CSE 5095 Natural Language Processing

SMART AI INTERVIEW COACH USING RAG

May 10, 2025

Maryam KafiKang
School of Computing
University of Connecticut
marykafi@uconn.edu

1 Abstract

This project introduces an AI-powered interview coach system using Retrieval-Augmented Generation (RAG). The system retrieves and generates real technical interview questions dynamically and evaluates candidate answers by providing scores and actionable feedback. It focuses on NLP, Deep Learning, and Machine Learning domains, with a dataset of 600 question–answer pairs.

A vector store was built using OpenAI and HuggingFace embeddings, indexed via ChromaDB for semantic search. GPT-4o-mini serves as the LLM for both question generation and answer evaluation. A Streamlit GUI interface enables users to interact through a conversational chat or a structured interview mode. Experiments tested the system's reasoning using synthetic knowledge about fictional models like LingoNet. Results show that the system can reason over added knowledge, retrieve correct context, and provide meaningful evaluation. Future work will extend the dataset and add adaptive questioning and voice interaction.

2 Introduction

This project aims to create an intelligent AI interview coach using Retrieval-Augmented Generation (RAG) to support technical interview preparation. The system is designed to ask domain-specific questions in NLP, Deep Learning, and Machine Learning, while also evaluating a candidate's answers with AI-generated feedback.

Traditional interview preparation lacks real-time feedback, personalized evaluation, and dynamic question adaptation. This project addresses these challenges by combining a retrieval system with a large language model to simulate a more interactive and adaptive interview process.

3 Method

This project implements a Retrieval-Augmented Generation (RAG) system to serve as an AI-powered interview coach. The system is designed to retrieve relevant interview questions from a pre-collected dataset, generate new interview questions, and evaluate candidate responses by providing feedback and scores. The implementation involved several key steps: data collection and preparation, embedding and indexing, retrieval and generation, and user interaction through a graphical interface.

The first step was data preparation. I collected a dataset of approximately 600 technical interview question–answer pairs by extracting text from various PDF resources and web scraping online interview question repositories. These pairs were cleaned, formatted, and saved into a JSON file. Each entry in the JSON contained a question and its corresponding

answer. To illustrate, Figure 1 shows an example of how the data was structured in JSON format, where each object represents a Q&A pair stored as key-value fields.

