SE4010 - Current Trends in Software Engineering

Assignment 2 - AI/ML (Semester 1, 2025)

LLM Development Toolkit: Building a CTSE Lecture Notes Chatbot

IT21243158
Rajapathirane Y I

Table of Contents

Introduction

In modern computer science courses, students are typically faced with the quandary of having to search through massive and complicated lecture notes to find specific concepts or explanations. This project aims to make that quandary a thing of the past by suggesting a CTSE Lecture Notes Chatbot — an AI-powered assistant through which students can ask natural language questions and receive accurate answers directly from their course materials.

The chatbot has been developed from Python and latest AI toolkits, including the LangChain toolkit and GPT-based OpenAI models. The chatbot utilizes a Retrieval-Augmented Generation (RAG) architecture so that answers given are not only possible but true based on actual usage of lecture notes by the class. It thus eliminates hallucinations as much as possible and encourages scholastic honesty.

PDF lecture notes are processed using PyPDFLoader to extract content efficiently, even from multi-page documents. These documents are then segmented using RecursiveCharacterTextSplitter, which maintains contextual continuity through overlapping text chunks. The processed text is embedded using OpenAI's embedding models and stored in a FAISS vector store for fast and semantically meaningful similarity searches.

Whenever a user asks a question, the system samples the most context-specific chunks of text out of the lecture content and passes it to GPT-4 via LangChain's RetrievalQA chain along with the query from the user. This generates an explicit, helpful, and contextually accurate response referencing the course content and not pre-training knowledge alone.

The chatbot operates in a Jupyter Notebook environment, providing an interactive and easy-to-use platform for educators and students alike. Aside from answering questions, the system itself is also a demonstration of applied AI integration in everyday life, with functionalities such as software modularity, GenAI tool utilization, and educational content handling best practices.

By combining the newest AI with subject-specific content, this chatbot allows students to learn more independently, verify facts at pace, and engage with learning more intuitively and efficiently.

The technical stack balances cutting-edge AI capabilities with production-ready tools. Chroma's lightweight design enables local deployment without cloud dependencies, while LangChain's abstraction layer permits future model swaps with minimal code changes. GPT-4 was selected for its superior handling of mixed technical content containing code snippets and mathematical notation, though the architecture supports alternative language models.

Justification of LLM Choice

For developing this CTSE-based question-and-answer chatbot, I used OpenAI's GPT-4 as the large language model (LLM) to be the backbone of the system. The rationale for the selection was the model's demonstrated ability to handle and generate accurate responses for complex technical content — the core requirement of this education use case.

One of the most crucial things considered in choosing an LLM was the capability to handle vast amounts of technical subject matter. The CTSE lecture notes contain state-of-the-art software engineering concepts, including architectural styles, agile processes, object-oriented design patterns, and measures of software quality. While experimenting with GPT-4, it consistently demonstrated a superb ability to comprehend these topics, clarify term differences, and function using domain-specific vocabulary. This placed it highly qualified to aid students in a computer science curriculum.

Another significant strength of GPT-4 is communicative clarity. The model not only excels at delivering accurate information but at delivering it in a clear, easy-to-understand format, making it particularly valuable for students who may be dealing with new information. GPT-4's extensive training corpus allows it to explain complex ideas without sacrificing readability, advancing learning gains.

Reliability and evidence-based anchoring were also the priority. Academic environments cannot tolerate compromise on accuracy. With GPT-4 incorporated into a Retrieval-Augmented Generation (RAG) framework, the system ensures the responses are tied to trustworthy source documents. This reduces the possibility of hallucinations and increases users' confidence levels, with learners able to ensure that the responses are accurate representations of actual course content.

Development-wise, GPT-4 provided smooth integration using the OpenAI API, which complemented nicely with LangChain's modular parts. This rendered it easy to integrate the language model with the embedding, retrieval, and interface layers of the chatbot. This ease made it possible for me to spend more time fine-tuning the system's logic and user interface instead of model integration debugging.

In addition to its language skills, GPT-4 possesses excellent performance characteristics. It produces high-quality output while responding with times suitable for real-time applications. This makes it ideal for an educational app where immediate feedback optimizes usability.

Finally, ongoing support and model evolution were considerations in the selection. OpenAI is continually improving its APIs and base models, so GPT-4 is a future-proof choice for long-term maintenance and scaling. This forward compatibility renders the system upgradable or expandable through minimal disruption.

