Sri Lanka Institute of Information Technology

Current Trends in Software Engineering - SE4010

Assignment II

Details

Name	**Reg. Number**
SHABEER M.S.M	IT21304088

TABLE OF CONTENTS

1. INTRODUCTION 1
2. JUSTIFICATION OF LLM CHOICE 2
3. JUSTIFICATION OF DEVELOPMENT APPROACH 4
3.1. Rag Based Application 4
3.2. Jupyter Notebook with Google Colab 4
3.3. Open-Source Tools and Tech 5
4. SYSTEM DESIGN 6
4.1. Data Ingestion and Preprocessing 7
4.2. Embedding and Vector Storage 9
4.3. Language Model Setup 11
4.4. RAG Pipeline Implementation 12
4.5. User Interaction and Chats 13
5. CHALLENGES AND LESSONS LEARNED 14
6. USAGE OF GENAI TOOLS 15
7. CONCLUSION 17
8. APPENDICE 18
8.1. GitHub Repo Link 18
8.2. Demo Video Link 18

1. INTRODUCTION

This project report presents the development of a Retrieval-Augmented Generation (RAG)-based chatbot designed to assist students by answering questions from CTSE lecture notes. Implemented in Google Colab using a Jupyter Notebook, the system extracts text from lecture PDFs, converts it into vector embeddings, and stores them in a vector database. When a query is made, the system retrieves relevant chunks and uses a language model to generate accurate, context-aware responses.

By combining traditional retrieval techniques with a generative language model, the RAG architecture overcomes the limitations of static LLMs. This hybrid setup ensures responses are both grounded in course content and linguistically coherent. The report outlines the system’s architecture, tool choices, development process, challenges encountered, and the role of generative AI tools, highlighting the solution’s practical value and scalability.

2. JUSTIFICATION OF LLM CHOICE

The LLM selected for this chatbot is **google/gemini-2.0-flash**, a member of Google’s Gemini family known for high performance and efficiency. Built on the same foundational research as Gemini Pro, the Flash variant is specifically optimized for speed and low-latency inference, making it ideal for real-time applications and resource-constrained environments.

Gemini Flash is a **decoder-based, text-to-text model** designed to handle a variety of natural language tasks such as question answering, summarization, reasoning, and dialogue generation. Delivered through Google's Generative AI platform via an API, it removes the need for complex local deployment or high-end hardware.

A key advantage of this model is its **instruction-tuned nature**, allowing it to follow structured prompts accurately and generate concise, contextually relevant responses. These qualities are crucial for a system aimed at delivering reliable answers grounded in academic content. The model's performance, accessibility, and efficiency were primary considerations during selection.

- **Low Latency and High Efficiency**: Specifically optimized for fast, real-time responses, making it ideal for chat-based applications running in constrained environments.
- **Instruction-Tuned Precision:** Its ability to follow structured prompts ensures that responses are not only relevant but also factually accurate critical requirement for educational tools relying on precise content from academic material.
- **Simplified Deployment via API:** Accessing the model through Google’s Generative AI API eliminates the need for infrastructure setup, such as GPU provisioning or model hosting, enabling seamless integration within lightweight environments like Google Colab.
- **Resource-Conscious Design:** Offering strong performance at a lower cost, Gemini Flash serves as a practical alternative to heavier models like GPT-4 or Gemini Pro, making it a cost-effective solution for students and research projects.

Table 1.0 LLMs Comparison

Model	**Parameters**	**Instruction Tuned**	**Deployment Method**	**Memory Footprint**	**Licensing**	**Ideal Use Case**
Gemini-2.0-Flash	Not Disclosed	Yes	API (Google Gen AI)	Low (API - managed)	Proprietary (Google)	Real-time chatbot, fast inference in cloud environments
Gemini Pro	Not Disclosed	Yes	API (Google Gen AI)	Low (API - managed)	Proprietary (Google)	Real-time chatbot, fast inference in cloud environments

3. JUSTIFICATION OF DEVELOPMENT APPROACH

The chatbot was developed using a Retrieval-Augmented Generation (RAG) architecture within a Jupyter Notebook on Google Colab. This setup leverages open-source tools and models to provide a practical, resource-efficient, and easily accessible solution. The design prioritizes educational usability while maintaining technical reliability, making it ideal for student-driven research and learning environments.

3.1. Rag Based Application

The system enhances language model outputs by grounding responses in externally retrieved context. It consists of two main components: a **retriever** that finds relevant document chunks, and a **generator** that formulates responses based on the retrieved context:

- **Contextual Accuracy**: RAG systems retrieve relevant information from a custom knowledge base (e.g., embedded lecture notes), ensuring responses are accurate, grounded, and reliable, minimizing hallucinations ideal for educational use.
- **Scalability**: Chroma, a fast and scalable vector store, enables efficient similarity-based text retrieval, ensuring quick responses and consistent performance even as more documents are added, supporting seamless scalability.
- **Modularity**: RAG's modular design separates retrieval and generation, allowing independent updates or replacements of components like the embedding model or LLM, enhancing maintainability and supporting future experimentation.

3.2. Jupyter Notebook with Google Colab

The implementation is carried out using a Jupyter Notebook hosted on Google Colab, chosen for its accessibility, interactivity, and computational support.

- **Interactive Development**: Jupyter Notebooks offer an interactive, cell-by-cell development environment, ideal for quick testing, visualizing outputs, and debugging, which suits educational and experimental projects.

- **Cloud-Based Accessibility**: Google Colab provides free access to CPU and GPU resources, making it an excellent option for running heavy models and computations without relying on local hardware, especially beneficial for those with limited computational resources.
- **Persistent Storage with Google Drive**: Integration with Google Drive allows persistent storage of project files (e.g., Chroma vector database, PDF lecture notes), ensuring accessibility across sessions and facilitating collaborative work without the risk of data loss.