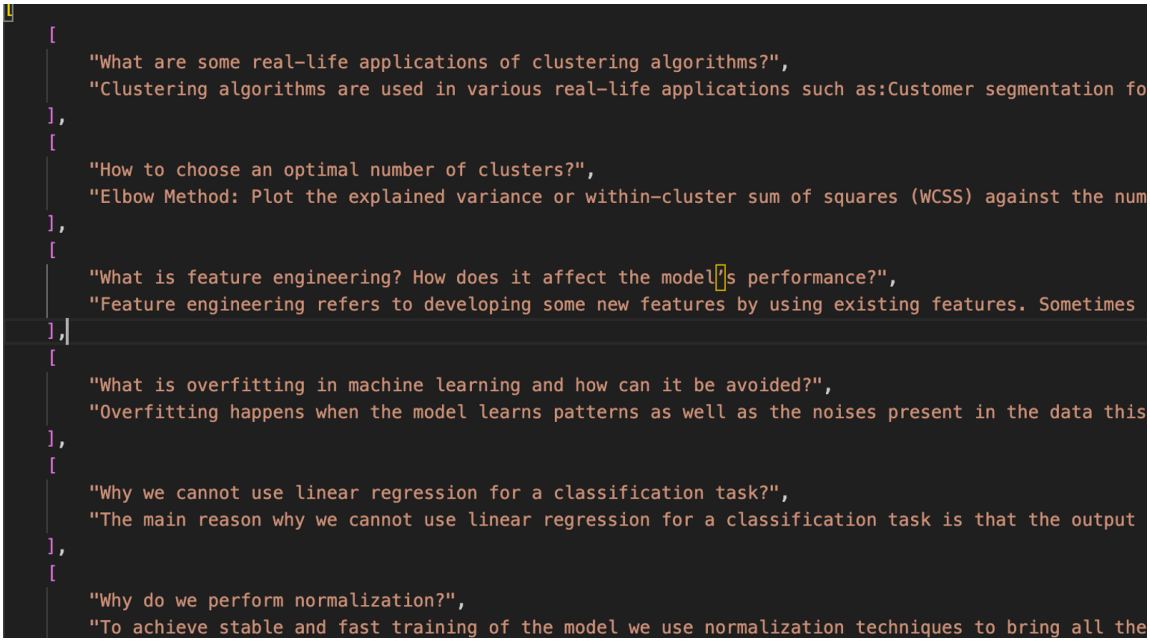

Figure 1: JSON data structure

Once the data was structured, it was converted into Document objects using LangChain's document model. Since some answers were lengthy, the documents were split into smaller chunks of 1,000 characters. This chunking process ensured that the retriever could work with more granular pieces of information and avoided exceeding the token limits of the language model.

After chunking, each text segment was embedded into a vector representation. I used OpenAI's text-embedding-3-small model to generate dense vector embeddings for each chunk, capturing the semantic meaning of the text. These embeddings were then indexed using ChromaDB, a vector database optimized for similarity search. Figure 2 presents the embedding and indexing pipeline, showing how raw text is transformed into vectors and stored in the vector database.

For the retrieval and generation workflow, I built two separate RAG chains using LangChain. The first chain focused on generating interview questions. When a user provided a topic (such as "NLP" or "Deep Learning"), the system retrieved the top 3 relevant question–answer chunks from the vector store and used GPT-4o-mini to generate a new interview question based on the retrieved context.

The second chain was responsible for evaluating candidate answers. After the system asked a question and the user submitted a response, it again retrieved supporting context

```python
with open("merged_data.json", "r", encoding="utf-8") as f:
    qa_data = json.load(f)

# Combine each Q&A pair into a single document
documents = []
for qa in qa_data:
    content = f"Question: {qa[0]}\nAnswer: {qa[1]}"
    documents.append(Document(page_content=content))

# 1. Configure a splitter that only subdivides if a chunk exceeds 1000 chars.
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,        # target <=1 000 chars per chunk
    chunk_overlap=200,      # keep 200 chars overlap for context
    length_function=len,    # measure by character count
)

# 2. Apply it—short docs (under 1000 chars) remain intact; long ones get split.
chunked_docs = text_splitter.split_documents(documents)

embedding_model = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = Chroma.from_documents(documents=chunked_docs, embedding=embedding_model)
retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": 3})
```

Figure 2: Diagram of embedding + indexing process

from the vector store. This context, along with the original question and the candidate's answer, was passed into GPT-4o-mini using a tailored prompt template. The model then returned a structured evaluation that included a score (from 1 to 10), detailed feedback, and a suggested model answer. Figure 3 illustrates this two-phase process.

To visualize the system as a whole, Figure 4 presents the high-level architecture of the RAG-based interview coach. The pipeline consists of the same two main phases—question generation and answer evaluation. In the first phase, the user initiates the interaction by requesting an interview question. The system retrieves semantically relevant data from the Chroma vector store, incorporates it into a prompt, and generates a question using GPT-4o-mini.

In the second phase, once the user answers the question, the system performs another round of retrieval and constructs a context-rich prompt containing the question, the user's response, and relevant supporting material. This is then evaluated by GPT-4o-mini to provide personalized feedback and scoring.

This architecture not only enables reasoning over the retrieved content but also supports multi-turn interaction. By maintaining chat history, the system generates follow-up questions and responses that reflect the full context of the conversation, creating a more realistic and

