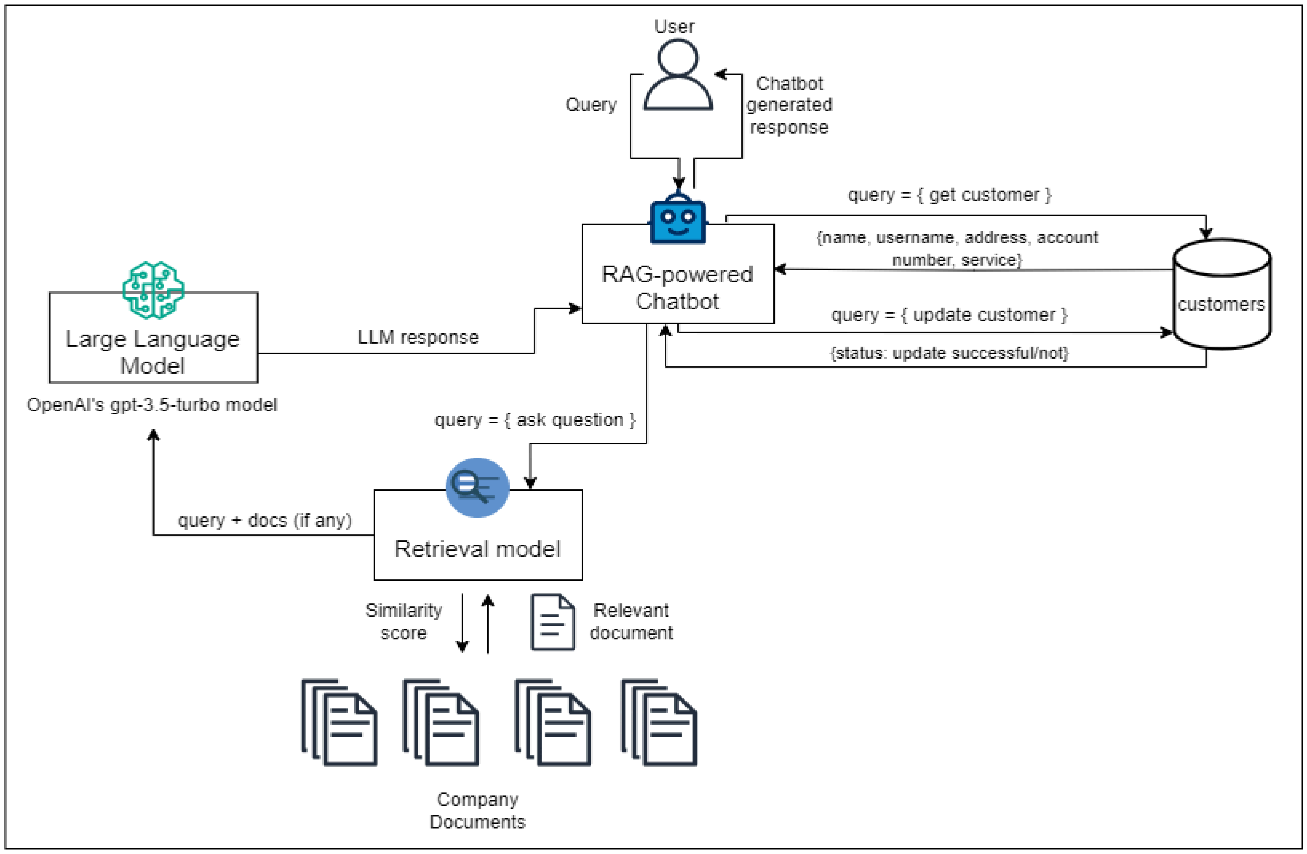
Task breakdown

As a full-stack AI/ML Engineer, I have been tasked with building an autonomous customer service bot. The customer-assistance chatbot I will be implementing in this project is of a Mobile and Internet Service Provider, Mob5GService. Mob5GService has a RAG instance that helps the LLM-powered chatbot to answer user's queries; the queries can be of two types: customer-specific or company-specific. Customer-specific queries include two functionalities: querying the database for customer data and sending update requests on customer data. Company-specific queries include general Q/A to the LLM and querying regarding company specific policy documents. For this implementation I have generated seven sample company documents including, Terms of Service, Privacy Policy, Device Policy, Acceptable Use Policy, Security Policy, Billing and Payment Policy and finally the Services and Products offered.

