# Introduction

Welcome to the Information Retrieval Week, where we will explore the capabilities of Large Language Models (LLMs) in processing unstructured data. One significant application of LLMs is enabling users to query unstructured data in natural language, which is gaining popularity in industries where data is inherently unstructured (e.g., finance, healthcare).
In this handbook, we will first set up the necessary libraries. Next, we will convert the documents into vector embeddings, which will later be retrieved based on the user's query. Finally, we will evaluate the performance of the LLM based on the response. Let's get started.

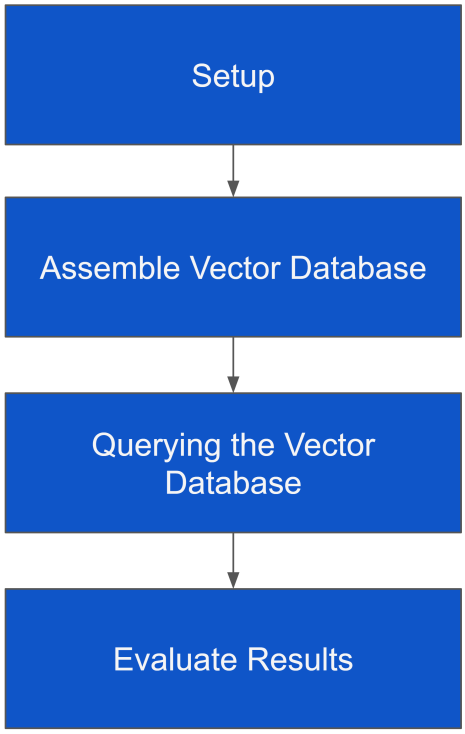


Figure 1

## 1. Setup

***Section Overview:*** *To begin, we'll first set up the required code for the retrieval task. We'll start by installing and importing all the necessary libraries. Then, we'll proceed to authenticate with the OpenAI API. These credentials we can get from the Azure portal as discussed in the Week: Prompt Engineering at Scale.*

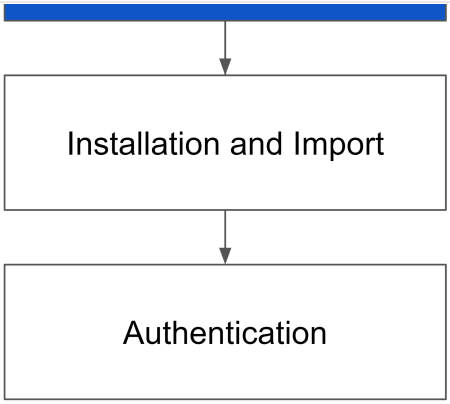


Figure 2

### 1.1 Installation and Import

***Section Overview:*** *The code you provided sets up the necessary environment for an information retrieval system. It installs the required packages, imports the necessary modules and classes, and prepares the system for further processing and querying of PDF documents using Chroma DB, LangChain, and the Azure OpenAI API.*

*Python----------*

```
!pip install -q openai==1.2.0 \
                tiktoken==0.6.0 \
                pypdf==4.0.1 \
                langchain==0.1.1 \
                langchain-community==0.0.13 \
                chromadb==0.4.22 \
                sentence-transformers==2.3.1
```

```
import json
import tiktoken
import pandas as pd
from openai import AzureOpenAI
from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain_community.document_loaders import PyPDFDirectoryLoader
from langchain_community.embeddings.sentence_transformer import (SentenceTransformerEmbeddings)
```

- **Install Libraries:** We will install all the necessary libraries here.
    - `openai` : Provides a Python interface to the OpenAI API, which we'll use for language processing tasks.
    - `tiktoken` : A fast BPE tokenizer for OpenAI's GPT models, used for text processing.
    - `pypdf` : A library for extracting text from PDF files.
    - `langchain` : A framework for building applications with large language models (LLMs).
    - `langchain-community` : Additional components and integrations for LangChain.
    - `chromadb` : A vector database for storing and querying embeddings.
    - `sentence-transformers` : A library for computing sentence embeddings.

