Objective:

The objective of this document is to summarize the steps and learnings while working on the Meaku chatbot. The problem statement is to design an AI bot for the HackerEarth website that can effectively answer questions about the company, encourage users to sign up for a demo, ask relevant questions to carry longer conversations, show source link for the responses and capture their contact details for further follow-up. The bot could run on a local domain for now providing a good customer experience.

Requirements:

1. **Bot Functionality:** Develop a conversational AI bot capable of:
    - Answering questions about HackerEarth, its products, services, and mission.
    - Providing information about available demos and their benefits.
    - Prompting users to sign up for a demo or contact HackerEarth for more details.
    - Ask relevant questions to users to carry longer conversations (aim for 4 min)
    - Show source URLs / docs so user can click and see where the given information was taken from (like perplexity)

2. **User Engagement and Conversion:** Implement strategies to engage users effectively and convert their interest into action. This includes persuasive messaging, clear call-to-action prompts, and personalized responses tailored to user queries.

3. **Data Capture and Storage:** Enable the bot to capture user contact details seamlessly. Upon user consent, collect information such as name, email address, company name, and any other relevant details. Utilize Google Sheets API or similar tools to securely store this data in a designated spreadsheet.

4. **Content and Design Alignment:** Ensure that the bot's language, tone, and design align with HackerEarth's branding and communication guidelines. Maintain consistency with the website's overall look and feel to provide a cohesive user experience.

In this document, we summarize a demo version of an AI bot that would solve the problem of generating more relevant content during the conversations. We implement RAG on top of a base model from Ollama and talk about observed results and future possibilities for improvement.

RAG implementation:

1. **Scrape web pages from HackerEarth:** In this step, we are using beautifulsoup library and following python code to extract the relevant information from one of the articles on

hackerearth websites. The information is stored in the form of documents *(.txt / .md format)* as plain text separated by '\n'. The pulled dataset requires some cleaning.

webscrapper.py

```
from bs4 import BeautifulSoup
import pandas as pd
import requests
import re
from loguru import logger

class get_HTML_tags(object):
   def __init__(self, url):
       self.url = url
       self.df = None

   def extract_links_and_text(self):
         """Extracts href links and corresponding text from a webpage and stores them in a
pandas DataFrame.

       Args:
         url: The URL of the webpage to scrape.

       Returns:
         A pandas DataFrame containing the extracted href links and text.
       """
       # driver = webdriver.Firefox()
       page = requests.get(self.url)
       # html = driver.page_source
       soup = BeautifulSoup(page.text, "html")
       # driver.quit

       links_data = []
       for link in soup.find_all("a"):
           href = link.get("href")
           text = link.text.strip()
           links_data.append({"text": text, "href": href})

       self.df = pd.DataFrame(links_data)
       logger.debug("extracting urls is completed")

   def extract_headings_paragraphs(url, output_file):
       """Scrapes a webpage and saves the content to a text file.

       Args:
         url: The URL of the webpage to scrape.
         output_file: The path to the output text file.
       """

       response = requests.get(url)
       soup = BeautifulSoup(response.content, "html.parser")
```

```
        text_content = ""
        for element in soup.find_all(["h1", "h2", "h3", "h4", "h5", "h6", "p", "li"]):
            text_content += element.text + "\n"

        with open(output_file, "w", encoding="utf-8") as f:
            f.write(text_content)
```

2. **Store documents to vectorized ChromaDB**: We tried multiple embedding models and most of them resulted in timeout or refused connection because of their paid tier service. OpenAI has closed out all their API from free services and finally the following API from HuggingFace worked for converting the documents to their embeddings and storing them into a chroma database.

```
from langchain_community.embeddings import HuggingFaceEmbeddings

embedder = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")

db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedder)

last_request_time = 0
RATE_LIMIT_INTERVAL = 10

chunks_with_ids = calculate_chunk_ids(chunks)

# Add or Update the documents.
existing_items = db.get(include=[])  # IDs are always included by default
existing_ids = set(existing_items["ids"])
print(f"Number of existing documents in DB: {len(existing_ids)}")

# Only add documents that don't exist in the DB.
new_chunks = []
for chunk in chunks_with_ids:
   if chunk.metadata["id"] not in existing_ids:
       new_chunks.append(chunk)

if len(new_chunks):
   print(f"Adding new documents: {len(new_chunks)}")
   new_chunk_ids = [chunk.metadata["id"] for chunk in new_chunks]
   db.add_documents(new_chunks, ids=new_chunk_ids)
   db.persist()
else:
   print("No new documents to add")

# for i, chunk in enumerate(chunks):
#     print(i)
#     current_time = time.time()
#     if current_time - last_request_time < RATE_LIMIT_INTERVAL:
#         time.sleep(RATE_LIMIT_INTERVAL - (current_time - last_request_time))
#     last_request_time = current_time
#     # Create a new DB from the documents.
```

```
#     max_retries = 2
#     for attempt in range(max_retries):
#         try:
#             db = Chroma.from_documents(
#                 [chunk], embedder, persist_directory=CHROMA_PATH
#                 )
#         except Exception as e:
#             if attempt == max_retries - 1:
#                 raise e
#             time.sleep(1 * (2**attempt))

# db.persist()

print(f"Saved {len(chunks)} chunks to {CHROMA_PATH}.")
```

3. **Identify Most relevant chunks:** In order to utilize the information from the documents stored in the above chroma db, we need to find the top k (k=5) chunks that contains most relevant information for the query text. We use a similarity score function to find these relevant sources and store them as context in the prompt template for the base model to work with most recent data.

```
# Query the RAG
query_text = "Why should you use HackerEarth Assessment?"

embedding_function = embedder
db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function)

# Search the DB.
results = db.similarity_search_with_score(query_text, k=5)
print(results)
```