Large Language Model and Architecture

The Large Language Model selected for this is OpenAI model "gpt-3.5-turbo". After the recent model updates as of 4th January, 2024. OpenAI deprecated several other models and the best model that provides accurate responses to completion's request include "gpt-3.5-turbo". Temperature determines the amount of creativity it is allowed for the model to use. Now let's understand it with the architecture diagram:



Approach

The approach used towards solving this is:

1. **Select the Model and set environment:**
   The choice of a large language learning model determines the accuracy of the model's response. We have used OpenAI's gpt-3.5-turbo model and defined its hyperparameters like how many tokens it can generate, the amount of creativity we want in our responses through setting the temperature. Next to access the OpenAI LLM we need to define the authorization API key as the environment variable to the application. It is a paid feature. Here's how you define the OPENAI-API-KEY. Replace the "sk-xxx" with your own key.
```
os.environ['OPENAI_API_KEY'] = 'sk-xxxx'
# Ensure OpenAI API key is set
openai.api_key = "sk-xxx"
```

2. **Set the logger:**
   To log all of the customers to bot interaction and the usage of resources. We have defined a logger file 'output.log' .To define the the logger we use the following:
```
logfile = "output1.log"

logger.add(logfile, colorize=True, enqueue=True)
handler_1 = FileCallbackHandler(logfile)
handler_2 = StdOutCallbackHandler()
```

   And the then to log output generated from chatbot we use the following statement wherever response is printed:
```
logger.info(f"Accessed: {resource} \nResponse: {response}")
```

3. **Define Customer database:**
   We need custom customer data to test our operations on. To do that first we need to define a database to store the customer data and fetch it from the database. We use SQLite3 to define our customer database. To define database:
```
def create_connection(db_file):
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        drop_table(conn)
        create_table(conn)
        add_sample_customers(conn)
        return conn
    except Error as e:
        print(e)
    return conn
```

We also have create and delete table functionality. Overview of these is that they define a customers tables in the database with the following columns:

```
username, password, name, email, address, account, serviceUse
```

Defining the personal details and the plan currently in use by the customer

4. **Define authentication:**
   One thing that we need to ensure is that the customer must authenticate itself before trying to access the customer data for this we have made an authentication mechanism if the user passes that then it is allowed access to database contents:

```
# authenticate customers for personalized tasks
def authenticate_customer(conn, account_number, password):
    try:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM customers WHERE account = ?
AND password = ?", (account_number, password))
        customer = cursor.fetchone()
        if customer:
            return True
        else:
            return False
    except Error as e:
        print("Error:", e)
        return False
```

5. **Define get and update function:**
   Next we need to define what happens if the user authenticates itself as a valid customer. It can get its information from the database and also can update its info in the database. Here’s how you can fetch data from the database:

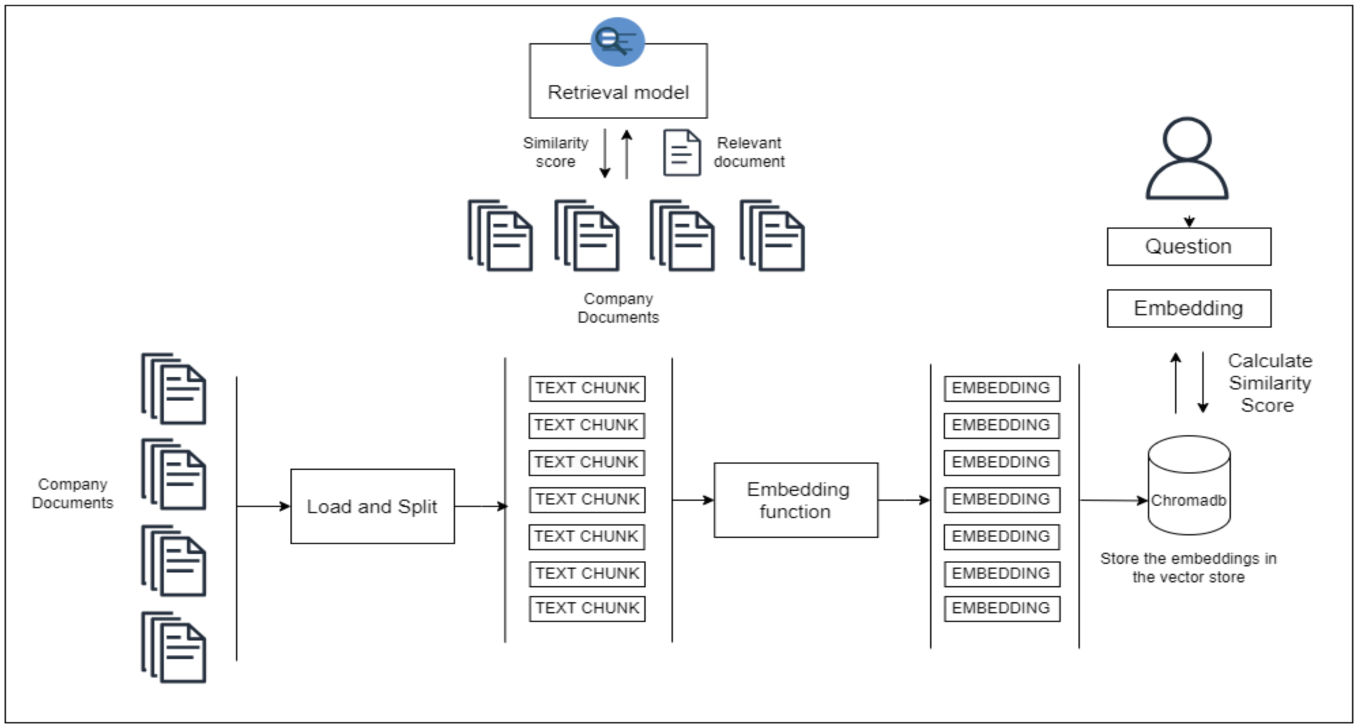
```
def query_customer(conn, username):
    print("Verification required")
    while True:
        account_number = input("Enter your account number: ")
        password = input("Enter your password: ")
        # Authenticate the customer
        if authenticate_customer(conn, account_number, password):
            print("Authentication successful!")
            break
        else:
            print("Invalid credentials. Please try again.")
            break
        return None

    try:
        sql_select_customer = f""" SELECT * FROM customers WHERE
username = ? """
        c = conn.cursor()
        c.execute(sql_select_customer, (username,))
        return c.fetchone()
    except Error as e:
        print(e)
        return None
```

And here is how you can update your data in the customer database:

```
def update_customer(conn, username, field, new_value):
    print("Verification required")
    while True:
        account_number = input("Enter your account number: ")
        password = input("Enter your password: ")
        # Authenticate the customer
        if authenticate_customer(conn, account_number, password):
            print("Authentication successful!")
            break
        else:
            print("Invalid credentials. Please try again.")
            break
        return None
    try:
        sql_update_customer = f""" UPDATE customers
                                   SET {field} = ?
                                   WHERE username = ? """
        c = conn.cursor()
        c.execute(sql_update_customer, (new_value, username))
        conn.commit()
    except Error as e:
        print(e)
        return False
    return True
```

One thing common in theses is that the user can't proceed until the user is authenticated thus it ensure that the confidential data of the customer cannot be accessed

6. **Define RAG functionality**
   We need to retrieve relative documents from the company's documents to provide users with query-specific documentation. To understand this we first need to analyze the working of the Retrieval Augmented Generation RAG. We first get the document, split them into chunks and calculate their embedding. To store the embeddings we use the Chromadb vector store.