- **Import Libraries:** The code then imports the libraries that were installed. Most of the libraries imported here are already discussed previously.
    - `json` : This will be ued for handling JSON data.
    - `tiktoken` : This is used for tokenizing text to count tokens used.
    - `pandas` : This is used for manipulating and analyzing data in tabular format.
    - `openai.AzureOpenAI` : Interacting with the Azure OpenAI API.
    - `langchain.text_splitter.RecursiveCharacterTextSplitter` : Splitting text into smaller chunks for processing.
    - `PyPDFDirectoryLoader` : Loads PDF files from a directory.
    - `SentenceTransformerEmbeddings` : Computes sentence embeddings using the Sentence Transformer model.
    - `Chroma` : Provides a vector store implementation using Chroma DB.

### 1.2 Authentication

***Section Overview:*** *This code snippet is used for OpenAI authentication we have already discussed previously. This code snippet is structured to read authentication credentials and configuration details from a JSON file, initialize an AzureOpenAI client with the provided credentials, and extract the deployment name for a information retrieval task. By loading credentials from an external file, the code ensures secure handling of sensitive information and allows for easy configuration changes without modifying the code directly. The extracted information is then used to set up the AzureOpenAI client fo interacting with the Azure OpenAI API, facilitating information retrieval tasks with the specified model and endpoint.*

*Python----------
---------*

```
creds = json.loads(data)

client = AzureOpenAI(
 azure_endpoint=creds["AZURE_OPENAI_ENDPOINT"],
 api_key=creds["AZURE_OPENAI_KEY"],
 api_version=creds["AZURE_OPENAI_APIVERSION"]
)

deployment_name = creds["CHATGPT_MODEL"]
```

*Python----------*

- **Upload and Read Configuration File:** Here we will read the config.json file. This is the same file we created in previous week of the course (Prompt Engineering at Scale) in the API Setup Hands-on Handbook.
  - `with open(...)`: This is a built-in Python function that opens a file and returns a file objec This will be used to read the file

- **Credentials Assignment:**
  - Assign the values from the `creds` dictionary to the corresponding attributes of the `openai module.These credentials are read by the openai function and act as authentication step.
  - `client = AzureOpenAI(...)`: This is the function that creates an instance of the AzureOpenAI class. This class is used to authenticate to the personalized Open AI model server and make requests to the API.

- **Deployment Name:**
  - `deployment_name = creds["CHATGPT_MODEL"]`: Assign the deployment name of the ch completion endpoint to the **deployment_name** variable.
  - This name is used to specify which chat model should be accessed when making requests t the Azure OpenAI API.

## 2. Assemble Vector Database (Indexing)

***Section Overview:*** *With all the necessary requirements in place, now we will create a vector database*

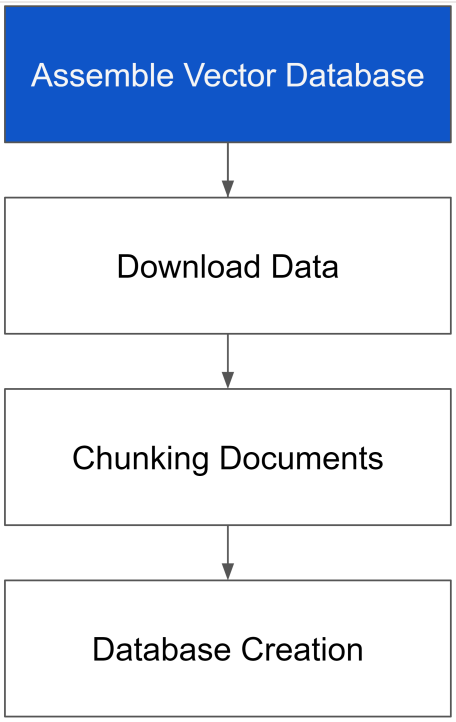


Figure 3

### 2.1 Download Data

***Section Overview:*** *Before starting the process lets download the data.The file with name* ***'tesla-annual-reports.zip'*** *will be provided to you in this section. Upload it in the google colab environment. The code snippet provided below is a command to extract data from a zip file. By unzipping the data file, you can access the contents it contains, such as annual reports and financial data that is compressed in the zip file.*

*Python*----------

```
# Extract data from the zip file
!unzip tesla-annual-reports.zip
```

*Python*----------

- **Load Dataset:**
  - `unzip` is a command-line utility for extracting files from a zip archive.
  - `tesla-annual-reports.zip` is the name of the zip file containing the data that needs to