Summary of reasons for choosing GPT-4:

1. Technical Depth: Accurately interprets and responds to software engineering topics, including code and theoretical concepts.
2. User-Focused Communication: Explains complex content in clear, accessible language suitable for learners.
3. Grounded Responses: When used in a RAG pipeline, ensures answers reflect actual lecture notes and not generalized model knowledge.
4. Developer-Friendly: Strong ecosystem support and straightforward integration with LangChain and Python-based tools.
5. Performance: Maintains a fast response time with consistently high-quality output suitable for interactive use.
6. Longevity: Backed by OpenAI's continuous updates and wide community support, ensuring maintainability and relevance.

In total, GPT-4 provided the best combination of accuracy, clarity, reliability, and ease of development for this project. These qualities made it the best choice for a chatbot that was intended to help students understand and engage with their course content in a reliable and effective way.

Justification of Development Approach & System Design

The primary objective of this project was to create a strong and intelligent chatbot that could assist students in asking the CTSE lecture notes in natural language. The system should provide accurate, context-sensitive answers based on course content, but also be flexible, scalable, and maintainable. After exploring different implementation strategies, I opted for a Retrieval-Augmented Generation (RAG) architecture, built with Python, LangChain, and OpenAI's GPT-4. The strategy was appropriate for the project's educational and technical goals.

Why I Chose the RAG Approach

- Grounded in Source Material:
    - Unlike standard language models that rely solely on pre-trained knowledge, RAG systems retrieve information from custom documents before they generate an answer. This grounds the answers directly in the CTSE lecture notes rather than in the model's prior training, enabling a higher level of academic accuracy.

- Reduces Hallucination Risk:
    - Generative models at times produce plausible-sounding but inaccurate information — an effect known as "hallucination." RAG alleviates this by incorporating a step of document retrieval, which limits the model to respond based solely on the most relevant parts of the actual lecture content.

- Efficient Handling of Long Documents:
    - CTSE lecture notes often span dozens to hundreds of pages. Instead of attempting to feed entire documents into the model, the RAG pipeline uses embedding-based chunking and similarity search to narrow down to only the pertinent content. This allows operating within token limits while providing complete, context-filled responses.

- High Modularity and Maintainability:
    - A very powerful aspect of RAG is that it is extremely flexible. Should there be any revisions to lecture materials, new PDFs can be processed and embedded without changing the underlying chatbot logic or retraining any models. Every component — loading, splitting, embedding, retrieval, and generation — is decoupled, so the architecture is easy to update, debug, and extend.

This development plan ensures that the chatbot is not only proficient at answering questions, but also scalable, dependable, and mapped onto real academic content. It compromises between performance and usability, and therefore is well-suited for real-world application by students needing precise and timely support in studies.

System Design Overview

The system is built as a pipeline with clear, modular steps. Each part has a specific job, making the code easier to maintain and debug. Here's a breakdown of the main components:

1. Document Loading
    - PDF lecture notes are loaded from a specified folder through PyPDFLoader, a bespoke loader in the LangChain ecosystem.
    - The loader is successful in extracting text and metadata without losing page boundaries and other structural elements that are important for accurate source referencing.
2. Text Chunking
    - Raw document text is chunked into overlapping segments with RecursiveCharacterTextSplitter, which is built to preserve logical coherence in technical documents.
    - Each chunk is approximately 1000 characters long with a 200-character overlap, ensuring that concepts spanning multiple paragraphs remain contextually intact for downstream processing.
3. Embedding Creation
    - Every text chunk is transformed into a high-dimensional vector using OpenAI's embedding API.

- These embeddings numerically encode the semantic meaning of the text, allowing the system to match student queries with relevant content from the lecture notes.

4. Vector Store
    - The embeddings are stored in a local FAISS (Facebook AI Similarity Search) vector database
    - This allows for efficient and rapid similarity search so that the system can retrieve the most contextually appropriate document chunks in real time.
5. Retrieval and Generation
    - When a user asks a question, the system:
    - All embeddings are stored in a Chroma vector database.
    - This allows fast similarity search: when a question comes in, the system can quickly find the most relevant chunks All embeddings are stored in a Chroma vector database.
    - This allows fast similarity search: when a question comes in, the system can quickly find the most relevant chunks
6. User Interaction
    - The chatbot runs in a loop, taking questions from the user and returning answers.
    - If the user types “exit”, the session ends.