3.3. Open-Source Tools and Tech

This project relies fully on open-source tools, ensuring transparency, low cost, and ease of learning. Their active community support makes them suitable for quick development and long-term use.

- **LangChain**: Manages the orchestration of the RAG pipeline, simplifying the integration of loaders, retrievers, and LLMs.
- **Chroma Vector DB**: A lightweight, easy-to-use vector store optimized for local and small-scale deployments.
- **Google Generative AI and GoogleGenerativeAIEmbeddings** : Provide access to cutting-edge LLMs and embedding models, such as *google/gemini-2.0- flash* and *embedding-001*, with seamless APIs.
- **PyPDFLoader**: A utility that extracts text from PDF files, making it easy to load and preprocess academic documents for use in NLP applications.
- **RecursiveCharacterTextSplitter**: Splits large chunks of text into smaller, manageable segments while preserving semantic coherence—crucial for effective embedding and retrieval in RAG systems.
- **gdown**: Simplify document handling, preprocessing, and data loading from Google Drive.

4. SYSTEM DESIGN

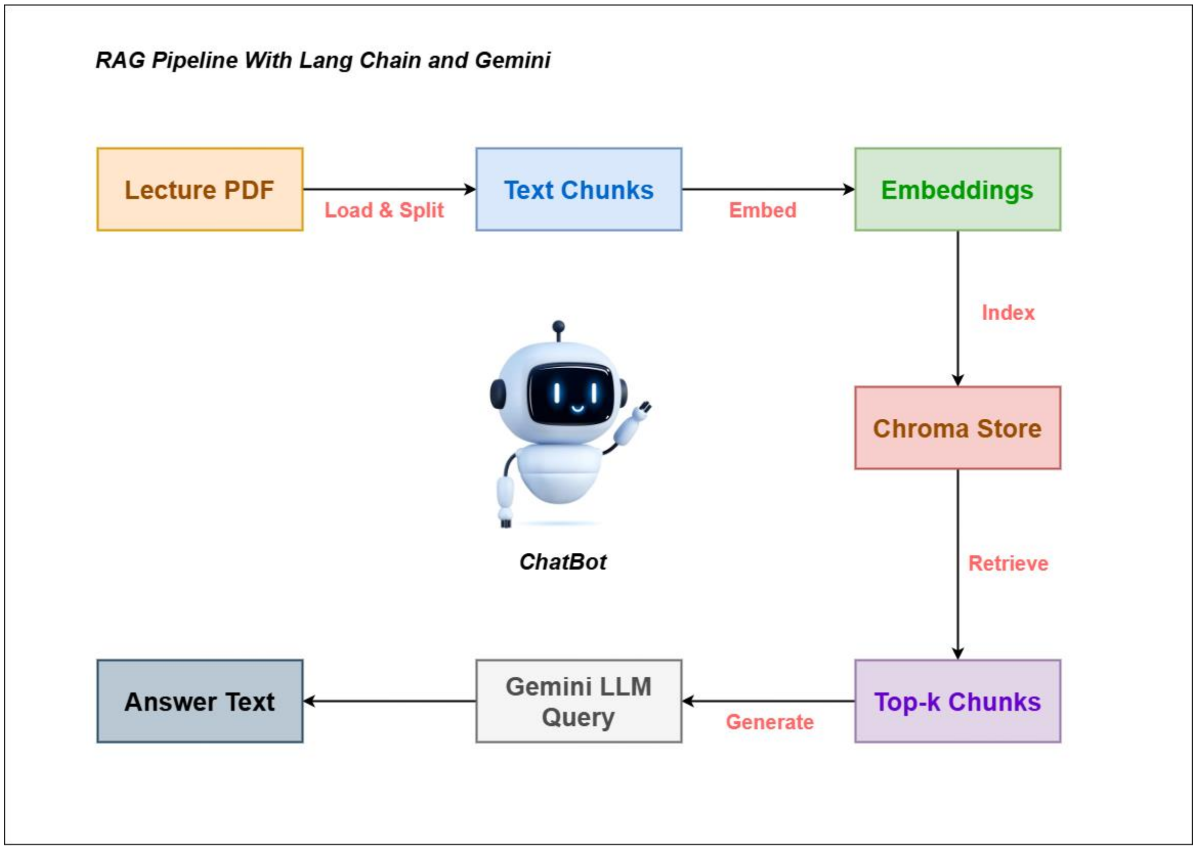
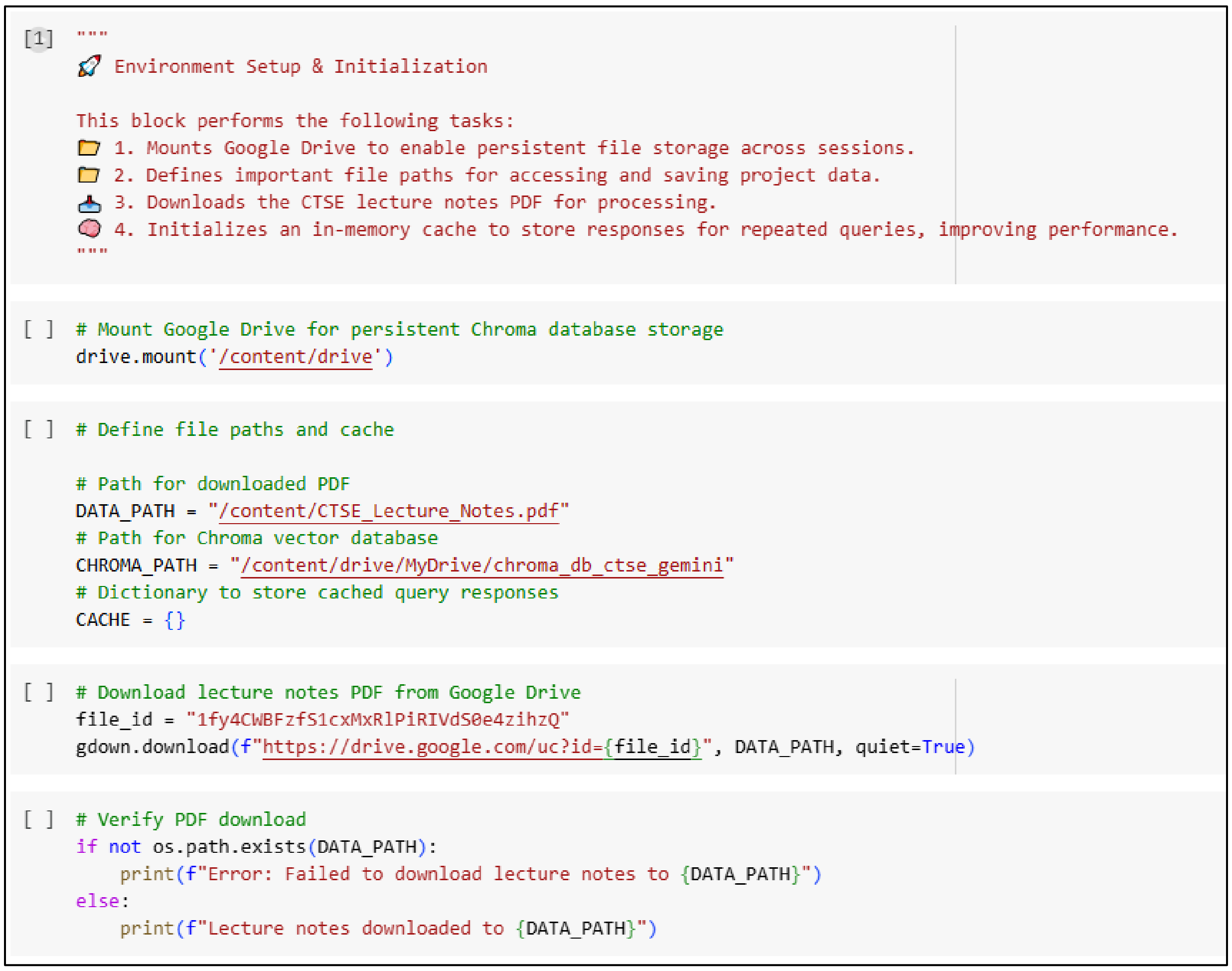