***Section Overview:*** *This code sets up a pipeline for loading PDF documents from a directory, extracting the text content, and splitting it into smaller chunks using a recursive character-based text splitter. The resulting chunks can be used for further processing, such as indexing and retrieval .*

***Embeddings Recap:***

*Let us first recap what embeddings are:*

- *Embeddings are a type of word representation that allows words with similar meaning to have a similar representation.*
- *They capture semantic properties of words and relations with other words.*

*Similar to word embedding models, sentence embedding models represent the meaning of whole sentences. They are derived by averaging word embeddings or using specialized embedding models.*

*A good method to choose an embedding model, is to use the embedding leaderboard and select an open source model that is performant for the use case at hand.*

*A good general purpose embedding model is* `gte-large`.

*Python*----------

```
pdf_folder_location = "tesla-annual-reports"
pdf_loader = PyPDFDirectoryLoader(pdf_folder_location)
```

```
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    encoding_name='cl100k_base',
    chunk_size=512,
    chunk_overlap=16
)
```

```
# Takes 5 minutes to run
tesla_10k_chunks_ada = pdf_loader.load_and_split(text_splitter)
len(tesla_10k_chunks_ada)
```

```
3342
```

*Python*----------

`pdf_loader = PyPDFDirectoryLoader(pdf_folder_location)`: This PyPDFDirectoryLoader is used to load PDF files from the specified directory. It will recursivel search for PDF files within the directory and its subdirectories.

- **Configuring the text splitter:**
  - `text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(...)`: The RecursiveCharacterTextSplitter is used to split the text extracted from the PDF files into smaller chunks. It uses the tiktoken library for encoding the text.
  - `encoding_name='cl100k_base`: Specifies the encoding to use, in this case, the cl100k_base encoding from the tiktoken library.
  - `chunk_size=512`: Sets the maximum size of each chunk to 512 characters.
  - `chunk_overlap=16`: Specifies the number of overlapping characters between consecutive chunks, which is 16 in this case.

- **Loading and splitting the PDF documents:**
  - `tesla_10k_chunks_ada = pdf_loader.load_and_split(text_splitter)`: Loads the PDF files from the specified directory and splits the extracted text into chunks using the configured text_splitter.
  - The process of loading and splitting the PDF documents can take some time, as indicated by the comment `# Takes 5 minutes to run`.
  - The `len(tesla_10k_chunks_ada)` line prints the total number of chunks created from the PDF documents.

### 2.3 Database Creation

***Section Overview:*** *With an embedding model and chunking strategy in place, we can set up an information processing pipeline, chunked and passed to the embedding model. The document chunk ar embedding is then stored as an entry in the vector database.*

***NOTE:*** *The vector databases themselves are full fledged databases. They support the entire gamut of Create, Read, Update and Delete (CRUD) operations. In this notebook, we will use `langchain` abstractions to chunk and create a persistent database. Refer to the Chroma DB documentation for an overview on CRUD operations.*

***Chroma DB:*** *Chroma DB is an open-source vector database that serves as an AI-native embedding database. It provides tools to simplify the use of embeddings, making it easy to install, run, and manage embeddings. Chroma DB offers a simple core API with functions for creating collections, adding documents, and querying for similar results. The database supports integrations with various vendors*

```
tesla_10k_collection = 'tesla-10k-2019-to-2023'
embedding_model = SentenceTransformerEmbeddings(model_name='thenlper/gte-large')

# Takes some time to run
vectorstore = Chroma.from_documents(
 tesla_10k_chunks_ada,
 embedding_model,
 collection_name=tesla_10k_collection,
 persist_directory='./tesla_db'
)
```

```
vectorstore.persist()

vectorstore_persisted = Chroma(
 collection_name=tesla_10k_collection,
 persist_directory='./tesla_db',
 embedding_function=embedding_model
)

retriever = vectorstore_persisted.as_retriever(
 search_type='similarity',
 search_kwargs={'k': 5}
)
```

```
!zip -r tesla_db.zip /content/tesla_db
```

*Python*----------

- **Creating the Chroma vector store**:
  - `tesla_10k_collection` is the name assigned to the Chroma collection that will store the embeddings of the PDF chunks.
  - `SentenceTransformerEmbeddings` is used to create an instance of the embedding mode In this case, the `'thenlper/gte-large'` model is specified, which is a pre-trained sentence embedding model.
  - `vectorstore = Chroma.from_documents(...)` :This block creates a Chroma vector sto from the `tesla_10k_chunks_ada` list of document chunks.