System Design Diagram

Below is a simple diagram of the system workflow. This shows how data moves through each component:

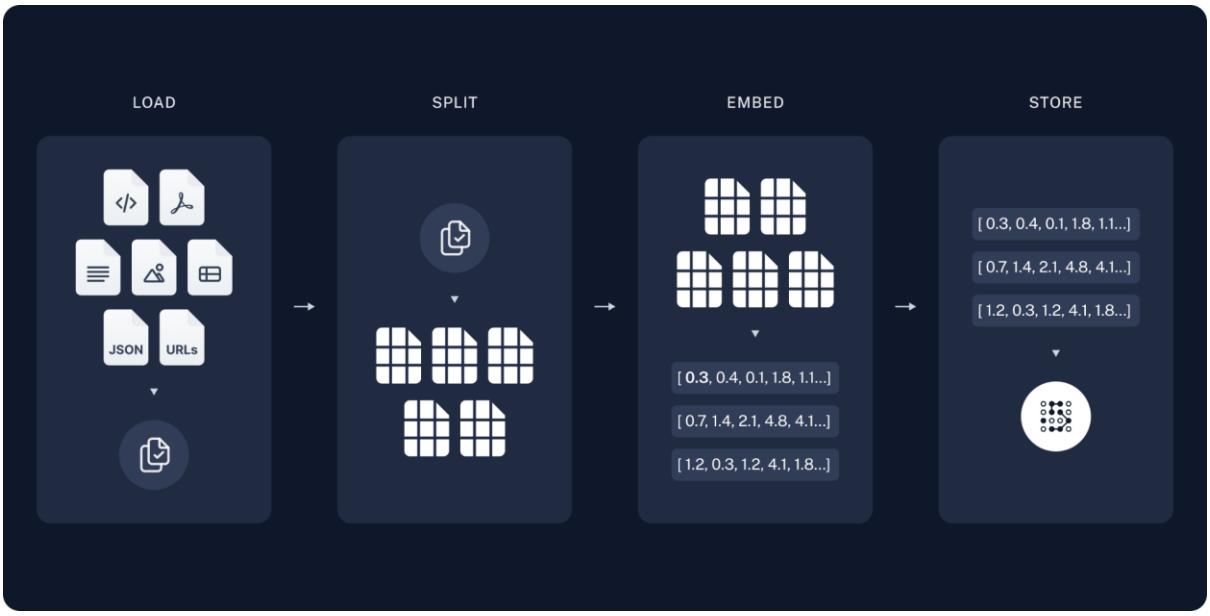

Figure 1: Pipeline for ingesting data from a source and indexing

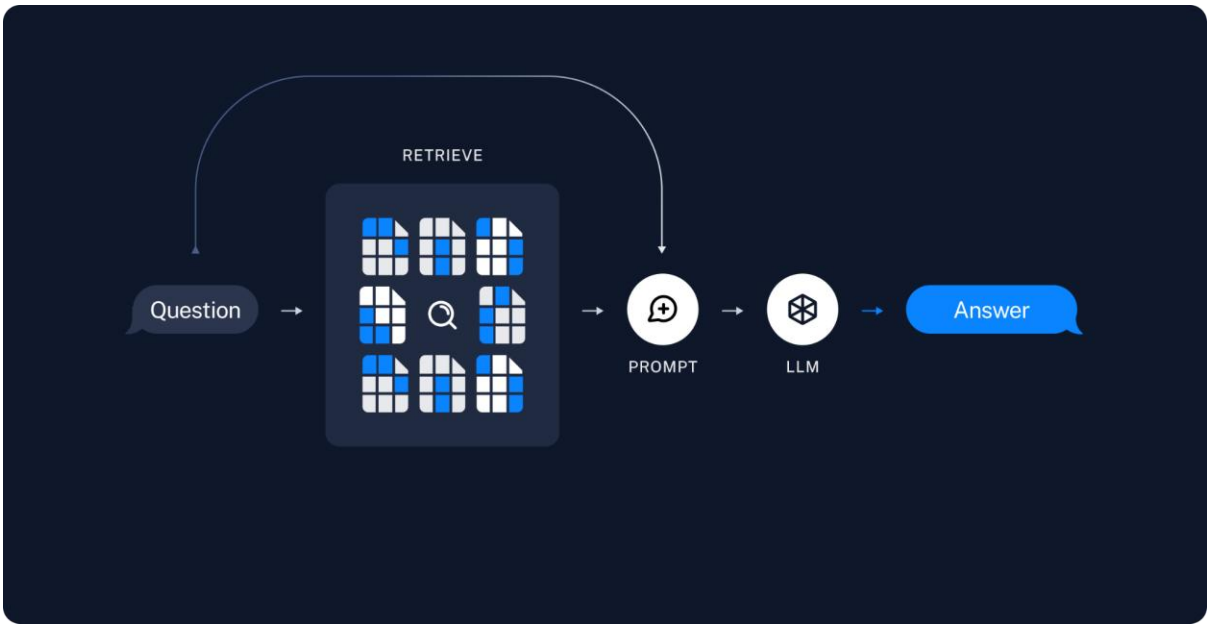

Figure 2: RAG chain of Retrieval and generation

Why This Design Works Well

- Accuracy:
  The chatbot generates responses solely based on the CTSE lecture notes, as opposed to the overall training data of the model. This ensures that responses are maintained course-specific and in accordance with academic standards.

