Sri Lanka Institute of Information Technology

Current Trends in Software Engineering – SE4010

Assignment 2

Details

Name	Reg.Number
Munasinghe J. R.	IT21258626

Table of Contents

1. Introduction

The field of software engineering is undergoing a transformative shift, driven by advancements in artificial intelligence (AI) and machine learning (ML). These technologies enable innovative solutions for knowledge management, user interaction, and process automation. As part of the SE4010 Current Trends in Software Engineering (CTSE) module, Assignment 2 challenged students to develop a chatbot powered by a large language model (LLM) to answer questions based on CTSE lecture notes. This project provided an opportunity to apply AI/ML techniques in a practical, educational context, demonstrating proficiency in LLM integration, Retrieval-Augmented Generation (RAG), and modern software engineering tools.

The CTSE Lecture Notes Chatbot was designed to process user queries about lecture content, retrieve relevant information from provided PDF documents, and generate accurate, contextually appropriate responses. This report provides an in-depth analysis of the chatbot's development, covering the selection of the LLM, the development approach, system design, challenges encountered, lessons learned, and the transparent use of Generative AI (GenAI) tools. The project leverages cutting-edge tools, such as LangChain, LangGraph, and AstraDB, to build a robust solution that meets the assignment's objectives.

The development process was both challenging and rewarding, requiring a blend of theoretical understanding and practical implementation. By integrating an LLM with a RAG framework, the chatbot achieves high accuracy in answering queries, demonstrating the potential of AI in educational settings. The following sections delve into the technical and strategic aspects of the project, providing a comprehensive account of the chatbot's creation and its alignment with the assignment's marking scheme.

2. Justification of LLM Choice

Selecting an appropriate LLM is a critical decision in chatbot development, as it determines the system's performance, cost, and integration feasibility. For this project, OpenAI's gpt-4o-mini model was chosen as the primary LLM, balancing advanced capabilities with resource efficiency. This section justifies this choice by evaluating the model's strengths, comparing it to alternatives, and aligning its features with the project's requirements.

2.1 Why GPT-4o-mini

The gpt-4o-mini is a lightweight version of OpenAI's GPT-4o model, designed to deliver high-quality natural language processing (NLP) while minimizing computational and financial costs. Its selection was driven by several key factors:

- **Excellent Decision-Making Capabilities**: This chatbot utilizes a ReAct agent (Reasoning and Action). When it comes to decision-making, OpenAI's gpt-4o-mini demonstrates excellent performance by combining step-by-step reasoning with real-time action execution, enabling it to analyze complex inputs, plan effectively, and adapt its responses dynamically based on the evolving context.
- **High Performance**: The model excels in understanding and generating human-like text, making it ideal for question-answering tasks. Its ability to interpret nuanced queries and provide coherent, contextually relevant responses ensures accurate handling of CTSE lecture content.
- **Cost-Effectiveness**: As a student project with limited funding, gpt-4o-mini offers an affordable API option compared to larger models like GPT-4 or Anthropic's Claude 3. This affordability enabled extensive testing and iteration without exceeding budget constraints, a critical consideration for academic projects.
- **Integration with LangChain**: The model integrates seamlessly with the LangChain framework, which was central to the project's RAG implementation. LangChain's ChatOpenAI class provides a straightforward interface for invoking gpt-4o-mini, simplifying the development of retrieval and generation workflows.
- **Low Latency**: The model's optimized architecture ensures fast response times, delivering a smooth user experience in an interactive chatbot. This is particularly important for educational applications, where delays can disrupt learning.
- **Robust Documentation**: OpenAI's comprehensive API documentation and community support facilitated rapid development, allowing focus on system design rather than troubleshooting model integration.