- The process of creating the vector store can take some time, as indicated by the comment `# Takes some time to run`.

- **Persist and Load persisted vector store**:
  - `vectorstore.persist()`:This line persists the vector store to the specified `persist_directory`, which is `'./tesla_db'` in this case.
  - After running this line, the vector store will be saved to disk, allowing for faster loading and retrieval in subsequent runs.
  - `vectorstore_persisted = Chroma(...)`:This line persists the vector store to the specified `persist_directory`, which is `'./tesla_db'` in this case.
  - This block creates a new instance of the Chroma vector store, loading the persisted data fro the `'./tesla_db'` directory.
  - The `collection_name` parameter specifies the name of the collection to load.
  - The `persist_directory` parameter sets the directory where the persisted data is located
  - The `embedding_function` parameter specifies the embedding model to use for generating embeddings during retrieval.

- **Setting up the retriever**:
  - `retriever = vectorstore_persisted.as_retriever(...)`:This line creates a retrieve from the persisted vector store.
  - The `search_type` parameter is set to `'similarity'`, indicating that the retriever will perform similarity-based search.
  - The `search_kwargs` parameter specifies additional options for the search. In this case, `{'k': 5}` is set, which means that the retriever will return the top 5 most similar chunks for a given query.

- **Zipping the persisted vector storer**:
  - The `zip` command is used to create the zip archive, and the `-r` option specifies recursive compression, which includes subdirectories.
  - The zip code creates a zip archive named `'tesla_db.zip'` containing the persisted vecto store directory `'/content/tesla_db'`.

## 3. Querying the Vector Database

***Section Overview:*** *Once the vector database is set up, embedding generated from the query text is compared with all the document embeddings in the vector database. The most similar documents are*

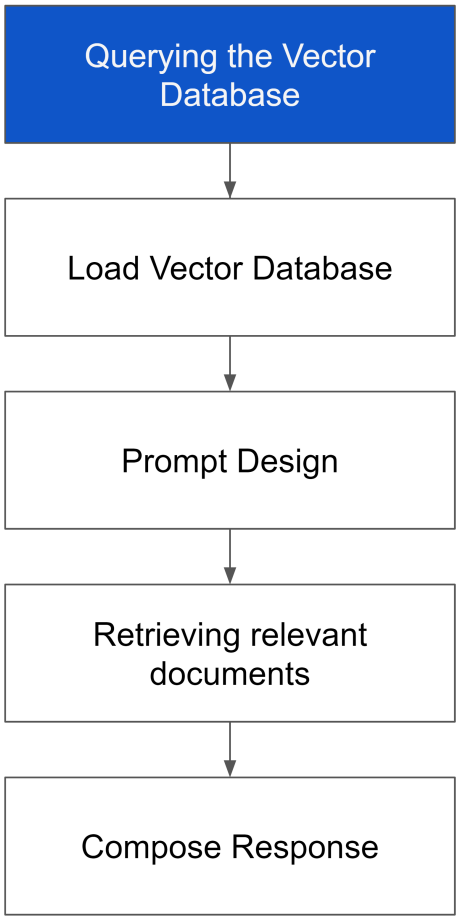


Figure 4

### 3.1 Load Vector Database

***Section Overview:*** *This code sets up a vector store using Chroma DB, loads embeddings from a pre-trained model, and configures a retriever for similarity-based search. The vector store is persisted to disk, and the retriever is set up to return the top 5 most similar chunks for a given query.*

*Python*----------

```
embedding_model = SentenceTransformerEmbeddings(model_name='thenlper/gte-large')
!gdown 1hWbAWhJr5xsl0sAvvEq9Wpo8ItCdZpdq
!unzip tesla_db.zip
```

```
tesla_10k_collection = 'tesla-10k-2019-to-2023'


vectorstore_persisted = Chroma(
```

```
)

retriever = vectorstore_persisted.as_retriever(
 search_type='similarity',
 search_kwargs={'k': 5}
)
```

*Python----------*

- **Downloading and Unzipping the Vector Store**:
  - `embedding_model = SentenceTransformerEmbeddings(model_name='thenlper/gte large')`: This line loads a pre-trained SentenceTransformer model named 'thenlper/gte-large'. This model is used to generate embeddings for the text data.
  - The `gdown` command is used to download a file from Google Drive. The file is identified by i ID `1hWbAWhJr5xsl0sAvvEq9Wpo8ItCdZpdq`.
  - The `unzip` command is used to extract the contents of the downloaded zip file, which is named `tesla_db.zip`. This step is necessary to access the contents of the zip file.

