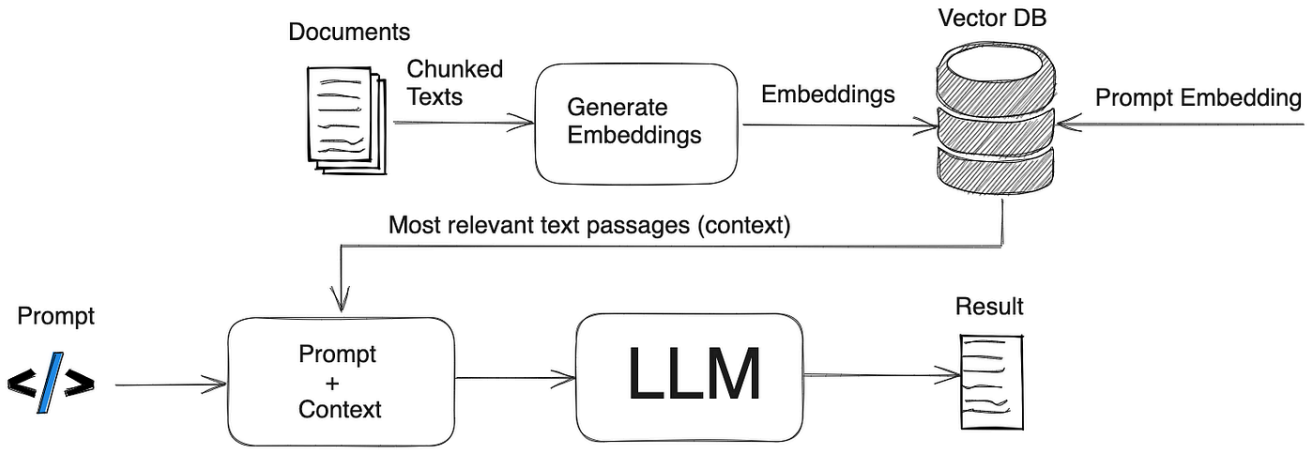
RAG-based AI Chatbot that answers questions about yourself

System Architecture

The following diagram shows the architecture used to create the application.



There are two main blocks in the architecture:

1. Data Ingestion: This includes loading documents, splitting into chunks, generate embeddings , retrievers and storing in a vector DB. It loads external documents like person's resume or personal webpage, split it into smaller chunks, generate embeddings using any LLM and storing it in a vector store such as a Chroma vector DB.
2. RAG System: - This includes retrieving documents similar to the query, which incorporates the LLM prompt with these documents, and communicates with the LLM to obtain an accurate answer.
   We would cover these blocks in detail going ahead.

Data Ingestion

1. Document Loaders and Text Splitter

Used LangChain's document loaders to load files like PDFs, TXT, and DOCX. The text splitter breaks documents into smaller pieces that fit within the model's context window. Chosen *RecursiveCharacterTextSplitter* because it helps keep paragraphs, sentences, and words together, preserving their meaning.
Following is the code snippet

```
def langchain_document_loader():
"""
Create document loaders for PDF, TXT, DOCX and HTML files.
"""
documents = []
txt_loader = DirectoryLoader(
TMP_DIR, glob="**/*.txt", loader_cls=TextLoader, show_progress=True
)
documents.extend(txt_loader.load())
pdf_loader = DirectoryLoader(
TMP_DIR, glob="**/*.pdf", loader_cls=PyPDFLoader, show_progress=True
)
documents.extend(pdf_loader.load())
doc_loader = DirectoryLoader(
TMP_DIR,
glob="**/*.docx",
loader_cls=Docx2txtLoader,
show_progress=True,
)
documents.extend(doc_loader.load())
html_loader = DirectoryLoader(
TMP_DIR, glob="**/*.html", loader_cls=BSHTMLLoader, show_progress=True
)
documents.extend(html_loader.load())
htm_loader = DirectoryLoader(
TMP_DIR, glob="**/*.htm", loader_cls=BSHTMLLoader, show_progress=True
)
documents.extend(htm_loader.load())
return documents
```

2. Embeddings and Vectorstore

Moving forward the smaller chunks is converted into numeric embeddings to store it into a vectorstore DB. Vectorstore helps searching vectors that are most similar to the query's embeddings. Used google *embedding-001* model for the embeddings and chroma DB to store them.