When the user puts in their question a similar embedding is generated and its similarity score is calculated with all the embeddings stored in the vector store. This similarity score helps in retrieving the relevant documents that may help answer user queries. If the query doesn't generate any relevant document. We employ prompt engineering to control the model's behavior let see how we did this in the code:

```
class RAG:
    def __init__(self, docs_dir: str, n_retrievals: int = 6,
chat_max_tokens: int = 300, model_name="gpt-3.5-turbo", temperature: float
= 0.7):
        self.__model = self.__set_llm_model(model_name, temperature)
        self.__docs_list = self.__get_docs_list(docs_dir)
        self.db = self.__set_chroma_db()
        self.__retriever = self.__set_retriever(k=n_retrievals)
        self.__chat_history =
self.__set_chat_history(max_token_limit=chat_max_tokens)

    def __set_llm_model(self, model_name="gpt-3.5-turbo", temperature:
float = 0.7):
        return ChatOpenAI(model_name=model_name, temperature=temperature)

    def __get_docs_list(self, docs_dir: str) -> list:
        loader = DirectoryLoader(docs_dir, recursive=True,
show_progress=True, glob="*.docx",
loader_cls=UnstructuredWordDocumentLoader)
        docs_list = loader.load_and_split()
        return docs_list

    def __set_chroma_db(self):
        embeddings = OpenAIEmbeddings()
        db = Chroma.from_documents(self.__docs_list, embeddings)
        return db

    def __set_retriever(self, k: int = 4):

        # Similarity search
        # query = "What did the president say about Ketanji Brown Jackson"
        # docs = db.similarity_search(query)
        metadata_field_info = [
            {
                "name": "source",
                "description": "The directory path where the document is
located",
                "type": "string"
            },
        ]
        document_content_description = "Personal documents"
        retriever = self.db.as_retriever(
            search_kwargs={"k": k}
          )
        return retriever
    def __set_chat_history(self, max_token_limit: int = 300):
        return ConversationTokenBufferMemory(llm=self.__model,
max_token_limit=max_token_limit, return_messages=True)

    def ask(self, question: str):
      # Perform similarity search with Chroma vector database
        relevant_docs = self.db.similarity_search(question)

        if relevant_docs:
            # If relevant documents are found, use them to generate a
response
            context = "\n".join([doc.page_content for doc in
relevant_docs])
            messages = [
                SystemMessage(content="""You are a chat assistant for a
mobile and internet service provider company Mob5GService.
                If irrelevant question are asked then explain that its not
your job."""),
                HumanMessage(content=f"Context: {context}\n\nQuestion:
{question}")
            ]
            response = self.__model(messages)
            response = response.content

            logger.info(f"Accessed: RAG \nResponse: {response}")
            #final_response = StrOutputParser().parse(response)
            #logger.info(final_response)

        else:
            # If no relevant documents are found, generate a response
directly using the language model
            response = self.__model([HumanMessage(content=question)])
            response = response.content

            logger.info(f"Accessed: Q/A model \nResponse: : {response}")

        return response
```

Notice how we have controlled the chatbot's performance if irrelevant questions are asked using a prompt that specifies its work.

7. **Define Chatbot:**

Now that we have all of our functionalities in place we can finally call the functionalities for customer-specific and company-specific queries using the chatbot. To do that we use the following code.

```
def chatbot(user_input, conn, rag):
    if user_input.startswith("get customer"):
        username = user_input.split(" ")[-1]
        customer = query_customer(conn, username)
        if customer:
            response = {
                "username": customer[0],
                "password": customer[1],
                "name": customer[2],
                "email": customer[3],
                "address": customer[4],
                "account": customer[5],
                "serviceUse": customer[6]
            }
            logger.info(f"Accessed: DB GET \nResponse:
{format_response(response)}" )
            return format_response(response)
        else:
            return {"error": "Customer not found"}
    elif user_input.startswith("update customer"):
        parts = user_input.split(" ")
        username = parts[2]
        field = parts[3]
        print(username, field)
        new_value = " ".join(parts[4:])
        success = update_customer(conn, username, field, new_value)
        if success:
            logger.info("Accessed: DB UPDATE \nResponse: Customer
updated")
            return {"status": "Customer updated"}
        else:
            logger.info("Accessed: DB UPDATE \nResponse: Customer not
found or update failed")
            return {"error": "Customer not found or update failed"}
    else:
        response = rag.ask(user_input)
        return response
```

We have simply called the above-mentioned functionalities in the chatbot function based on the customer message.