Figure 1.0 Chatbot System Diagram

The chatbot is architected as a modular pipeline that systematically converts static lecture notes into a dynamic and intelligent question-answering system. The entire pipeline is composed of distinct, well-defined stages each responsible for a specific task, beginning with document ingestion, followed by text preprocessing, embedding generation, context retrieval, and finally, response generation. This step-by-step, component-based design not only enhances the system's clarity and organization but also enables greater flexibility in development and maintenance. By isolating each function within its own module, developers can easily troubleshoot issues, optimize performance, and upgrade or replace individual components such as switching embedding models or fine-tuning the language model without disrupting the functionality of the overall system. This approach supports rapid experimentation, continuous improvement, and scalability, making it ideal for academic and research-oriented applications.

4.1. Data Ingestion and Preprocessing

Convert a PDF of lecture notes into manageable, searchable text units. A PDF file containing CTSE lecture notes is downloaded from Google Drive using the gdown utility, enabling seamless integration with cloud storage.



```
[1] """
    🚀 Environment Setup & Initialization

    This block performs the following tasks:
    📁 1. Mounts Google Drive to enable persistent file storage across sessions.
    📂 2. Defines important file paths for accessing and saving project data.
    📥 3. Downloads the CTSE lecture notes PDF for processing.
    🧠 4. Initializes an in-memory cache to store responses for repeated queries, improving performance.
    """
```

```
[ ] # Mount Google Drive for persistent Chroma database storage
    drive.mount('/content/drive')
```

```
[ ] # Define file paths and cache

    # Path for downloaded PDF
    DATA_PATH = "/content/CTSE_Lecture_Notes.pdf"
    # Path for Chroma vector database
    CHROMA_PATH = "/content/drive/MyDrive/chroma_db_ctse_gemini"
    # Dictionary to store cached query responses
    CACHE = {}
```