3. Retrievers

Retrievers are used to return the relevant documents against a query. Worked on three retrievers which are as follows:

1. **Vectorstore-backed retriever**
   It uses semantic search to find documents in a Vectorstore with three search options:
   • **Similarity search**: Finds the top k similar results.
   • **MMR search**: Ensures both similarity and diversity.
   • **Similarity score threshold**: Sets the minimum relevance level.
   Filtering out irrelevant results improves the accuracy and cost-effectiveness of LLM calls.
   Following code snippet is used

```
def Vectorstore_backed_retriever(
vectorstore, search_type="similarity", k=4, score_threshold=None
):
"""create a vectorsore-backed retriever
Parameters:
search_type: Defines the type of search that the Retriever should perform.
Can be "similarity" (default), "mmr", or "similarity_score_threshold"
k: number of documents to return (Default: 4)
score_threshold: Minimum relevance threshold for
similarity_score_threshold (default=None)
"""
search_kwargs = {}
if k is not None:
search_kwargs["k"] = k
if score_threshold is not None:
search_kwargs["score_threshold"] = score_threshold
retriever = vectorstore.as_retriever(
search_type=search_type, search_kwargs=search_kwargs
)
return retriever
```

2. **Contextual Compression Retriever**
   It removes irrelevant info from documents. It first gets initial documents from the base retriever, then passes them through a Document Compressor, which reduces or removes irrelevant parts.
   The Document Compressor Pipeline contains the following steps:
   1. **Split** the initial documents into smaller chunks using CharacterTextSplitter.
   2. **Remove redundancies** with EmbeddingsRedundantFilter.
   3. **Filter** the most relevant chunks using EmbeddingsFilter with a similarity threshold and set k to 16.
   4. **Reorder** the chunks with LongContextReorder, putting the most relevant parts at the top and bottom.
      Following Code snippet is used

```
def create_compression_retriever(
embeddings, base_retriever, chunk_size=500, k=16,
similarity_threshold=None
):
# 1. splitting docs into smaller chunks
splitter = CharacterTextSplitter(
chunk_size=chunk_size, chunk_overlap=0, separator=". "
)
# 2. removing redundant documents
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)
# 3. filtering based on relevance to the query
relevant_filter = EmbeddingsFilter(
embeddings=embeddings, k=k, similarity_threshold=similarity_threshold
)
# 4. Reorder the documents
reordering = LongContextReorder()
# 5. create compressor pipeline and retriever
pipeline_compressor = DocumentCompressorPipeline(
transformers=[splitter, redundant_filter, relevant_filter, reordering]
)
compression_retriever = ContextualCompressionRetriever(
base_compressor=pipeline_compressor, base_retriever=base_retriever
)
return compression_retriever
```

3. **Cohere Reranker**
   This retriever rerank indexes the documents from most to least semantically relevant to the query.

```
def CohereRerank_retriever(
base_retriever, cohere_api_key, cohere_model="rerank-multilingual-v2.0",
top_n=10
):
compressor = CohereRerank(
cohere_api_key=cohere_api_key, model=cohere_model, top_n=top_n
)
retriever_Cohere = ContextualCompressionRetriever(
base_compressor=compressor, base_retriever=base_retriever
)
return retriever_Cohere
```

RAG System

1. **ChatModel**
   Used Google Generative AI models like gemini-pro and gemini-1.5-flash.
   The key parameters to these models are:
   - **Temperature**: Controls randomness in token selection. Higher values lead to more diversity and creativity. Selected lower values.
   - **Top_p**: Sets the cumulative probability cutoff for token selection. Higher values also increase diversity. Set it to 0.95 as it is industry norm
2. **Prompt Template**
   It is used to generate responses from LLM. Created two templates where former is used to generate a standalone question given the chat history and. follow-up question and later instructs the LLM to answer the question based solely on the provided context which is chat history and retrieved document.
   Prompts are:

```
condense_question_prompt = PromptTemplate(
input_variables=["chat_history", "question"],
template="""Given the following conversation and a follow up question,
rephrase the follow up question to be a standalone question, in its
original language.\n\n
Chat History:\n{chat_history}\n
Follow Up Input: {question}\n
Standalone question:""",
)
template = f"""Answer the question at the end, using only the following
context (delimited by <context></context>).
Your answer must be in the language at the end.
<context>
{{chat_history}}
{{context}}
</context>
Question: {{question}}
Language: {language}.
"""
```