- **Creating the Chroma Vector Store**:
  - `vectorstore_persisted = Chroma(...)`: This block creates a Chroma vector store with the specified name tesla_10k_collection
  - The `persist_directory` parameter specifies the directory where the vector store will be persisted.
  - The `embedding_function` parameter specifies the embedding model to use for generating embeddings during retrieval.

- **Creating the Chroma Vector Store**:
  - `retriever = vectorstore_persisted.as_retriever(...)`: This line creates a retriever from the persisted vector store.
  - The `search_type` parameter is set to `'similarity'`, indicating that the retriever will perform similarity-based search.
  - The `search_kwargs` parameter specifies additional options for the search. In this case, `{'k': 5}` is set, which means that the retriever will return the top 5 most similar chunks for a given query.

*clearly defining the format for providing context and asking questions. It ensures that the assistant can focus on answering user queries effectively based on the provided context without revealing any additional information.*

*Python*----------

```
qna_system_message = """
You are an assistant to a financial services firm who answers user queries on annual reports.
User input will have the context required by you to answer user questions.
This context will begin with the token: ###Context.
The context contains references to specific portions of a document relevant to the user query.

User questions will begin with the token: ###Question.

Please answer user questions only using the context provided in the input.
Do not mention anything about the context in your final answer. Your response should only contain t
he answer to the question.

If the answer is not found in the context, respond "I don't know".
"""
qna_user_message_template = """
###Context
Here are some documents that are relevant to the question mentioned below.
{context}

###Question
{question}
"""
```

*Python*----------

- **Prompt Design:**
  - qna_system_message: This message provides instructions and guidelines for the Q&A syster It explains the role of the assistant, the format of user input, the use of context tokens, and the expected behavior when answering user questions.
  - qna_user_message_template: This template is used to structure the user input for the Q&A system. It includes placeholders for the context and the user question.

to provide an answer based on the context.

### 3.3 Retrieving relevant documents

***Section Overview:*** *Now that we have created the examples for the prompt, let us create the final prom by combining all the components. We will create a function for the same so that we can call it again when required.W*

*Python*----------

```
user_input = "What was the annual revenue of the company in 2022?"
relevant_document_chunks = retriever.get_relevant_documents(user_input)
len(relevant_document_chunks)
```

```
5
```

```
for document in relevant_document_chunks:
 print(document.page_content.replace("\t", " "))
 break
```

```
systems.
In 2020, we recognized total revenues of $31.54 billion, representing an increase of $6.96 billion compared to the prior year. We continue to ramp
production, build new manufacturing capacity and expand our operations to enable increased deliveries and deployments of our products and further revenue
growth.
In 2020, our net income attributable to common stockholders was $721 million, representing a favorable change of $1.58 billion compared to the prior
year. In 2020, our operating margin was 6.3%, representing a favorable change of 6.6% compared to the prior year. We continue to focus on operational
efficiencies, while we have seen an acceleration of non-cash stock-based compensation expense due to a rapid increase in our market capitalization and updat
to our business outlook.
We ended 2020 with $19.38 billion in cash and cash equivalents, representing an increase of $13.12 billion from the end of 2019. Our cash flows from
operating activities during 2020 was $5.94 billion, compared to $2.41 billion during 2019, and capital expenditures amounted to $3.16 billion during 2020,
compared to $1.33 billion during 2019. Sustained growth has allowed our business to generally fund itself, but we will continue a number of capital-intensiv
projects in upcoming periods.
Management Opportunities, Challenges and Risks and 2021 Outlook
Impact of COVID-19 Pandemic
```

*Python*----------

- **User Query and Retrieval of Relevant Documents**:
  - The `user_input` variable contains the user's query: "What was the annual revenue of the company in 2022?"
  - The `retriever.get_relevant_documents(user_input)` method is used to retrieve relevant document chunks based on the user's query.
  - The length of the `relevant_document_chunks` is calculated to determine the number of relevant document chunks retrieved.