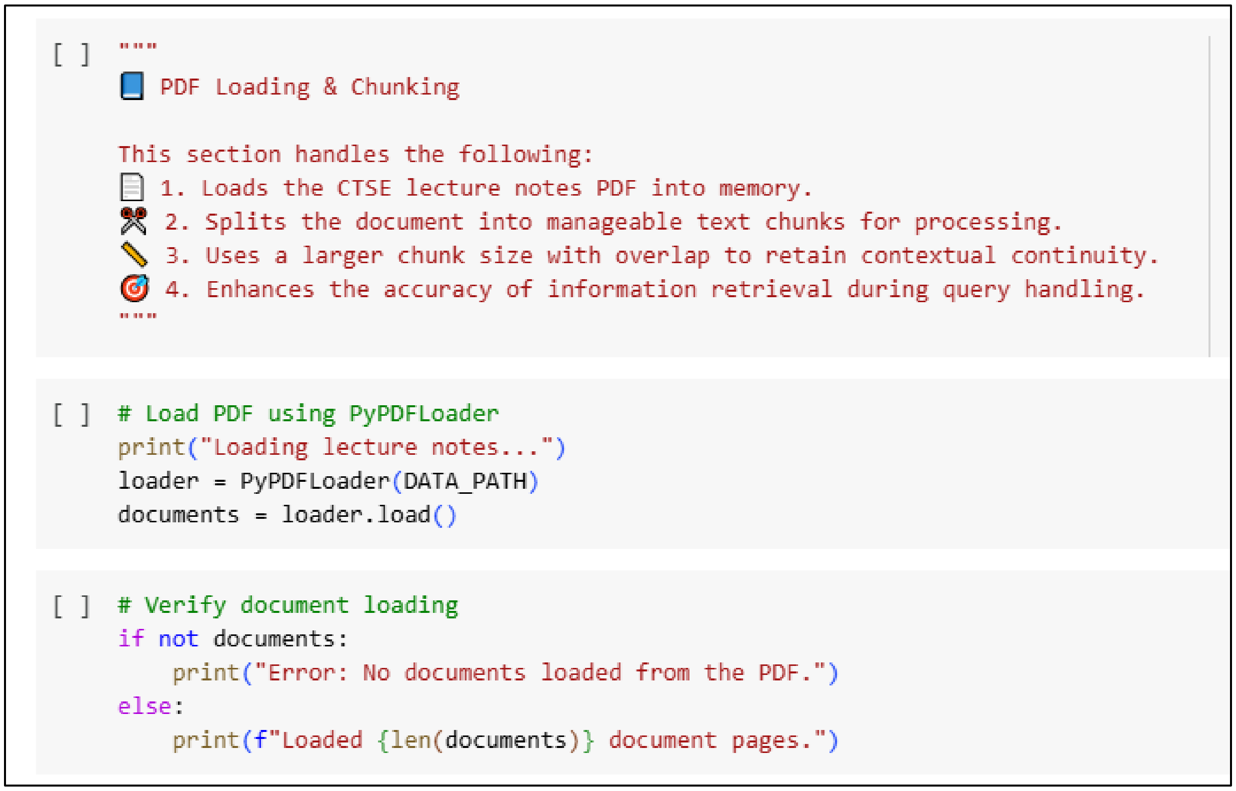
```
[ ] # Download lecture notes PDF from Google Drive
    file_id = "1fy4CWBFzfS1cxMxRlPiRIVdS0e4zihzQ"
    gdown.download(f"https://drive.google.com/uc?id={file_id}", DATA_PATH, quiet=True)
```

```
[ ] # Verify PDF download
    if not os.path.exists(DATA_PATH):
        print(f"Error: Failed to download lecture notes to {DATA_PATH}")
    else:
        print(f"Lecture notes downloaded to {DATA_PATH}")
```

Figure 2.0 Lecture Notes PDF Downloading

Text Extraction

PyPDFLoader (from LangChain) is utilized to extract raw text from PDFs, maintaining the integrity of document structure by preserving headers, paragraphs, and other elements in a clean textual format.



```
[ ] """
    📘 PDF Loading & Chunking

    This section handles the following:
    📄 1. Loads the CTSE lecture notes PDF into memory.
    ✂️ 2. Splits the document into manageable text chunks for processing.
    📏 3. Uses a larger chunk size with overlap to retain contextual continuity.
    🎯 4. Enhances the accuracy of information retrieval during query handling.
    """
```

```
[ ] # Load PDF using PyPDFLoader
    print("Loading lecture notes...")
    loader = PyPDFLoader(DATA_PATH)
    documents = loader.load()
```

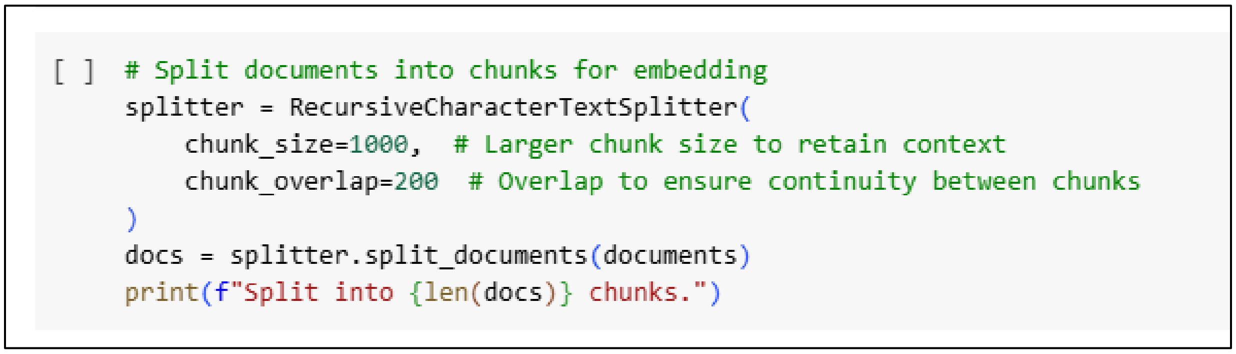
```
[ ] # Verify document loading
    if not documents:
        print("Error: No documents loaded from the PDF.")
    else:
        print(f"Loaded {len(documents)} document pages.")
```

Figure 3.0 Text Extraction

Text Chunking

The extracted text is segmented into chunks using the ***RecursiveCharacterTextSplitter***. Each chunk is around 1000 characters long, with a 200-character overlap between consecutive chunks.

Note: - Why? Larger chunks help retain important context (such as paragraphs or examples), while the overlap ensures that critical boundary information is preserved, enhancing the relevance of answers during retrieval.