3. **Memory**
   It stores the chat history, which is a simple buffer that last K interactions and summarizes the conversation.
   The `output_key` is set to `'answer'` and `input_key` to `'question'`, allowing the memory to track and save user questions and AI answers.

```
def create_memory():

"""Creates a ConversationBufferMemory for the LLM"""
memory = ConversationBufferMemory(
return_messages=True,
memory_key="chat_history",
output_key="answer",
input_key="question",
)
return memory
```

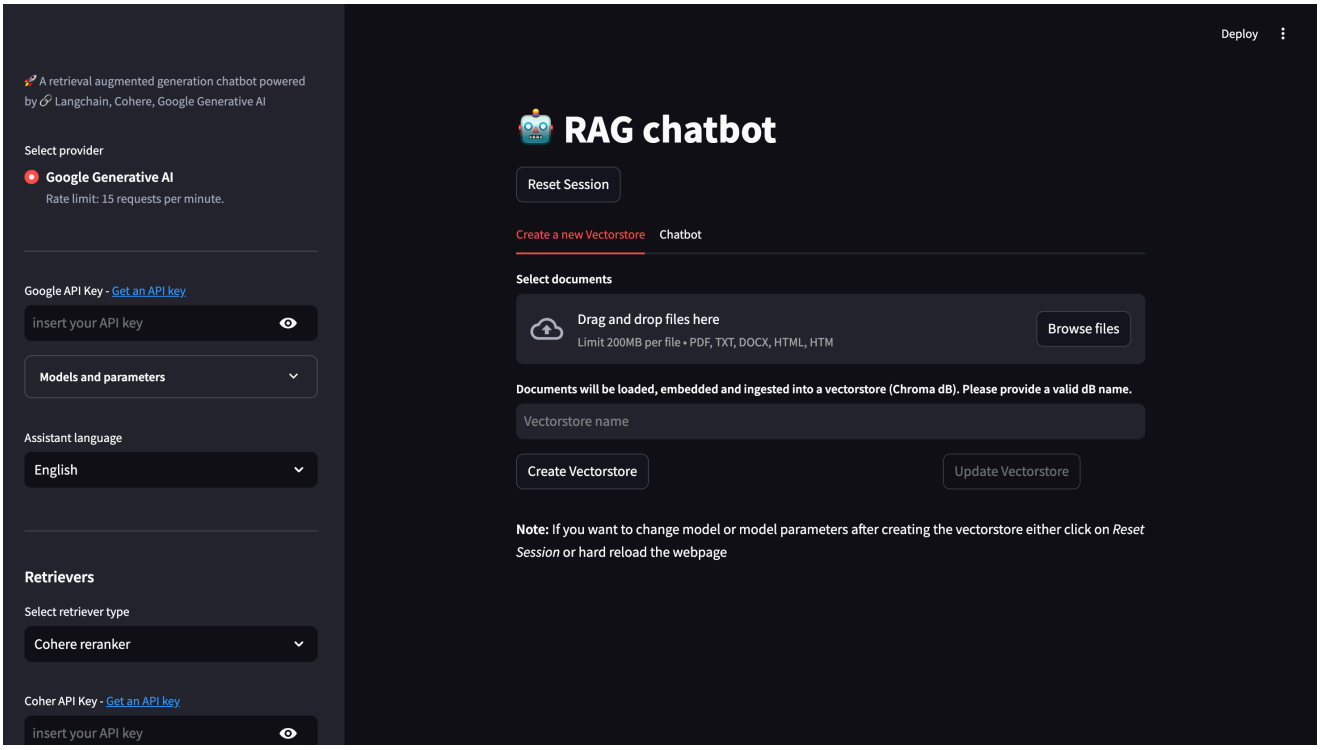
4. **Conversational Retrieval Chain**
   All the components Retriever, ChatModel LLM, Memory, and Prompt are used together in the ConversationalRetrievalChain.
   The ConversationalRetrievalChain is a built-in chain that connects these components, enabling to chat with the documents. It first sends the follow-up question and chat history to the LLM, which rephrases the question into a standalone query. The Retriever then finds relevant documents based on this query. Finally, the LLM uses these documents, along with the question and chat history, to provide an answer.
   Following is the code snippet