dynamic interview experience.

```
def evaluate_llm():
    contextualize_q_system_prompt = """You are an assistant for question-answering tasks. \
    Use the following pieces of retrieved context to answer the question. \
    If you don't know the answer, just say that you don't know. \
    Use three sentences maximum and keep the answer concise.{context}"""
    contextualize_q_prompt = ChatPromptTemplate.from_messages(
        [("system", contextualize_q_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}")])
    history_aware_retriever = create_history_aware_retriever(llm, retriever, contextualize_q_prompt)
    qa_system_prompt = """
        You are a senior technical interviewer assistant.

        Interview Question:
        {input}

        Candidate's Answer:
        {user_answer}

        Relevant Knowledge Base Excerpts:
        {context}
        Evaluate the candidate's answer based on the following criteria:
            a) Score the candidate's answer from 1 (poor) to 10 (excellent).
            b) Provide concise, actionable feedback on how to improve.
            c) Supply the "model" (ideal) answer.
        Format your response as:
            Score: <number>
            Feedback: <text>
            Model Answer: <text>

        Begin now.
        """
    qa_prompt = ChatPromptTemplate.from_messages(
        [("system", qa_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("system", "{input}"),
        ("human", "Candidate's Answer: {user_answer}")])
    question_answer_chain = create_stuff_documents_chain(llm, qa_prompt)
    rag_chain = create_retrieval_chain(history_aware_retriever, question_answer_chain)
    return rag_chain
```

Figure 3: Generate queestion and Evaluate User's answer

To allow users to interact with the system, I developed a web-based graphical user interface using Streamlit. The interface consists of two main tabs: a general chat mode for open-ended questions, and an interview simulation mode where the system generates an interview question, takes the user's answer, and provides evaluation feedback. The system also maintains a history of the conversation throughout the session. This chat history is passed to the language model at each step, allowing the model to generate context-aware

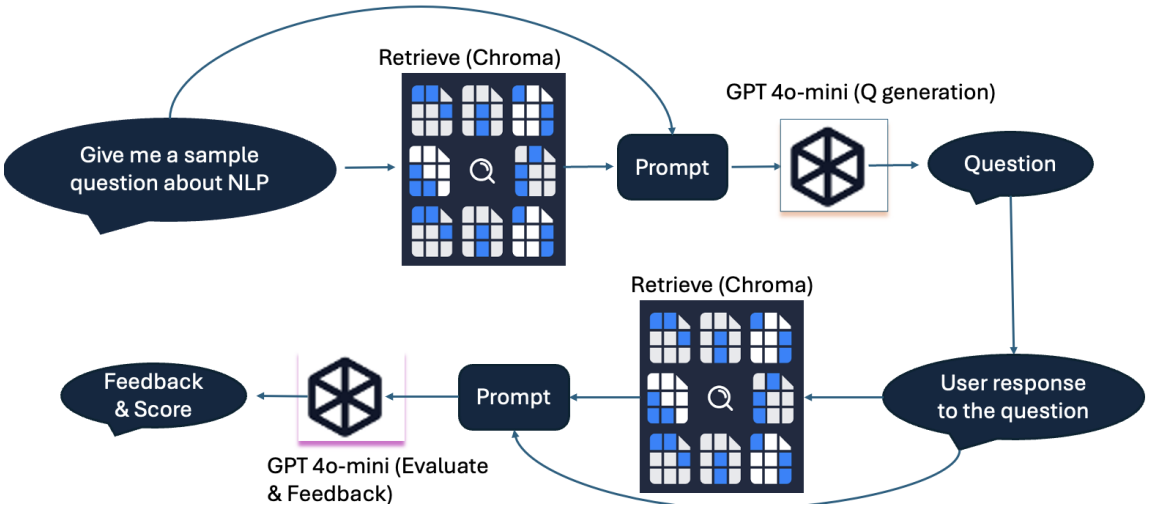

Figure 4: Model Architecture

responses and follow-up questions. The chat interface displays this history in a scrollable window, with a fixed input box at the bottom for a smooth and user-friendly experience.

