Sri Lanka Institute of Information Technology

Faculty of Computing

Department of Software Engineering

Year 4 Semester 2

SE4010 – Current Trends in Software Engineering

Assignment 2 - AI/ML

LLM Development Toolkit: Building a CTSE Lecture Notes Chatbot

Student IT Number	Student Name
IT21203176	Dias A.H.J.S.S.

2025 May

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

This report documents the design and development of a simple chatbot capable of answering questions based on CTSE (Current Trends in Software Engineering) lecture notes. The chatbot was built inside a Jupyter Notebook using LangChain and Gemini 1.5 Flash from Google Generative AI. The objective was to demonstrate the use of a Large Language Model (LLM) integrated with a document retriever to provide context-aware responses. The CTSE lecture notes in PDF format were processed and used to populate a FAISS vector database, enabling semantic search and retrieval over the content.

2 Justification of LLM Choice

Gemini 1.5 Flash from Google Generative AI was selected for the chatbot due to the following benefits:

- Low Latency and High Efficiency: Designed for speed, Gemini 1.5 Flash processes queries quickly, making it suitable for interactive applications like chatbots.
- Extended Context Handling: It supports large context windows, enabling it to process longer document chunks and maintain coherence in its responses.
- Native LangChain Support: LangChain provides dedicated support for Google Generative AI through *ChatGoogleGenerativeAI*, simplifying model integration.
- Performance/Cost Trade-off: Compared to larger models like GPT-4 or Gemini 1.5 Pro, Flash provides a good balance of performance and resource usage, especially suitable for a lightweight academic assistant.
- Multilingual and Domain Adaptability: While not fine-tuned for CTSE-specific content, its architecture allows effective generalization over retrieved domain-relevant documents, ensuring accurate responses.

In this educational chatbot context, Gemini 1.5 Flash provides the best combination of speed, context awareness, integration ease, and cost-efficiency.

3 Justification of Development Approach

The development approach follows a modular Retrieval-Augmented Generation (RAG) pipeline, which is effective for querying a specific set of documents in this scenario, CTSE lecture notes. The system design can be broken down into the following components:

- Document Ingestion: PDF files from a local *ctse_notes* directory is loaded using *PyPDFLoader*, which ensures that multi-page academic content is converted into structured LangChain *Document* objects.
- Text Splitting: The documents are split into smaller overlapping chunks (1000 characters with 200 overlap) using *CharacterTextSplitter*. This ensures semantic coherence and maximizes the chances of accurate information retrieval by the embedding model.
- Embedding Generation: Each chunk is embedded using the *GoogleGenerativeAIEmbeddings* with the "*embedding-001*" model. This converts textual data into high-dimensional vectors suitable for similarity search.
- Vector Database: The FAISS vector store is created from the embedded chunks. FAISS enables fast and accurate retrieval of the most relevant chunks based on semantic similarity to a user query.
- LLM Integration via RetrievalQA: A *RetrievalQA* chain is created using *ChatGoogleGenerativeAI* with Gemini 1.5 Flash as the underlying model. This enables the model to answer questions by consulting retrieved chunks from the vector database rather than relying purely on prior training.
- Chat Interface: A simple loop allows interactive querying. The system supports real-time Q&A based on lecture content.

Why this system design?

- It decouples document processing from model interaction, making the system modular and reusable.
- It leverages semantic retrieval, which enhances factual accuracy and ensures grounded responses.
- It is scalable, enabling easy extension to additional documents or switching out LLMs with minimal changes.