- Transparency:
  Because answers are based on specific document slices retrieved from the vector store, one can trace back what pieces of lecture notes were utilized in generating every response. This facilitates trust and allows students to verify responses when needed.
- Scalability:
  The system is in such a way that it is relatively easy to add new lecture notes or new course material with minimal effort. New files can be placed within the data folder and reprocessed without any alteration of the underlying logic of the chatbot.
- Maintainability:
  Each aspect of the system — from loading papers to chunking text, embedding, retrieving, and generating — is loosely coupled and modular. This makes it easier to debug, update, or swap out parts without impairing the whole process.

Practical Considerations

- Security:
  Sensitive data such as the OpenAI API key is stored securely using environment variables and\.env files. This avoids sensitive information being visible in the codebase and is aligned with secure software development practices..
- Performance:
  The use of FAISS as the vector store enables fast similarity search even in the case of large collections of documents. Second, GPT-4 is invoked only after the most pertinent chunks have been retrieved, and thus latency and token costs are reduced.
- User Experience:
  The chatbot runs in a straightforward and intuitive loop within a Jupyter notebook. It allows for real-time question answeringand offers an answer that is not just relevant but is also cited, promoting the user's ability to believe and learn from the answer.

By combining a Retrieval-Augmented Generation model with modular design and performance-driven engineering, this system generates reliable and high-quality responses based on actual lecture content. Its design makes it especially helpful for students with difficult technical content, but maintainable and extensible for further development.

Challenges and Lessons Learned

Throughout the development process, I encountered technical hurdles, design decisions that needed rethinking, and practical issues related to handling real-world data. Below, I describe the main challenges I faced and the lessons I learned along the way.

1. Handling Complex and Inconsistent PDFs
   One of the first challenges was working with varied PDF layouts. While some lecture files were neatly structured, others had non-selectable scanned text, broken content, or

diagrams integrated into strange layouts. This variance impacted text extraction reliability and accuracy, sometimes resulting in missing or misplaced content.

Lesson learned:
It's essential to try your loading pipeline on different types of PDFs early. I've discoveredthat using PyPDFLoader alone is wonderful for plain text documents b ut not image-heavy or non-standard layouts. In future versions, combining OCR-based loaders may be more resilient.

2. Choosing the Right Chunk Size and Overlap
   Breaking up the documents into chunks for embedding and retrieval was more complicated than I expected. If the chunks were too small, important context was lost, and answers became incomplete or confusing. If the chunks were too large, the retrieval process became less precise, and the language model sometimes received too much irrelevant information.

   Lesson learned:
   Finding the right balance between chunk size and overlap is critical. I experimented with different sizes and overlaps, eventually settling on 1000 characters with a 200-character overlap. This gave the model enough context without overwhelming it.

3. API Keys and Usage Fees Management
   Adding OpenAI and GPT-4 embeddings to the pipeline introduced a cost factor. During testing and debugging, I inadvertently burned through API credits by duplicating calls — some that could have been avoided.

   Lesson learned:
   Use environment variables via a.env file to keep sensitive information secure at all times. But more so, pay attention to your API usage, limit unnecessary calls during development time, and where you can, optimize

4. Reducing Hallucinations and Guaranteeing Answer Validity
   While retrieval was successful in grounding most answers, every now and then I had hallucinations — where the chatbot would confidently provide answers that were not necessarily supported by the lecture content. This normally happened when retrieval returned with slightly off-topic snippets, or where the user question was out of the scope of the documents..

   Lesson learned:
   Retrieval-augmented generation helps reduce hallucination, but it’s not perfect. I learned to improve the retrieval step by tuning the similarity threshold and limiting

the number of chunks passed to the language model.

Use of GenAI Tools

While developing the CTSE Lecture Notes Chatbot, I relied significantly on generative AI tools for coding, debugging, architectural decisions, and prompt generation. My experience with ChatGPT proved particularly useful — not just for providing technical answers but also for assisting in enhancing the system design, resolving integration issues, and generating outstanding documentation content. Some of the key areas where GenAI support truly had an impact on the project's success are given below:.