- For each document, it prints the content of the document's page, replacing any tab characte with spaces.
- The loop is set to break after printing the content of the first relevant document chunk.

### 3.4 Compose Response

***Section Overview:*** *Now that we have created the examples for the prompt, let us create the final prom by combining all the components. We will create a function for the same so that we can call it again when required.W*

*Python*----------

```
context_list = [d.page_content for d in relevant_document_chunks]
context_for_query = ". ".join(context_list)

prompt = [
 {'role':'system', 'content': qna_system_message},
 {'role': 'user', 'content': qna_user_message_template.format(
 context=context_for_query,
 question=user_input
 )
 }
]

try:
 response = client.chat.completions.create(
 model=deployment_name,
 messages=prompt,
 temperature=0
 )

 prediction = response.choices[0].message.content.strip()
except Exception as e:
 prediction = f'Sorry, I encountered the following error: \n {e}'

print(prediction)
```

*Python----------*

- **Generating Context**:
  - `context_list = [d.page_content for d in relevant_document_chunks]:` This block generates the context for the user query by joining the content of the relevant document chunks with periods and spaces. and `user_message_template` (template for user input).It constructs a prompt that includes system messages, user messages with example reviews, and assistant responses.
  - The context is used to provide additional information to the Azure OpenAI model to help it generate a more accurate response.

- **Preparing the Prompt**:
  - `prompt=[...]`: This block prepares the prompt for the Azure OpenAI API by creating a list dictionaries.
  - The prompt includes two elements:
    - The first element is a system message that provides context for the user query.
    - The second element is a user message that includes the context and the user's question.

- **Generating the Response**:
  - `response = client.chat.completions.create(...):` This block uses the Azure OpenAI API to generate a response based on the prompt.
  - The `client.chat.completions.create` method is used to generate the response.
  - The `model` parameter specifies the model to use for generating the response.
  - The `messages` parameter specifies the prompt to use for generating the response.
  - The `temperature` parameter controls the level of creativity in the response.
  - If an exception occurs during the generation of the response, the code catches the exceptio and prints an error message.
  - `print(prediction):` This line prints the generated response to the console.

## 4. Evaluate Results

***Section Overview:*** *We will now evaluate the responses generated. Since RAG is a special case of text-*

- *Groundedness/Faithfulness: How factually accurate the answer is given the context?*
- *Relevance: How relevant is the context retrieved given the query?*

*These two metrics check the quality two components of the RAG system - retrieval and generation. To evaluate these components individually, it is common to use LLM-as-a-judge method to check the quality of the RAG system. We write two prompts corresponding to each of these metrics and using the user input, context and answer, evaluate the performance of the RAG system.*

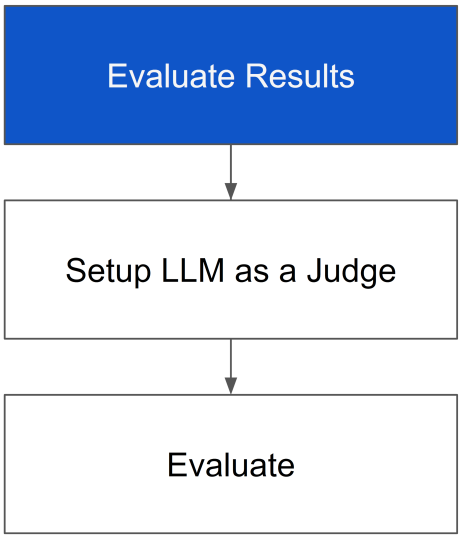


Figure 5

### 4.1 Setup LLM as a Judge

***Section Overview:*** *Let us now use the LLM-as-a-judge method to check the quality of the RAG system on two parameters - retrieval and generation. We illustrate this evaluation based on the answers generated to the question from the previous section. Here we will setup the prompt for the evaluation.*