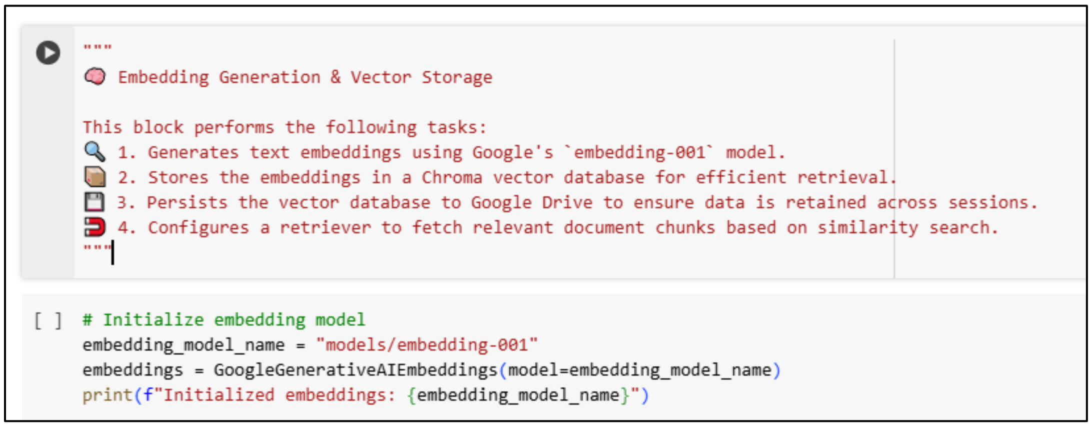
```
[ ] # Split documents into chunks for embedding
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,  # Larger chunk size to retain context
        chunk_overlap=200  # Overlap to ensure continuity between chunks
    )
    docs = splitter.split_documents(documents)
    print(f"Split into {len(docs)} chunks.")
```

Figure 4.0 Text Chunking

4.2. Embedding and Vector Storage

The text chunks are converted into numerical vectors that capture their semantic meaning, allowing for more accurate comparisons and retrieval. These vectors represent the underlying concepts and relationships within the text, enabling efficient storage and quick retrieval based on the similarity of content. By storing these vectors, the system can rapidly match and return relevant chunks during a query, improving the overall retrieval process.

- **Embedding Model: -** The chatbot utilizes embedding-001, a pre-trained Google model designed for semantic similarity, to convert each text chunk into a high-dimensional vector that represents its underlying meaning.



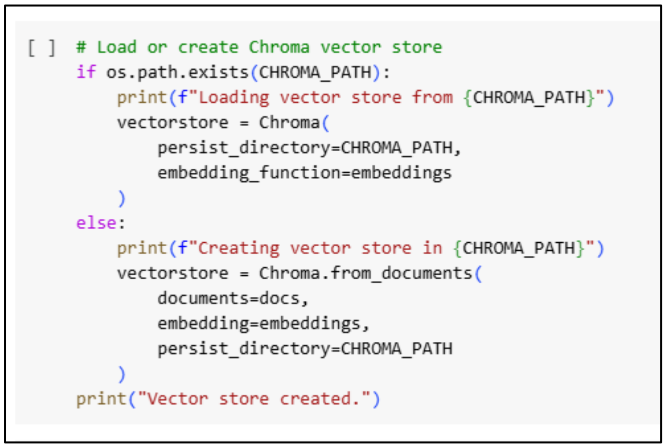
```
"""
🧠 Embedding Generation & Vector Storage

This block performs the following tasks:
🔍 1. Generates text embeddings using Google's `embedding-001` model.
📦 2. Stores the embeddings in a Chroma vector database for efficient retrieval.
💾 3. Persists the vector database to Google Drive to ensure data is retained across sessions.
🧲 4. Configures a retriever to fetch relevant document chunks based on similarity search.
"""
```

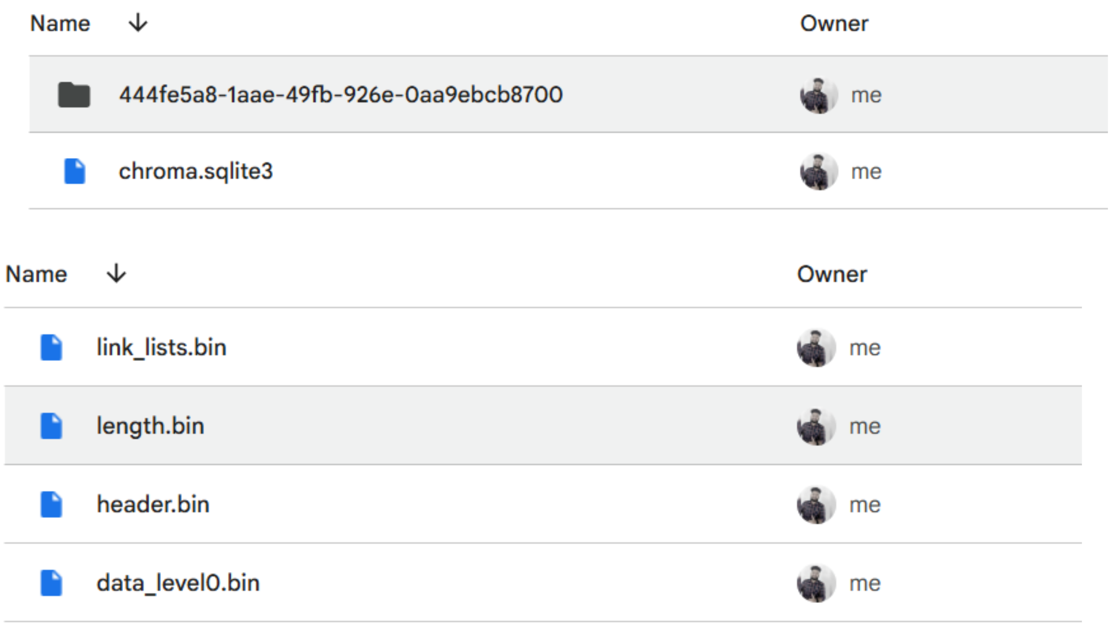
```
[ ] # Initialize embedding model
    embedding_model_name = "models/embedding-001"
    embeddings = GoogleGenerativeAIEmbeddings(model=embedding_model_name)
    print(f"Initialized embeddings: {embedding_model_name}")
```