8. **Initialize database, RAG and chatbot:**

To start the chatbot first we need to establish the database connection and define the instance for the RAG functionality and call it using a chatbot. To start a connection we use:

```
# Initialize database connection
conn = create_connection("customers.db")
```

To initialize RAG we use:

```
# Initialize RAG instance
rag = RAG(
    docs_dir='/content/Documents',  # Name of the directory where the
documents are located
    n_retrievals=1,  # Number of documents returned by the search
    chat_max_tokens=300,  # Maximum number of tokens that can be used in
chat memory
    temperature=1.2,  # How creative the response will be
)
```

And finally to start the chatbot we use the main function:

```
print("\nType 'exit' to exit the program.")
while True:
    user_input = input("Enter your message: ")
    if user_input.lower() == "exit":
        break
    response = chatbot(user_input, conn, rag)
    print('Response:', response)
```

This is the complete functionality of the chatbot.

9. **Create Flask application**

   To create an API for the application using REST protocol. To do that we use a flask application and to do that we define another file api_setup.py to have all the endpoints to access the chatbot functionality.

10. **Define endpoints:**

The endpoints in the flask application are deployed for three. `get_customer, update customer, ask_question.`

To do this we import functionalities from the chatbot file and call then in endpoints like

```
from dotenv import load_dotenv
from flask import Flask, request, jsonify
from chatbot import RAG, create_connection, chatbot, query_customer,
update_customer

load_dotenv()
###
@app.route('/get_customer', methods=['GET'])
def get_customer():
    username = request.args.get('username')
    customer = query_customer(conn, username)
    if customer:
        response = {
            "username": customer[0],
            "password": customer[1],
            "name": customer[2],
            "email": customer[3],
            "address": customer[4],
            "account": customer[5],
            "serviceUse": customer[6]
        }
        return jsonify(response)
    else:
        return jsonify({"error": "Customer not found"}), 404

@app.route('/update_customer', methods=['POST'])
def update_customer():
    data = request.get_json()
    username = data.get('username')
    field = data.get('field')
    new_value = data.get('new_value')
    success = update_customer(conn, username, field, new_value)
    if success:
        return jsonify({"status": "Customer updated"})
    else:
        return jsonify({"error": "Customer not found or update failed"}),
400

@app.route('/ask_question', methods=['GET'])
def ask_question():
    question = request.args.get('question')
    response = rag.ask(question)
    return jsonify({"response": response})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

If we run this there is a JSON app running that gives the exact same response but at a URL on a specific port.

11. **Check for parallelism**
The chatbot also has the ability to answer multiple questions using the same session. The application won't terminate unless it is instructed to.
12. **Containerize the application**
To containerize the application we use a docker setup to deploy it in a container because we are using GPU in our execution we use nvidia-cuda base image

```
FROM nvidia/cuda:11.0-base
```

We define a working directory and copy our files to the working directory

```
# Set working directory
WORKDIR /AfinitiSubmission
RUN mkdir /AfinitiSubmission/Documents

# Copy your application code into the container
COPY rag_afiniti.py /AfinitiSubmission
COPY Documents/ServicesandProducts /AfinitiSubmission/Documents
COPY Documents/DevicePolicy /AfinitiSubmission/Documents
COPY Documents/PrivacyPolicy /AfinitiSubmission/Documents
COPY Documents/BillingPaymentPolicy /AfinitiSubmission/Documents
COPY Documents/AcceptableUsePolicy /AfinitiSubmission/Documents
COPY Documents/TermsofService /AfinitiSubmission/Documents
```

Next we install all the requirements for the the application using pip

```
# Install necessary dependencies
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
 && rm -rf /var/lib/apt/lists/*