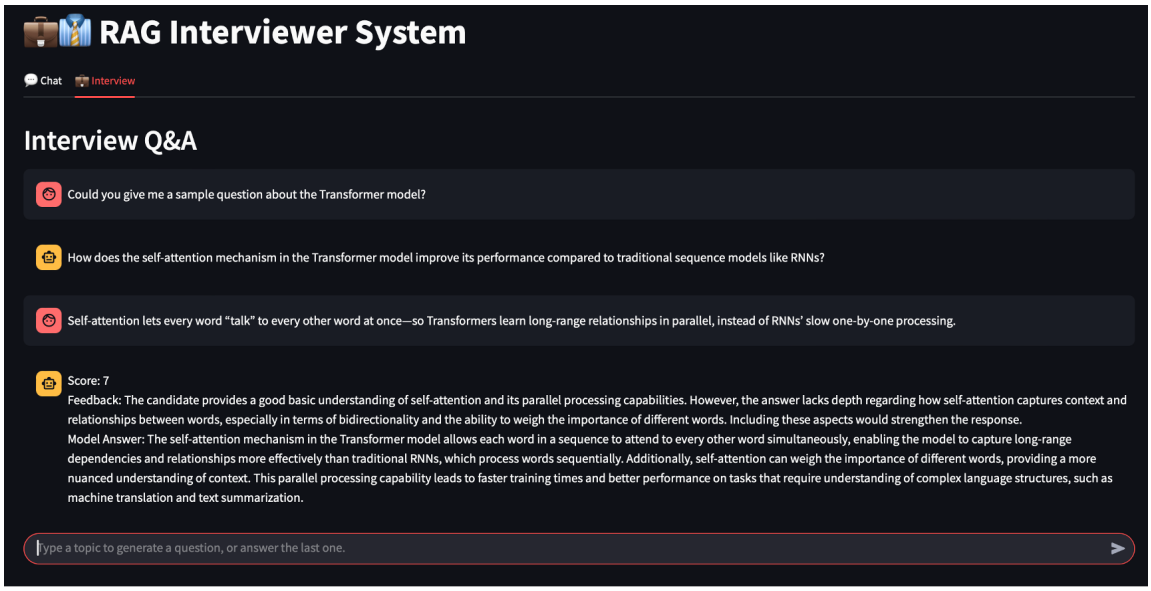

Figure 5: Interviewer mode

Throughout the implementation, LangChain's modules such as `create_history_aware_retriever`,

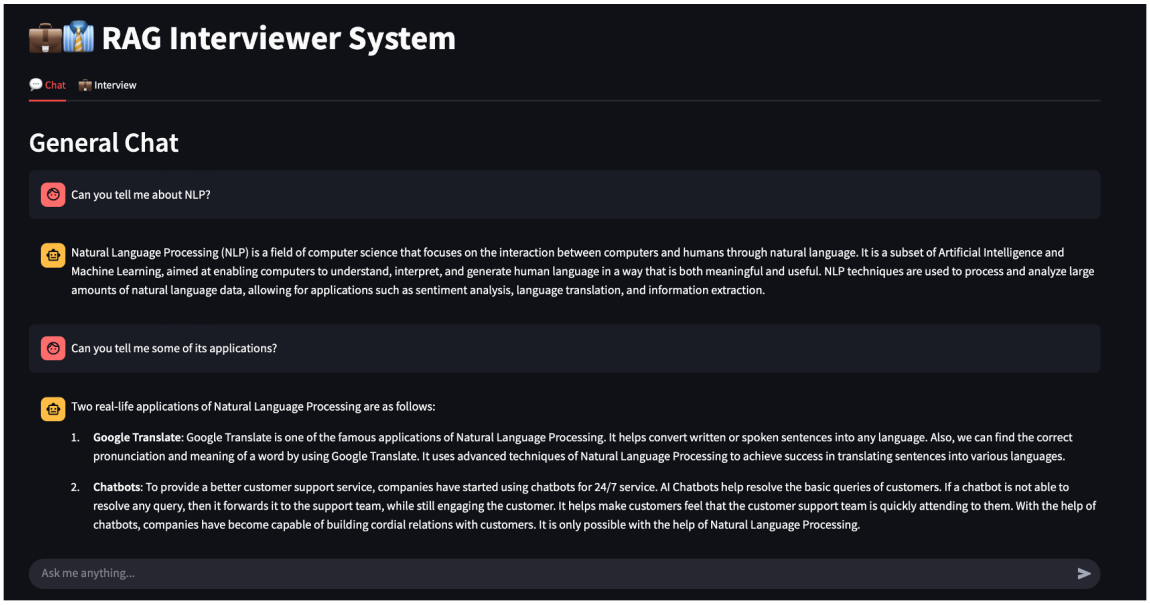

Figure 6: General Chat mode

`create_stuff_documents_chain`, and `create_retrieval_chain` were used to connect the retrieval and generation components. By combining these tools, the system supports interactive interview practice that dynamically generates questions, retrieves relevant knowledge, and evaluates answers using a large language model augmented with retrieved context.