*Python----------*

```
rater_model = deployment_name # 'gpt-35-turbo'
answer=prediction

groundedness_rater_system_message = """
You are tasked with rating AI generated answers to questions posed by users.
You will be presented a question, context used by the AI system to generate the answer and an AI generated answer to the question.
In the input, the question will begin with ###Question, the context will begin with ###Context while the AI
```

```
1 - The metric is not followed at all
2 - The metric is followed only to a limited extent
3 - The metric is followed to a good extent
4 - The metric is followed mostly
5 - The metric is followed completely

Metric:
The answer should be derived only from the information presented in the context.

Instructions:
1. First write down the steps that are needed to evaluate the answer as per the metric.
2. Give a step-by-step explanation if the answer adheres to the metric considering the question and context a
s the input.
3. Next, evaluate the extent to which the metric is followed.
4. Use the previous information to rate the answer using the evaluaton criteria and assign a score.
"""

relevance_rater_system_message = """
You are tasked with rating AI generated answers to questions posed by users.
You will be presented a question, context used by the AI system to generate the answer and an AI generated
answer to the question.
In the input, the question will begin with ###Question, the context will begin with ###Context while the AI
generated answer will begin with ###Answer.

Evaluation criteria:
The task is to judge the extent to which the metric is followed by the answer.
1 - The metric is not followed at all
2 - The metric is followed only to a limited extent
3 - The metric is followed to a good extent
4 - The metric is followed mostly
5 - The metric is followed completely

Metric:
Relevance measures how well the answer addresses the main aspects of the question, based on the context.
```

```
1. First write down the steps that are needed to evaluate the context as per the metric.
2. Give a step-by-step explanation if the context adheres to the metric considering the question as the input.
3. Next, evaluate the extent to which the metric is followed.
4. Use the previous information to rate the context using the evaluaton criteria and assign a score.
"""

groundedness_user_message_template = """
###Question
{question}

###Context
{context}

###Answer
{answer}
"""

relevance_user_message_template = """
###Question
{question}

###Context
{context}

###Answer
{answer}
"""
```

*Python*----------

- **Setup LLM as a Judge**:
  - `rater_model = deployment_namee:` The rater_model variable is set to the name of the model used for rating the AI-generated answers.
  - `groundedness_rater_system_message:` This message provides instructions and

for the relevance rater. It explains the task, evaluation criteria, and metric for rating the AI-generated answers.

- `groundedness_user_message_template:` This template is used to structure the input fo the groundedness rater. It includes placeholders for the question, context, and answer.
- `relevance_user_message_template):` This template is used to structure the input for the relevance rater. It includes placeholders for the question and context.

### 4.2 Evaluate

***Section Overview:*** *This code generates prompts for the groundedness and relevance raters and uses the Azure OpenAI API to generate responses based on these prompts. The responses are then printed the console.*

*Python*----------

```
groundedness_prompt = [
 {'role':'system', 'content': groundedness_rater_system_message},
 {'role': 'user', 'content': groundedness_user_message_template.format(
 question=user_input,
 context=context_for_query,
 answer=answer
 )
 }
]

response = client.chat.completions.create(
 model=rater_model,
 messages=groundedness_prompt,
 temperature=0
)

print(response.choices[0].message.content)
```

```
relevance_prompt = [
 {'role':'system', 'content': relevance_rater_system_message},
```

```
 }
]

response = client.chat.completions.create(
 model=rater_model,
 messages=relevance_prompt,
 temperature=0
)

print(response.choices[0].message.content)
```

*Python*----------

- **Groundedness Prompt**:
  - `groundedness_prompt = [...]:` This prompt includes the system message for the groundedness rater and the user message template with placeholders for the question, context, and answer.
- **Generating the Groundedness Response:**
  - `response = client.chat.completions.create(...):` This block uses the Azure OpenAI API to generate a response based on the prompt.
  - The `client.chat.completions.create` method is used to generate the response.
  - The `model` parameter specifies the model to use for generating the response.
  - The `messages` parameter specifies the prompt to use for generating the response.
  - The `temperature` parameter controls the level of creativity in the response.
  - The generated response is printed to the console.
- **Relevance Prompt**:
  - `groundedness_prompt = [...]:` This prompt includes the system message for the relevance rater and the user message template with placeholders for the question and context.
- **Generating the Relevance Response:**
  - `response = client.chat.completions.create(...):` This block uses the Azure OpenAI API to generate a response based on the prompt. Here we will pass the relevance