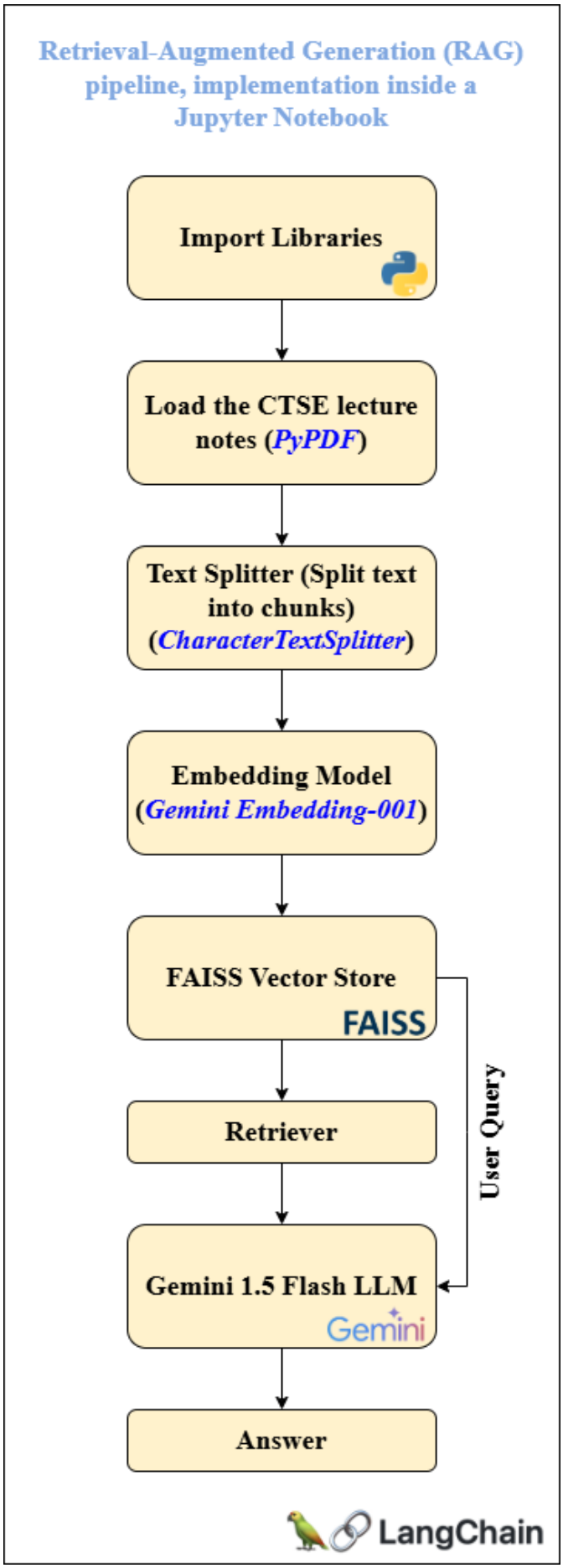

Figure 3.1 System Architecture Diagram

4 Use of GenAI Tools

The system utilizes Google Generative AI models (Gemini 1.5 Flash and Embedding-001) through LangChain's interfaces. Here's how GenAI tools and prompts are used:

4.1 Prompt 1: Chat Model Initialization

```
# Initialize Gemini chat model
chat_model = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0.3)
```

Figure 4.1 Chat model initialization

- Purpose: Initializes the chatbot model with low creativity (temperature 0.3) to focus on factual, grounded answers.
- Effect: Ensures reliable and concise responses when answering academic questions.

4.2 Prompt 2: RetrievalQA Setup

```
# Create RetrievalQA chain using FAISS vector store
qa_chain = RetrievalQA.from_chain_type(
    llm=chat_model,
    chain_type="stuff",
    retriever=vector_db.as_retriever(),
    return_source_documents=True
)
```

Figure 4.2 RetrievalQA setup

- Purpose: Combines the Gemini model with the FAISS retriever to build a RAG pipeline.
- Effect: Allows context-based responses grounded in CTSE documents.