Figure 5.0 Embedding Model Initialization

- **Vector Database: -** The vectors are stored in Chroma, an open-source vector database optimized for similarity search, and persisted in Google Drive, enabling the chatbot to retain knowledge across sessions.



```
[ ]  # Load or create Chroma vector store
     if os.path.exists(CHROMA_PATH):
         print(f"Loading vector store from {CHROMA_PATH}")
         vectorstore = Chroma(
             persist_directory=CHROMA_PATH,
             embedding_function=embeddings
         )
     else:
         print(f"Creating vector store in {CHROMA_PATH}")
         vectorstore = Chroma.from_documents(
             documents=docs,
             embedding=embeddings,
             persist_directory=CHROMA_PATH
         )
     print("Vector store created.")
```

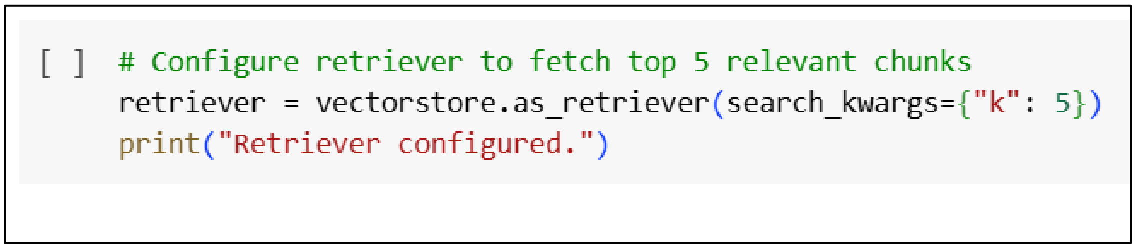


Name ↓	Owner
444fe5a8-1aae-49fb-926e-0aa9ebcb8700	me
chroma.sqlite3	me

Name ↓	Owner
link_lists.bin	me
length.bin	me
header.bin	me
data_level0.bin	me

Figure 6.0 Chroma Database Files in Google Drive

- **Retriever Configuration: -** During query processing, the retriever selects the top 5 most relevant chunks using vector similarity, ensuring the language model has access to context that is both pertinent and well-supported.



```
[ ]  # Configure retriever to fetch top 5 relevant chunks
     retriever = vectorstore.as_retriever(search_kwargs={"k": 5})
     print("Retriever configured.")
```

Figure 7.0 Retriever Configuration

4.3. Language Model Setup

Generate human-like, informative answers based on retrieved context.

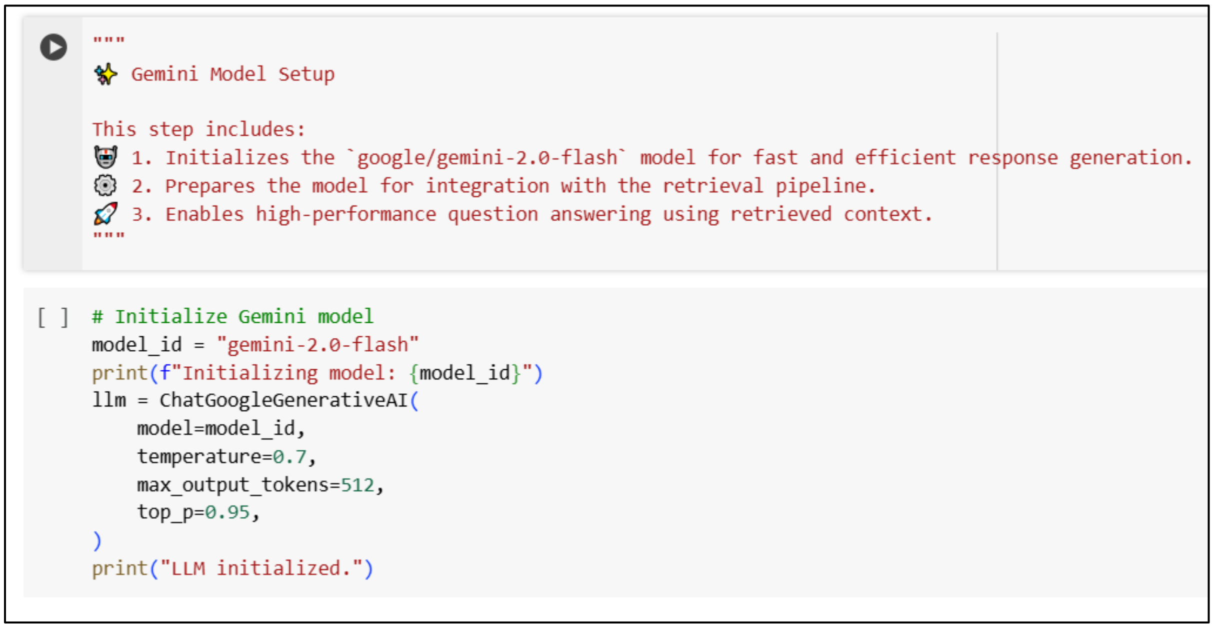
Choice Of the Modal

The chatbot uses *google/gemini-2.0-flash*, an instruction-tuned LLM offered by google.The model is integrated using Google's Generative AI API, authenticated with an API key securely input via getpass.

LangChain Integration

The model is wrapped in ***ChatGoogleGenerativeAI*** for compatibility with LangChain's RAG framework with the following parameters:

- temperature=0.7: Adds controlled randomness for natural language variation.
- max_output_tokens=512: Limits the length of generated responses.
- top_p=0.95: Enables nucleus sampling to improve diversity.



```
"""
✨ Gemini Model Setup

This step includes:
🤖 1. Initializes the `google/gemini-2.0-flash` model for fast and efficient response generation.
⚙️ 2. Prepares the model for integration with the retrieval pipeline.
🚀 3. Enables high-performance question answering using retrieved context.
"""
```