All development was done on my laptop (Apple M1, 16 GB RAM, macOS). Although it lacks a discrete GPU, the system ran efficiently using OpenAI's API for embeddings and language model calls. The dataset, containing around 600 Q&A pairs collected from PDFs and web scraping, was stored in JSON format. ChromaDB indexing and retrieval were handled locally, allowing fast semantic search without high-end hardware.

4 Experimental Results and Discussion

The system was tested to evaluate its ability to retrieve relevant information and answer questions based on the indexed dataset. To specifically test the system's reasoning and inference capabilities, I added a set of synthetic Q&A pairs into the dataset about fictional language models named LingoNet and LingoPer. These models do not exist outside of this dataset, which allowed me to isolate the system's reliance on the retrieved information rather than any prior knowledge from the language model itself.

Figure **??** shows a snippet of the JSON data that was added to the system. This data included questions like "What is LingoNet?" and "What is LingoPer?" along with their

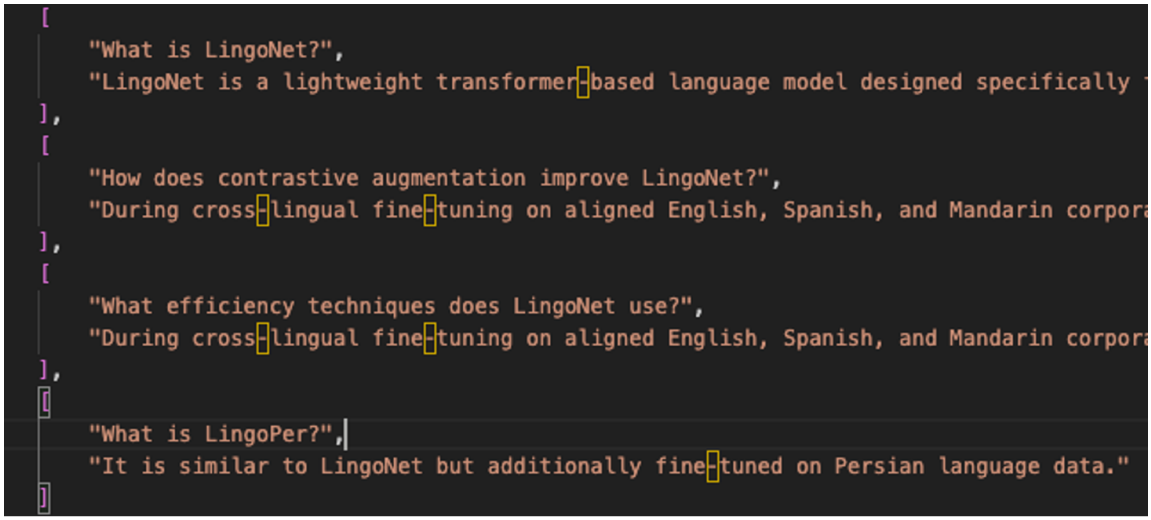

Figure 7: Fictional data

corresponding answers. By introducing this fictional data, I was able to test whether the system could retrieve and reason over the specific facts present only in the database.

When the system was queried through the general chat interface, as shown in Figure **??**, it successfully retrieved and presented the correct answers for questions such as "Can you tell me what LingoNet is?" and "Can you tell me what LingoPer is?" The system's responses accurately reflected the content stored in the JSON data, demonstrating that retrieval was functioning correctly and that the generation step was using the retrieved information to answer.

Interestingly, when asked a question like "Can you tell me other models that are like LingoNet?" the system responded with "I don't know." This is an expected outcome, since no such relation or answer was included in the indexed dataset, highlighting that the system answers are bounded by the retriever's outputs rather than hallucinating unsupported content from the language model.

Additionally, when a follow-up prompt like "Give more explanation about LingoPer please" is submitted, the system is able to generate an extended answer using the retrieved knowledge. This shows that the system can build on previous context to elaborate when prompted for more detail.