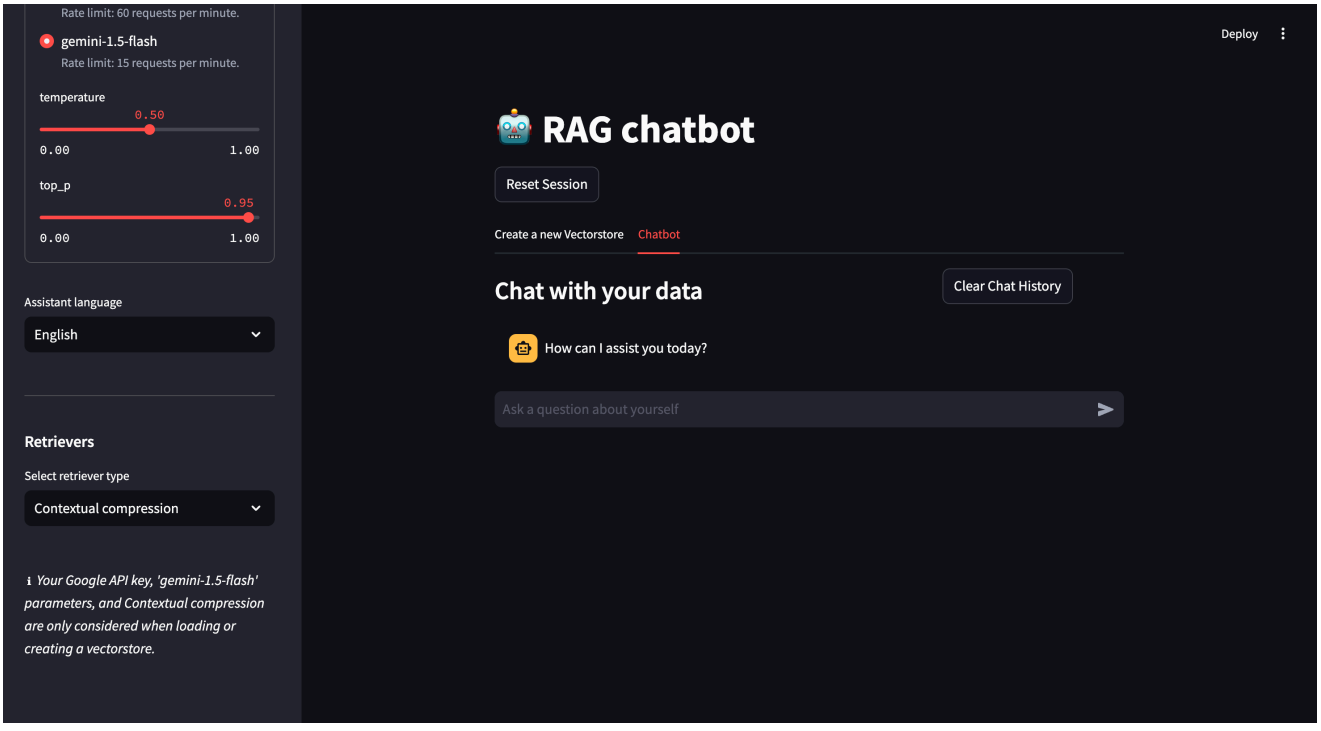
```
standalone_query_generation_llm = ChatGoogleGenerativeAI(
google_api_key=llm_api_key,
model=llm_selected_model,
temperature=0.1,
convert_system_message_to_human=True,)
response_generation_llm = ChatGoogleGenerativeAI(
google_api_key=llm_api_key,
model=llm_selected_model,
temperature=temperature,
top_p=top_p,
convert_system_message_to_human=True,
)
# 5. Create the ConversationalRetrievalChain
chain = ConversationalRetrievalChain.from_llm(
condense_question_prompt=condense_question_prompt,
combine_docs_chain_kwargs={"prompt": answer_prompt},
condense_question_llm=standalone_query_generation_llm,
llm=response_generation_llm,
memory=memory,
retriever=retriever,
chain_type=chain_type,
```

```
verbose=False,
return_source_documents=True,
)
```

Interface

Used streamlit to create UI so as to interact with the documents and get answers for the questions from the LLM.
Following are the snapshots from the UI:





Conclusion

Used different retrievers as experiments to evaluate the quality of the application. Cohere Ranker came out to be most effective followed by Contextual Compression Retriever and in last vectorstore backed retriever. Evaluated the quality of the application purely on the human instict.

Future Scope

Can use different LLMs like openai model, various free open source LLMs to get more effective results. Can work on a benchmark to evaluate the conversation based on different retrievers, model parameters and model.