4.3 Prompt 3: User Query in Chatbot Loop

```
while True:
    query = input("You: ")
    if query.lower() in ["exit", "quit"]:
        print("Exiting chatbot. Goodbye!")
        break

    result = qa_chain.invoke(query)
    answer = result["result"]
```

Figure 4.3 User query

User Input Prompt:

```
You:  DISTRIBUTED SYSTEM
```

Figure 4.4 User input

Lecture Note Content:

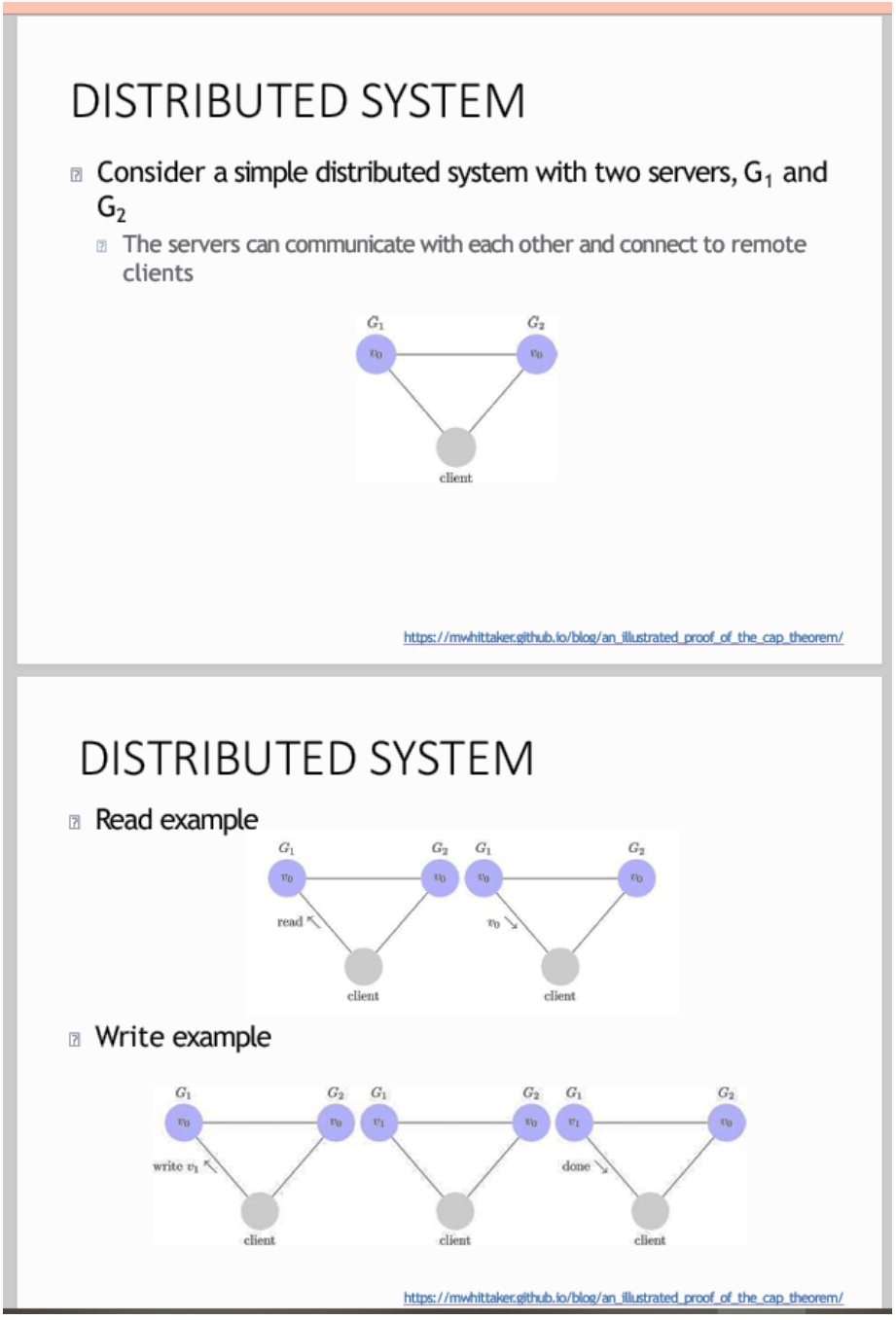

Figure 4.5 The lecture slide related to above question

Output (Sample):

```
You:  DISTRIBUTED SYSTEM
Bot: Based on the provided text, a distributed system is a system with multiple servers (like G1 and G2 in the example) that can communicate with each ot
her and connect to remote clients.  There are different types of distributed systems, categorized as CP, CA, and AP, based on trade-offs between consiste
ncy, availability, and partition tolerance.  Understanding these trade-offs, as described by the CAP theorem, is crucial for designing distributed databa
ses and systems.
```

Figure 4.6 Answer for the above question

- Effect: Demonstrates how the model retrieves context from notes and generates a well-structured answer.