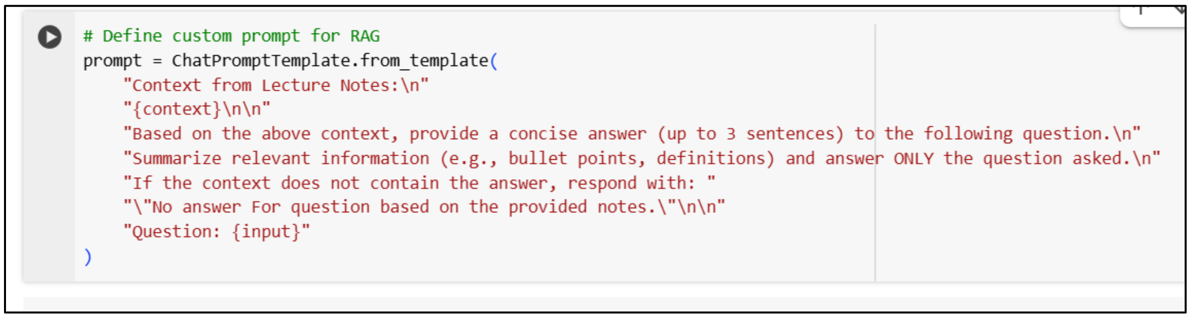
```
# Initialize Gemini model
model_id = "gemini-2.0-flash"
print(f"Initializing model: {model_id}")
llm = ChatGoogleGenerativeAI(
    model=model_id,
    temperature=0.7,
    max_output_tokens=512,
    top_p=0.95,
)
print("LLM initialized.")
```

Figure 8.0 LangChain Integration

4.4. RAG Pipeline Implementation

Combine document retrieval and language generation into a seamless question-answering workflow. A custom prompt template guides the LLM to generate concise, context-based answers (maximum 3 sentences).

- If the retrieved context lacks the relevant information, the model is instructed to respond with: "No answer for question based on the provided notes."
- This design minimizes hallucination and ensures responses remain grounded in course material.



```
# Define custom prompt for RAG
prompt = ChatPromptTemplate.from_template(
    "Context from Lecture Notes:\n"
    "{context}\n\n"
    "Based on the above context, provide a concise answer (up to 3 sentences) to the following question.\n"
    "Summarize relevant information (e.g., bullet points, definitions) and answer ONLY the question asked.\n"
    "If the context does not contain the answer, respond with: "
    "\"No answer For question based on the provided notes.\"\n\n"
    "Question: {input}"
)
```

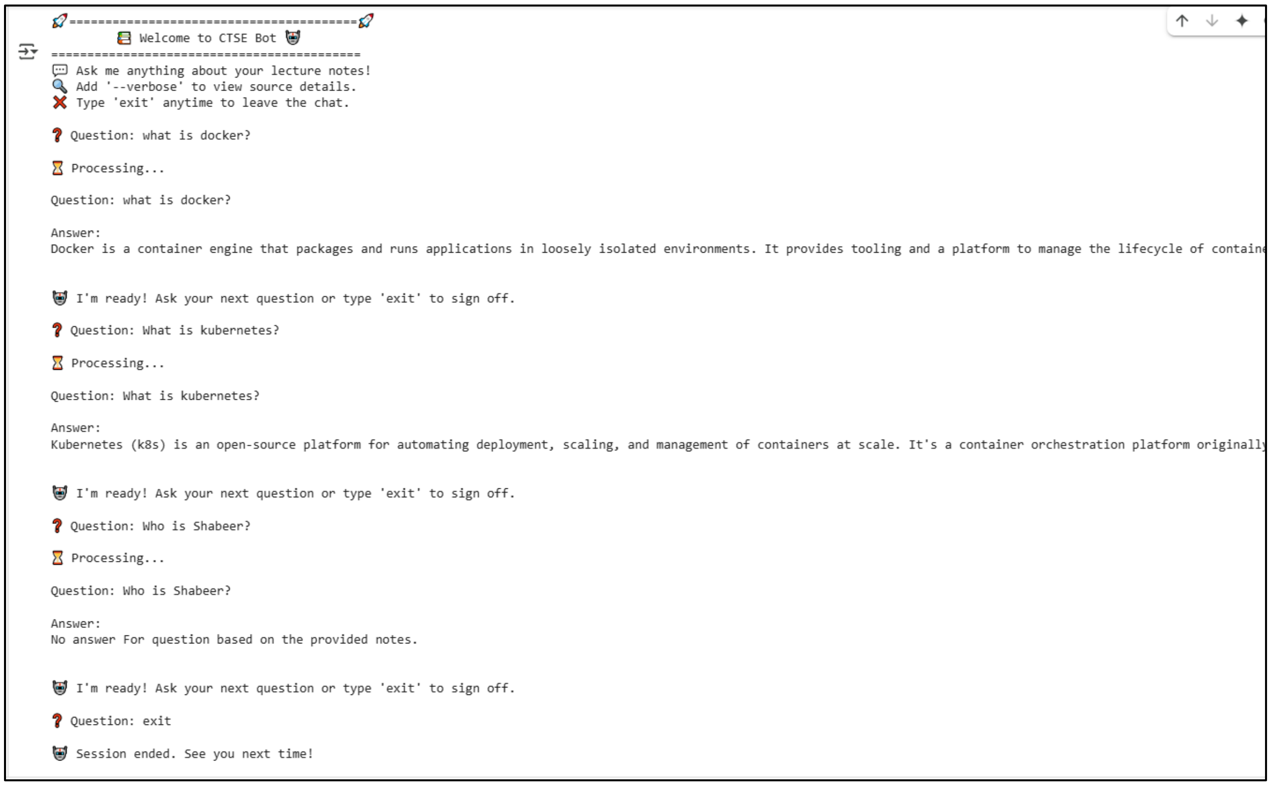
Figure 9.0 Prompt Template

Document Processing Chain & Retrieval Chain

The ***create_stuff_documents_chain*** function composes retrieved chunks into a single context block for the LLM to reference during generation. The ***create_retrieval_chain*** function links the retriever and the LLM, forming the core RAG pipeline that processes queries end-to-end