# Install additional dependencies
RUN pip3 install python-dotenv \
    langchain \
    langchain-openai \
    langchain-community \
    openai \
    unstructured \
    tiktoken \
    lark \
    langchain-chroma \
    "unstructured[docx]" \
    loguru
```

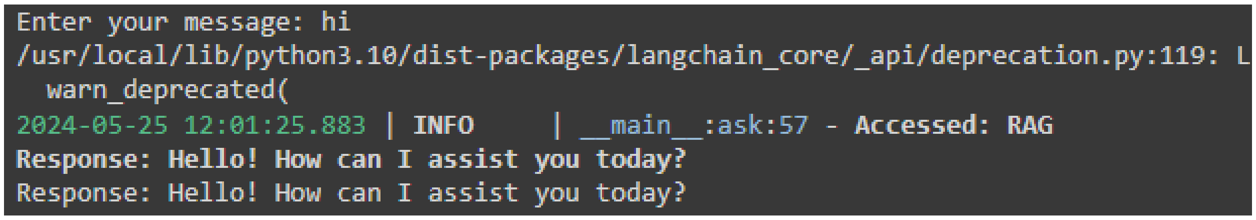
And finally define environment variables (if any) and run the application:

```
# Set environment variables
ENV OPENAI_API_KEY="sk-xxx"
# Command to run your application
CMD ["python3", "rag_afiniti.py"]
```

Now the application can be deployed.

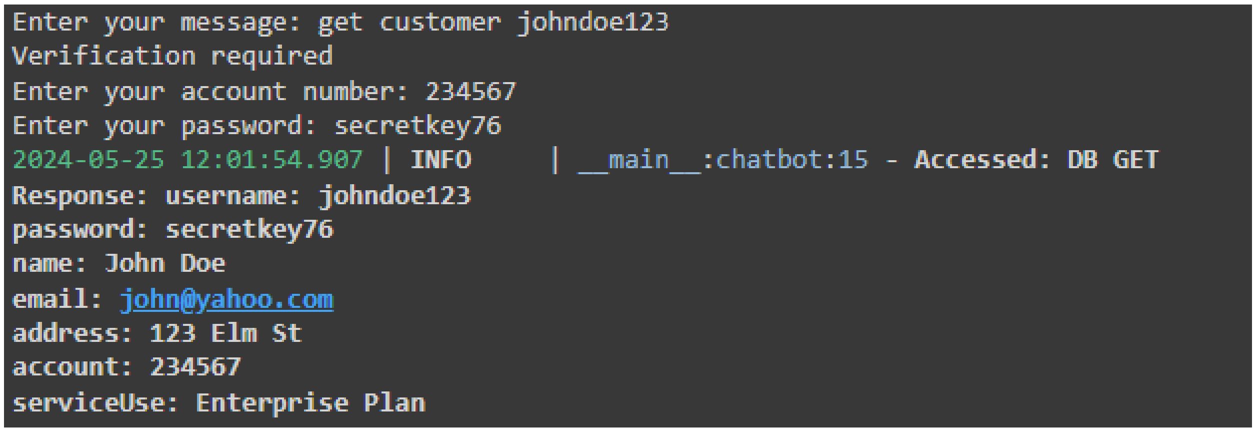
Testing of our application

To test our application, we can use the chatbot
For example
To answer an answer not document related and it is also logged



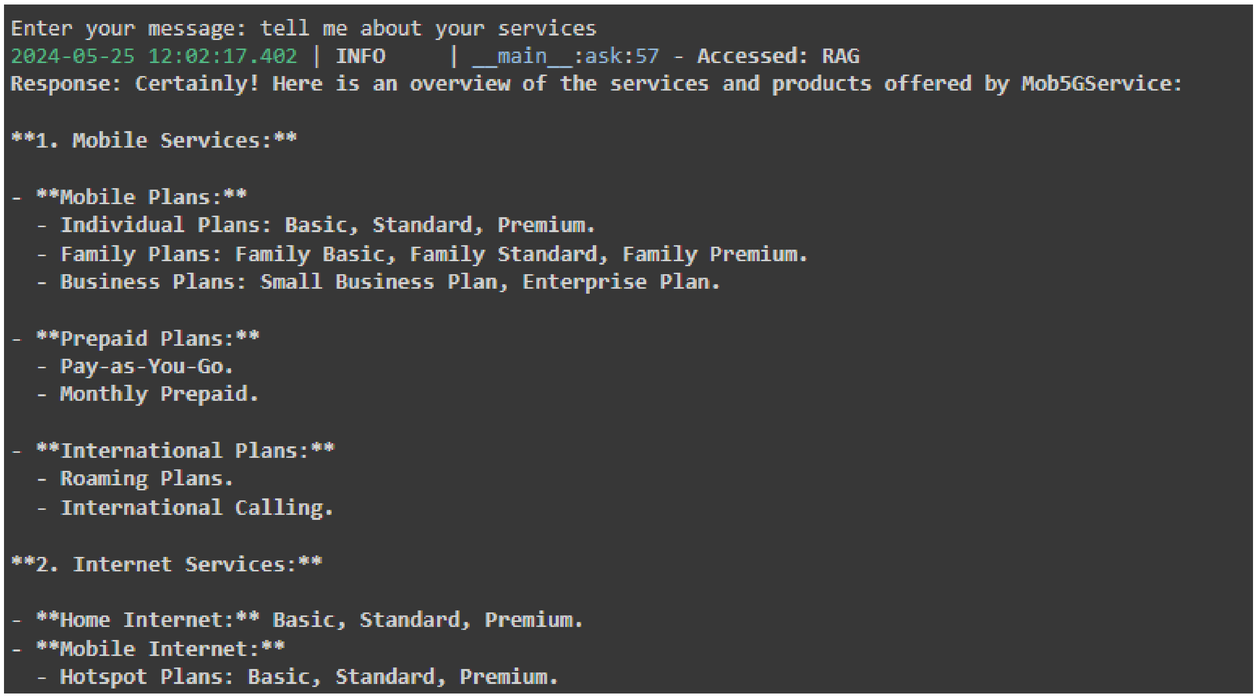
```
