

```
token_splitter =
SentenceTransformersTokenTextSplitter(chunk_overlap=0,
tokens_per_chunk=256)

token_split_texts = []
for text in character_split_texts:
   token_split_texts += token_splitter.split_text(text)
```

4. Loading the embedding model

We can use a sentence transformer to convert our chunks into embeddings. What sentence transformer does is allow you to embed entire sentences or even small documents like we have here by pooling the output of all the token embeddings to produce a single dense vector

```
from chromadb.utils.embedding_functions import
SentenceTransformerEmbeddingFunction

embedding_function = SentenceTransformerEmbeddingFunction()
```

5. Load Chroma

We start by generating the collection of vectors stores from the given pdf so that our RAG model can extract the similar chunks to the input question.

```
import chromadb

chroma_client = chromadb.Client()
chroma_collection =
chroma_client.create_collection("microsoft_annual_report_2022",
embedding_function=embedding_function)

ids = [str(i) for i in range(len(token_split_texts))]

chroma_collection.add(ids=ids, documents=token_split_texts)
chroma_collection.count()
```

6. Create the whole pipeline

```
import os
import openai
from openai import OpenAI
from dotenv import load_dotenv, find_dotenv

query = "What was the total revenue?"

results = chroma_collection.query(query_texts=[query], n_results=5)
retrieved_documents = results['documents'][0]

_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']

openai_client = OpenAI()

def rag(query, retrieved_documents, model="gpt-3.5-turbo"):
  information = "\n\n".join(retrieved_documents)

  messages = [
    {
      "role": "system",
      "content": "You are a helpful expert financial research assistant.
Your users are asking questions about information contained in an
annual report."
      "You will be shown the user's question, and the relevant
information from the annual report. Answer the user's question using
only this information."
    },
    {"role": "user", "content": f"Question: {query}. \n Information:
{information}"}
  ]

  response = openai_client.chat.completions.create(
    model=model,
    messages=messages,
  )
```

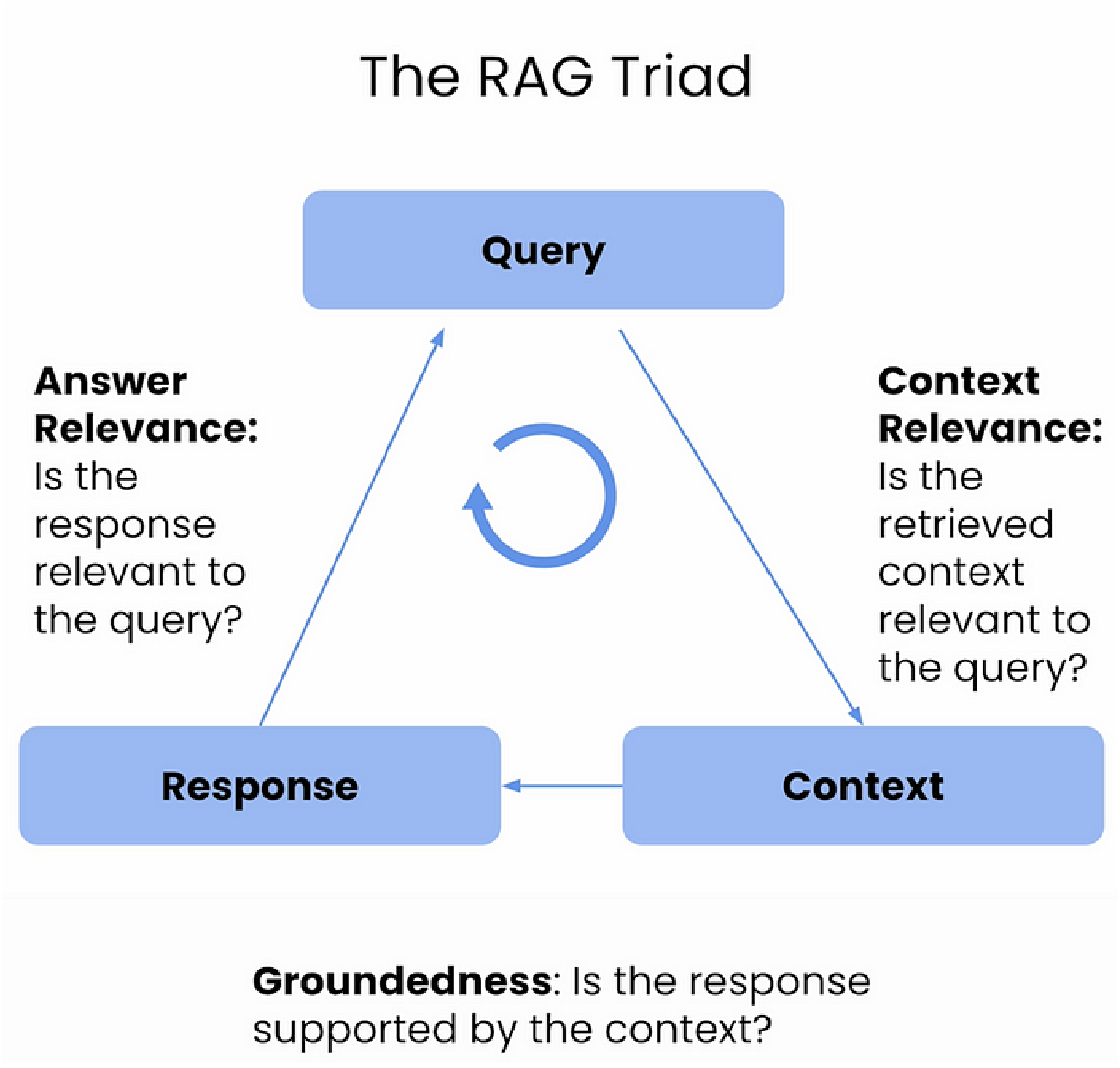
```
    content = response.choices[0].message.content
    return content

output = rag(query=query,
retrieved_documents=retrieved_documents)

print(word_wrap(output))
```

Ok, we have learned how to implement RAG in our system but how do we know that it works accurately? Is there a way to evaluate its performance, actually there are three.

How do we evaluate it?

The evaluation of a RAG pipeline is often referred as the RAG Triad. There are three different evaluation metrics that all measure a different aspect of the RAG process.

- Groundedness checks whether the response from LLM is relevant to the retrieved chunk or not, since the LLM might not have used the chunk instead answered the question on its own.
- Context relevance is similar to the groundedness, the difference is that the context relevance measures the relevancy between the retrieve chunks and the question rather than the response.
- Answer relevance checks the accuracy of the response since the response might be similar to the retrieved chunk and be completely illogical in terms of the accuracy of the answer.


The RAG Triad
Query
Answer Relevance: Is the response relevant to the query?
Context Relevance: Is the retrieved context relevant to the query?
Response
Context
Groundedness: Is the response supported by the context?

RAG Triad[1]

The library Truelens offers built-in functions to use these performance metrics on your RAG pipeline.

Conclusion

Retrieval Augmented Generation (RAG) offers a robust solution to some of the inherent limitations of large language models (LLMs) by integrating a retrieval step that provides relevant context from external sources. This process not only enhances the accuracy and relevance of the generated responses but also ensures transparency by referencing the original sources of information. Through its three-component pipeline — ingestion, retrieval, and synthesis — RAG efficiently combines the strengths of traditional LLMs with the precision of targeted information retrieval. By implementing RAG using tools such as Chroma and evaluating its performance with the RAG Triad, we can significantly improve the reliability and utility of AI-generated answers. As the landscape of AI continues to evolve, RAG stands out as a critical advancement in making LLMs more reliable and contextually aware.