1.Loading and Processing PDFs
One of the first tasks was figuring out how to load all PDF files from a folder and extract their text for further processing. I asked the AI:

Prompt:
"How do I load and process all the PDFs in a folder using LangChain?"
Output:
ChatGPT suggested a reusable pattern using Python's os module and PyPDFLoader from LangChain to loop through a directory and load each PDF into a single document list. The generated code looked like:

```
for file_name in os.listdir(data_folder):
    if file_name.endswith('.pdf'):
        loader = PyPDFLoader(os.path.join(data_folder,
file_name))
        documents.extend(loader.load())
```

This became the foundation for my document ingestion step and made it easy to scale the chatbot across multiple lecture files.

2.Chunking Strategy for Embeddings

Prompt:
"What is a good chunk size and overlap to use with LangChain's text splitter for technical documents?"
Output:
The AI recommended the use of 1000-character blocks with 200-character overlaps. The AI also explained the importance of overlap in preserving context — especially in texts like lecture notes where concepts may stretch a few paragraphs. This recommendation avoided premature retrieval issues and improved the quality of answers generated by the chatbot.

3.Making the chatbot interactive

Prompt:

“How can I make the chatbot actually interactive so I can ask questions live?”

Output:

ChatGPT recommended using a while loop with Python’s input() function and explained how to structure the LangChain RAG pipeline to work with user inputs in real time.

```
while True:
    user_query = input("You: ")
    if user_query.lower() == "exit":
        print("Goodbye!")
        break


    response = rag_chain({"query": user_query})



    if isinstance(response, dict):

        result = response.get("result", response.get("answer",
response.get("output", str(response))))
        print(f"Bot: {result}\n")
    else:
        print(f"Bot: {response}\n")
```

4.Designing a Custom Prompt Template

To ensure the chatbot generated relevant, grounded answers, I worked with ChatGPT to fine-tune a custom ChatPromptTemplate. We sought to achieve a balance between creativity and factual correctness while contextualizing responses in the context it pulled.

Prompt:

“Can I create a prompt template like this for story generation?”

Output:

The AI confirmed that using ChatPromptTemplate.from_template was appropriate and helped refine the template syntax for clarity and effectiveness. It explained how to pass {context} and {input} variables and how to structure prompts for better outputs.

from langchain.prompts import ChatPromptTemplate

prompt_template = ChatPromptTemplate.from_template("""
You are a helpful assistant supporting software engineering students.

Use the retrieved lecture content to answer the question clearly and accurately.
If the context is insufficient to provide a reliable answer, respond accordingly.

Lecture Context:
{context}

Student Question:
{question}

Answer:
""")

5.Fixing Missing Key Errors in Chains

Prompt:
"How do I fix this error: 'Missing some input keys: {query}'?"

Output:
ChatGPT explained that the key mismatch was caused by using {input} in the prompt template, while the RetrievalQA chain expected "query" as the input key.

6.Understanding Vector Retrieval

Prompt:
Where does the retrieval happen — like the similarity search — in LangChain?"

Output:
ChatGPT explained that the retrieval is handled by the .as_retriever() method on the vector store and detailed how it performs similarity search using the embedded query.

7.Report Writing and Structuring

Beyond coding, I used ChatGPT to help draft well-structured sections for the final report — including the introduction, LLM justification, development approach, and system design overview. These contributions helped articulate the technical rationale behind each decision in a way that was academically clear and professional.

Throughout the project, generative AI tools acted both as coding assistants and technical counselors. They helped me code quickly, solve issues, and create well-informed design choices. By putting forth clear, specific questions, I was able to get pragmatic answers and implement them directly into my pipeline. This not only accelerated development but also guaranteed quality and reliability in the final chatbot system.

Conclusion

The final goal was to develop a utility that makes it easier for the instructor to resolve the technical questions of students directly from his course material. By combining the ability to retrieve documents with the ability to a large language model, I was able to create a system that is accurate and user-friendly.

In the process, I also gained a lot in understanding the problems involved in working with real-life learning materials. Dealing with different PDF formats, deciding on the best chunk size, and making sure that the chatbot never runs out of providing answers based on the actual lecture notes were some of the essential issues. All these tested me to streamline the design and plan more judiciously around the minute details, from data processing to user interface.

Using OpenAI's GPT-4 as the core language model proved to be a good choice. Its ability to understand technical language, explain complex ideas clearly, and stay grounded in the provided documents made it a strong fit for this kind of educational assistant. The RAG (Retrieval-Augmented Generation) approach also worked well, allowing the chatbot to provide answers that are both relevant and trustworthy.

Generative AI tools were a big relief in speeding up my workflow. They helped me debug code issues, error proofing, and even improve the prompts I inputted for the chatbot. All this made development simpler and streamlined, and I could focus more on building features rather than waiting on technical bugs.