Enter your message: hi
/usr/local/lib/python3.10/dist-packages/langchain_core/_api/deprecation.py:119: L
  warn_deprecated(
2024-05-25 12:01:25.883 | INFO     | __main__:ask:57 - Accessed: RAG
Response: Hello! How can I assist you today?
Response: Hello! How can I assist you today?
```

To get a customer information from the database



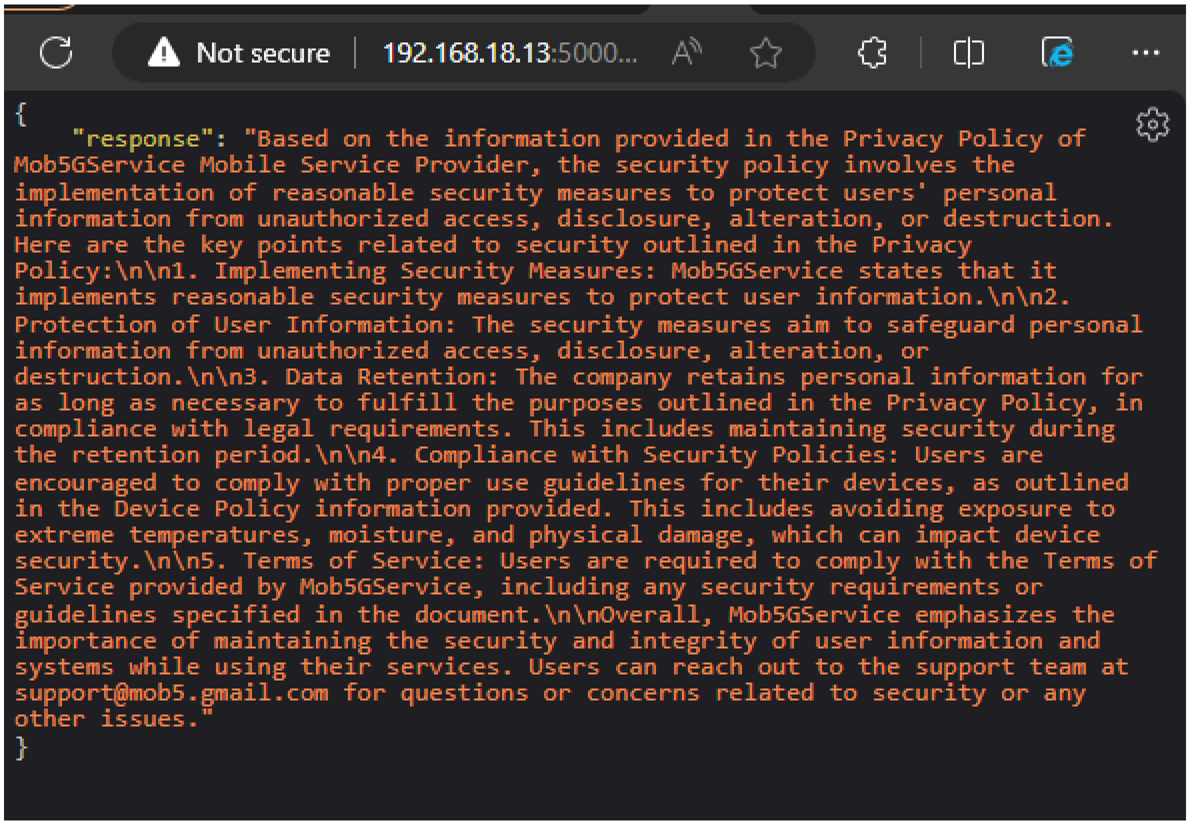
```
Enter your message: get customer johndoe123
Verification required
Enter your account number: 234567
Enter your password: secretkey76
2024-05-25 12:01:54.907 | INFO     | __main__:chatbot:15 - Accessed: DB GET
Response: username: johndoe123
password: secretkey76
name: John Doe
email: john@yahoo.com
address: 123 Elm St
account: 234567
serviceUse: Enterprise Plan
```

To answer a document-related question