Overall, the system successfully retrieved the correct factual content from the vector index and generated accurate responses based on the retrieved information. This validates that the retrieval-augmented generation pipeline is effectively combining retrieval and generation steps, rather than relying solely on the language model's pretrained knowledge.

This experiment also demonstrates the interpretability of the system, because answers are tied to retrieved documents, it is possible to trace back the source of each answer,

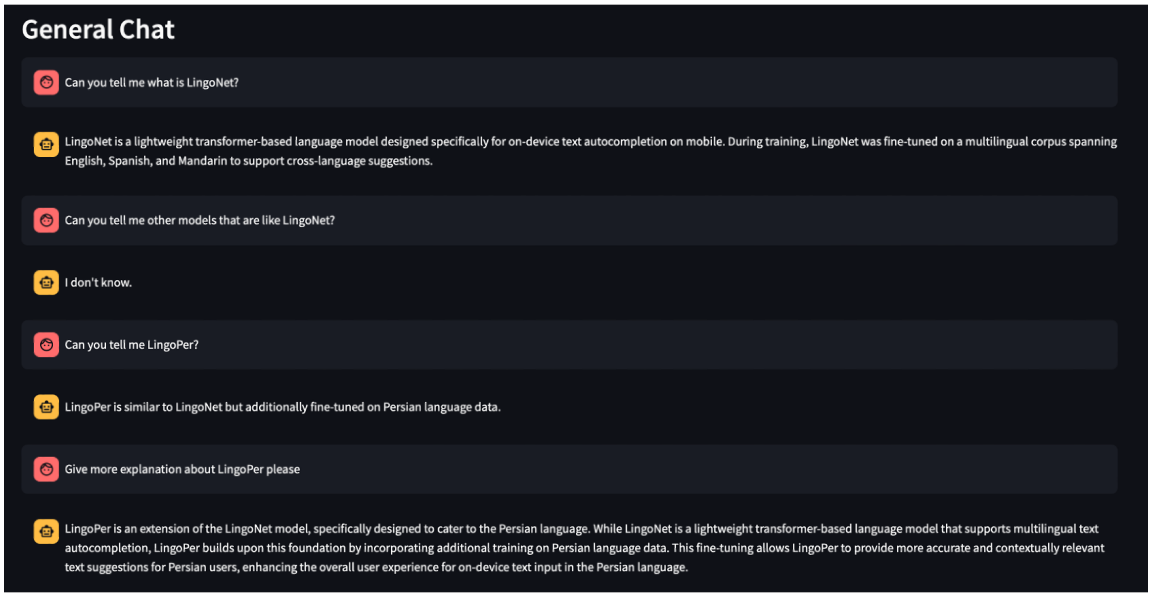

Figure 8: Inference Capability

increasing trustworthiness in the outputs.

5 Conclusion

This project demonstrates the potential of applying Retrieval-Augmented Generation (RAG) to create an intelligent, interactive interview coaching system. By combining a semantic retriever with a large language model, the system is able to generate domain-specific interview questions, evaluate candidate answers, and deliver real-time feedback through an easy-to-use Streamlit GUI interface.

The results show that the system can reason over custom added knowledge, retrieve relevant information accurately, and provide actionable, structured feedback to help candidates improve their answers. It also successfully simulates a dynamic interview experience by supporting follow-up questions and adaptive interaction.

For future development, I plan to extend the system by incorporating voice input and output, enabling users to practice answering questions verbally. I also aim to add job-specific interview customization, where questions are tailored based on a candidate's target role or job description. Another planned improvement is adaptive question difficulty, allowing the system to automatically adjust question complexity based on the user's performance over time. These enhancements will make the system more interactive, personalized, and closer to a real-world interview experience.

6 Jupyter Notebook/ Python code

The implementation of the RAG interview system was developed in Python using Jupyter Notebook for prototyping, testing, and visualization. The code integrates LangChain, OpenAI, HuggingFace, and ChromaDB libraries to build the retrieval and generation pipelines. Additional scripts were written to process the dataset, generate embeddings, and create the vector index. A Streamlit app was also developed to provide an interactive user interface for question generation, answer evaluation, and feedback.

All source code, including the Jupyter Notebook and Streamlit application, have been submitted as part of the project materials.