4.5. User Interaction and Chats

The chatbot provides an intuitive and interactive interface for users to ask questions and receive accurate, well-formatted answers. It operates within a continuous chat loop using a while statement and input() to accept user queries, with an optional verbose mode that displays the retrieved source documents for added transparency and easier debugging. The system enhances readability by formatting responses in an interactive manner, making the output clear and user-friendly. Additionally, basic error handling is built in to gracefully manage empty inputs, issues with document loading, and model-related exceptions, ensuring a smooth and reliable user experience throughout the interaction.



```
🚀=======================================🚀
        📚 Welcome to CTSE Bot 🤖
=========================================
💬 Ask me anything about your lecture notes!
🔍 Add '--verbose' to view source details.
❌ Type 'exit' anytime to leave the chat.

❓ Question: what is docker?

⏳ Processing...

Question: what is docker?

Answer:
Docker is a container engine that packages and runs applications in loosely isolated environments. It provides tooling and a platform to manage the lifecycle of containe


🤖 I'm ready! Ask your next question or type 'exit' to sign off.

❓ Question: What is kubernetes?

⏳ Processing...

Question: What is kubernetes?

Answer:
Kubernetes (k8s) is an open-source platform for automating deployment, scaling, and management of containers at scale. It's a container orchestration platform originally


🤖 I'm ready! Ask your next question or type 'exit' to sign off.

❓ Question: Who is Shabeer?

⏳ Processing...

Question: Who is Shabeer?

Answer:
No answer For question based on the provided notes.


🤖 I'm ready! Ask your next question or type 'exit' to sign off.

❓ Question: exit

🤖 Session ended. See you next time!
```

Figure 10.0 Chatbot Outputs

5. CHALLENGES AND LESSONS LEARNED

1. **Environment Configuration Pitfalls**
   - **Challenge: -** Initial setup of paths and environment variables caused file-not-found errors and inconsistent access to Google Drive.
   - **Solution: -** Standardized file path handling and used Colab's *drive.mount()* with proper validation checks.
   - **Lesson: -** A reliable configuration step is crucial to avoid cascading runtime issues during development.

2. **Token Mismanagement**
   - **Challenge: -** Hard-coded or outdated API tokens led to repeated authorization failures.
   - **Solution: -** Adopted getpass() for secure entry and added pre-checks for token validity.
   - **Lesson: -** Managing sensitive credentials securely is essential for smooth API interactions.

3. **Debugging Complexity**
   - **Challenge: -** Diagnosing issues in a multi-step RAG pipeline was time-consuming without transparency.
   - **Solution: -** Introduced verbose mode to log each stage's output including retrieved chunks and intermediate embeddings.
   - **Lesson: -** Verbose logging simplifies debugging and improves trust in the system's decisions.

6. USAGE OF GENAI TOOLS

Generative AI tools like Grok and ChatGPT played a significant role throughout the development process. They assisted in debugging complex code snippets, optimizing logic, correcting grammatical errors in documentation, and offering best practices for structuring both code and written content. These tools also provided helpful suggestions for improving code readability, resolving errors quickly, and enhancing the overall quality of technical writing. The following prompts highlight specific areas where their contributions proved especially valuable.

1. **Prompts for Debugging**

   - Prompt 01

     "My embedding results seem irrelevant to the query intent. How can I improve the semantic match between queries and document chunks?

     - Output

       Identified the suboptimal embedding model and suggested switching to a domain-specific encoder. I recommend adding query augmentation and preprocessing with sentence transformers.

     - Impact

       Boosted the relevance of retrieved documents, resulting in higher-quality, context-aware answers.

   - Prompt 02

     "The chatbot sometimes returns empty responses or times out during inference. Here's the traceback: [error text]. How can I make it more robust?"

     - Output

       Diagnosed a timeout issue due to model overload and suggested adding exception handling with retry logic. Recommended using lightweight models for non-critical queries.

     - Impact

       Improved chatbot stability and ensured fallback responses in case of inference failure, enhancing the user experience.

- Prompt 03

  "I'm getting an 'Invalid API Key' error when initializing the gemini-2.0-flash model with langchain-google-genai. Here's my code and the error: [code snippet with error text]. How can I resolve this?"

  - Output

    Identified that the error likely stems from an invalid or incorrectly configured Google API Key.

    Suggested using *getpass* to securely input the token.

    Suggested to verify the key's permissions for Gemini models.

    Provided a corrected code snippet with proper access requesting steps.

  - Impact

    Enabled successful evaluation of the model access issue, allowing successful model loading and inference.

- Prompt 04

  "The retriever sometimes fails to return relevant lecture chunks for specific technical questions. How can I make the retrieval more precise?"

  - Output

    Recommended increasing chunk overlap to preserve context and using a smaller chunk size to reduce noise. Suggested implementing hybrid retrieval (vector + keyword).

  - Impact

    Enhanced the precision of document retrieval, improving the chatbot's ability to respond to domain-specific and edge-case queries.

7. CONCLUSION

The chatbot successfully achieves the assignment's goal by delivering an interactive, Retrieval-Augmented Generation (RAG)-based question-answering system tailored for CTSE lecture notes. By integrating ***google/gemini-2.0-flash***, ***Chroma***, and ***LangChain***, the solution remains both powerful and efficient well-suited to the resource limitations of ***Google Colab***. Throughout development, key software engineering principles such as modular design, parameter optimization, and robust error handling proved essential for building a reliable NLP application. Challenges like PDF parsing and improving retrieval accuracy were addressed through iterative testing and leveraging generative AI tools. Overall, this project showcases the practical application of RAG-based systems in educational settings and lays the groundwork for future innovations in GenAI-powered academic tools.

8. APPENDICE

8.1. GitHub Repo Link

8.2. Demo Video Link