```
Enter your message: tell me about your services
2024-05-25 12:02:17.402 | INFO     | __main__:ask:57 - Accessed: RAG
Response: Certainly! Here is an overview of the services and products offered by Mob5GService:

**1. Mobile Services:**

- **Mobile Plans:**
  - Individual Plans: Basic, Standard, Premium.
  - Family Plans: Family Basic, Family Standard, Family Premium.
  - Business Plans: Small Business Plan, Enterprise Plan.

- **Prepaid Plans:**
  - Pay-as-You-Go.
  - Monthly Prepaid.

- **International Plans:**
  - Roaming Plans.
  - International Calling.

**2. Internet Services:**

- **Home Internet:** Basic, Standard, Premium.
- **Mobile Internet:**
  - Hotspot Plans: Basic, Standard, Premium.
```

Similarly we can can get the answers with API like for example
I asked about the security policy of the Mob5GService. This is the response I got in JSON



Not secure | 192.168.18.13:5000...

```
{
    "response": "Based on the information provided in the Privacy Policy of Mob5GService Mobile Service Provider, the security policy involves the implementation of reasonable security measures to protect users' personal information from unauthorized access, disclosure, alteration, or destruction. Here are the key points related to security outlined in the Privacy Policy:\n\n1. Implementing Security Measures: Mob5GService states that it implements reasonable security measures to protect user information.\n\n2. Protection of User Information: The security measures aim to safeguard personal information from unauthorized access, disclosure, alteration, or destruction.\n\n3. Data Retention: The company retains personal information for as long as necessary to fulfill the purposes outlined in the Privacy Policy, in compliance with legal requirements. This includes maintaining security during the retention period.\n\n4. Compliance with Security Policies: Users are encouraged to comply with proper use guidelines for their devices, as outlined in the Device Policy information provided. This includes avoiding exposure to extreme temperatures, moisture, and physical damage, which can impact device security.\n\n5. Terms of Service: Users are required to comply with the Terms of Service provided by Mob5GService, including any security requirements or guidelines specified in the document.\n\nOverall, Mob5GService emphasizes the importance of maintaining the security and integrity of user information and systems while using their services. Users can reach out to the support team at support@mob5.gmail.com for questions or concerns related to security or any other issues."
}
```

Final Insights:

A RAG powered LLM boy can be extremely beneficial to get customer's attention. RAG helps in efficiently extracting information from documents and getting accurate responses from the LLM. Managing the prompt for the chatbot is very important. For example, if I set the prompt as "You are a helpful assistant"
The LLM would answer all related and unrelated questions. Like "who is the president of the United States?" and LLM would respond accurately. But since the check is applied only relevant information can be extracted out of the customer-service chatbot. Also user authentication helps managing the security of the database.