5 Challenges and Lessons Learned

5.1 Module Import Errors

Challenge:

Initially encountered *ModuleNotFoundError* for modules like,

- *langchain_community.document_loaders*
- *langchain_google_genai*
- *langchain.embeddings*

Fix & Lesson:

Ensured all necessary packages were installed and up to date. Switched from deprecated imports (*langchain.document_loaders*) to the updated ones (*langchain_community.document_loaders*). This taught the importance of keeping track of evolving library versions and checking official documentation or GitHub issues for guidance.

5.2 Chunking Strategy and Content Accuracy

Challenge:

The chatbot initially returned inaccurate or overly broad answers due to improper text chunking.

Fix & Lesson:

Tweaked the *chunk_size* and *chunk_overlap* using *CharacterTextSplitter* to better preserve context within each chunk. Realized that balancing chunk size and overlap is key for improving answer relevance in retrieval-augmented generation.

5.3 Embedding Model Compatibility

Challenge:

Faced errors when using the wrong embedding model, or incorrect model names for Google Generative AI.

Fix & Lesson:

Successfully used *GoogleGenerativeAIEmbeddings* with the model "*models/embedding-001*" and confirmed setup with a valid API key. Learned that model naming must match the provider's supported list exactly.

5.4 Loading Multiple PDFs

Challenge:

The loader wasn't initially recognizing multiple PDF files in a directory.

Fix & Lesson:

Implemented a loop using *os.listdir()* to dynamically load all PDF files in the ctse_notes folder.

5.5 Using FAISS for Vector Search

Challenge:

Uncertainty around connecting document chunks with FAISS vector storage.

Fix & Lesson:

Used *FAISS.from_documents()* method correctly after embeddings were generated. Discovered how FAISS efficiently supports similarity search and integrates seamlessly with LangChain.

5.6 OpenAI API Quota Limitations

Challenge:

During the initial stages of development, the system was configured to use OpenAI's API for both embeddings and the ChatGPT model. However, a quota limitation issue was encountered, which blocked further API usage and disrupted the chatbot development process.

Lesson Learned:

To overcome this challenge, the embedding and LLM components were migrated to Google's Generative AI stack (Gemini), specifically using:

- GoogleGenerativeAIEmbeddings (embedding-001) for vector generation
- ChatGoogleGenerativeAI (gemini-1.5-flash) for the LLM

This provided a seamless and quota-free experience during testing, while also ensuring compatibility and improved integration between components from the same provider.

6 Conclusion

The chatbot built using LangChain and Gemini 1.5 Flash successfully answers user questions based on CTSE lecture notes. This project helped explore core concepts of Retrieval-Augmented Generation (RAG), document preprocessing, semantic search, and GenAI tool integration. The experience underlined the importance of chunking strategies, embedding models, and LLM prompt tuning in building useful academic assistants.

7 References

1. LangChain Documentation – https://docs.langchain.com
2. Google Generative AI – https://ai.google.dev
3. FAISS: Facebook AI Similarity Search – https://github.com/facebookresearch/faiss
4. SE4010 Lecture Note – LLM Development Toolkit (Lecture 02- Part 2 (Upgraded))
5. Gemini API Docs – https://ai.google.dev/docs
6. LLM-Chatbots – An introduction to the new world of bots (Sophie Hundertmark, Oct 12, 2024)
7. How to Build an Intelligent QA Chatbot on your data with LLM or ChatGPT (Mahesh, Jul 2, 2023)

8 Video Demonstration

https://drive.google.com/drive/folders/1PZx9Bn-mjQxnscBTddPHmKTmHOGcqRvi?usp=sharing