4. **Prepare the prompt and feed the LLM:** The above results are converted to a single text string called context and fed to LLM base models using prompt template which uses the context to answer the query text. The response of LLM with and without RAG is drastically different with the former being a more accurate representation of the information about the query text. The relevancy was manually checked and in future requires a labeled dataset to capture the evaluation metrics. The selected model from Ollama is based on local computers RAM constraints, in future with better machines, higher end models can be deployed.

```
from langchain.prompts import ChatPromptTemplate
from langchain_community.llms.ollama import Ollama

context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results])

RAG_PROMPT_TEMPLATE = """
```

```
Answer the question based on the following context:

{context}

---

Answer the question based on the above context: {question}
Do not repeat any information and be precise in your response.
"""

prompt_template = ChatPromptTemplate.from_template(RAG_PROMPT_TEMPLATE)
prompt = prompt_template.format(context=context_text, question=query_text)
# print(prompt)

# model = Ollama(model="mistral")
model = Ollama(base_url="http://localhost:11434", model="qwen:1.8b")
response_text = model(prompt)

sources = [doc.metadata.get("id", None) for doc, _score in results]
formatted_response = f"Response: {response_text}\nSources: {sources}"
print(formatted_response)
```

Chatbot

In order to design a chatbot, we first need to think about a prompt template that can calculate the context every time a user asks questions and embed it into the prompt which can then be fed to LLM for response. We utilized the following message template which is converted to prompt using ChatPromptTemplate.

```
meaku_message = [
   (
       "system",
       "You are a helpful assistance for HackerEarth company. \
You first greet the customer in one line then collect their details such as name, email,
contact number. \
You wait for the customer to respond then respond using the following context. \
{context_text} \
Answer the question using the above context and don't repeat anything. \
Your job is to engage customers asking about what they are looking for on the website so ask
questions after responding. \
You respond in a short, very conversational friendly style. \
The company is about provide Human Resource services to their customers, the services
include \
1. Repository of pre-built questions \
2. Assess a large pool of candidates in 38 different programming languages \
3. Versatile platform \
4. Advanced proctoring settings for assessments \
```

```
5. Detailed reports and analytics \
6. Dedicated account manager and product specialist available 24/7 \
7. Evaluates tests automatically to shortlist the best candidates \
If they are not looking for anything specific, you provide them details about what the
company does and how its products can benefit the customer, \
            ",
    ),
    ("human", "{input_text}"),
]

prompt_template = ChatPromptTemplate.from_messages(meaku_message)
```

Once we have the prompt template, we now need some functions and a frontend UI that can collect all the inputs from humans and responses from LLM and append them to a message template for LLM to keep track of the context of the conversations.
The following code gets the context from the chroma database we designed earlier and feeds it to the meaku message template to design the prompt based on the input text. The prompt is then used by the completion function to return the content from the response.

```
def get_context(query_text):
    embedding_function = HuggingFaceEmbeddings(
        model_name="sentence-transformers/all-MiniLM-L6-v2"
    )
    db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function)

    # Search the DB.
    results = db.similarity_search_with_score(query_text, k=5)
    context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results])
    return context_text

def get_completion_from_messages(prompt, input_text, model=llm):
    context_text = get_context(input_text)
    chain = prompt | model
    response_text = chain.invoke(
        {"context_text": context_text, "input_text": input_text}
    )
    # response = openai.ChatCompletion.create(
    #     model=model,
    #     messages=messages,
    #     temperature=0, # this is the degree of randomness of the model's output
    # )
    return response_text.content
```

Finally, we use panel library to create an interactive dashboard for users to interact with our RAG implemented LLM model

```
import panel as pn
```

```
panels = []  # collect display
def collect_messages(_):
    input_text = inp.value_input
    inp.value = ""
    prompt_template.extend([("human", f"{input_text}")])
    response = get_completion_from_messages(prompt_template, input_text, model=llm)
    prompt_template.extend([("system", f"{response}")])
    panels.extend(pn.Row("Human:", pn.pane.Markdown(input_text, width=600)))
    panels.extend(
        pn.Row(
            "System:",
            pn.pane.Markdown(
                response, width=600, styles={"background-color": "#F6F6F6"}
            ),
        )
    )

    return pn.Column(*panels)


inp = pn.widgets.TextInput(value="Hi", placeholder="Enter text here...")
button_conversation = pn.widgets.Button(name="Chat!")

interactive_conversation = pn.bind(collect_messages, button_conversation)

dashboard = pn.Column(
    inp,
    pn.Row(button_conversation),
    pn.panel(interactive_conversation, loading_indicator=True, height=300),
)

dashboard
```

Upon eyeballing the response, the LLM model with RAG implementation is doing a pretty decent job, few things that can be done to improve the chatbot even further are below:

1. **Fine Tuning:** We can build a customer fine-tuning dataset that contains input, output pairs in the form of question answer template or simple sentence response template to improve the accuracy of the model.
2. **Agentic RAG:** Instead of choosing top 5 chunks during RAG implementation, we can add another model layer on top of context text to identify how relevant these chunks are to the input text going beyond just matching for similarity based scores. This will allow a use of LLM model to re-write the input query optimized for the user journey and company relevance.
3. **High-end LLM models:** This document was written based on LLM models that are smaller in size and easy runnable using 8GB CPU RAM. Given the resources, high end LLM models like mistral, llama 3.1 can be applied at the chatbot layer to make the meaku chatbot agent even more accurate in handling customer's request.