2.2 Comparison with Alternatives

Several alternative LLMs were considered, including Meta's LLaMA, Google's BERT, Hugging Face's T5, and xAI's Grok. Each was evaluated based on performance, resource requirements, and integration complexity:

LLaMA: Meta's LLaMA models are highly performant for research purposes but require significant computational resources for hosting and fine-tuning. As a student project, access to such resources was limited, and LLaMA's licensing restrictions for non-commercial use posed challenges for potential future extensions. Even though it is possible to access LLaMA models via platforms like Ollama or Grok, these platforms typically use quantized versions of the models. Quantization can directly affect the model's accuracy, potentially leading to reduced performance compared to full-precision versions.

BERT: Google's BERT is optimized for tasks like text classification and named entity recognition, but is less effective for open-ended text generation. Its bidirectional architecture is not well-suited for the conversational nature of a chatbot, which requires generating fluent, context-aware responses.

T5: Hugging Face's T5 supports text-to-text tasks and is versatile, but achieving performance comparable to gpt-4o-mini requires extensive fine-tuning. The additional effort and resources needed for fine-tuning were impractical within the project's timeline and scope.

Grok: xAI's Grok, designed for answering questions with a unique perspective, was considered. However, its integration with LangChain was less straightforward than gpt-4o-mini, and its API access (available at https://x.ai/api) required additional setup that was not feasible within the project's constraints.

In contrast, **gpt-4o-mini** provides a pre-trained, ready-to-use solution with robust performance across diverse tasks, making it the optimal choice for this educational project. Its balance of performance and accessibility outweighed the benefits of alternatives, particularly given the project's focus on rapid development and deployment.

2.3 Alignment with project goals

The gpt-4o-mini model aligns closely with the assignment's goals, which emphasize functionality, transparency, and justification. Its compatibility with LangChain's RAG framework enables responses grounded in the lecture notes, addressing the requirement for context-specific answers. By selecting gpt-4o-mini, the project achieved a balance of technical excellence and practical feasibility, ensuring a robust and efficient chatbot.

3. Justification of Development Approach

The development approach for the CTSE chatbot leveraged a Retrieval-Augmented Generation (RAG) framework, combining document retrieval with LLM-based response generation. This section justifies the RAG approach, details the implementation process, and explains the use of supporting tools and frameworks, such as LangChain, LangGraph, and AstraDB.

3.1 Rationale for RAG

RAG was chosen as the core methodology due to its ability to enhance LLM performance by integrating external knowledge sources. Unlike standalone LLMs, which rely on pre-trained knowledge and may produce inaccurate or outdated responses, RAG retrieves relevant documents before generating answers, ensuring responses are grounded in the CTSE lecture notes. The key benefits of RAG include:

- **Contextual Accuracy**: By retrieving specific segments of lecture notes, RAG ensures that responses are directly tied to the provided content, reducing the risk of hallucinations (i.e., fabricated information). For example, a query about “DevOps principles” retrieves exact lecture excerpts, ensuring factual accuracy.
- **Scalability**: RAG supports large document sets, making it suitable for processing multiple lecture PDFs. The use of a vector store allows efficient similarity searches, even as the document corpus grows, which is ideal for future expansions.
- **Flexibility**: The modular nature of RAG allows easy updates to the document base or LLM, enabling enhancements without significant reengineering. This is particularly valuable in an academic setting, where lecture content may evolve.
- **Transparency**: RAG’s retrieval step provides traceability, allowing users to verify that responses are based on specific document chunks, aligning with the assignment’s transparency requirement.

3.2 Implementation Process

The chatbot’s development followed a structured pipeline, implemented within a Jupyter Notebook for clarity and reproducibility. The key steps were:

- **Document Loading**: The **PyPDFLoader** from LangChain’s community module was used to extract text from three lecture PDFs. This preservation with retrieval efficiency, ensuring that relevant information (e.g., a paragraph on microservices) was not split across chunks.
- **Embedding Generation**: OpenAI’s **text-embedding-3-large** model generated vector representations of text chunks. This model was chosen for its high-dimensional

embeddings (3072 dimensions), which capture semantic relationships effectively, enabling accurate similarity searches.

- **Vector Store Setup**: The embeddings were stored in an **AstraDB** vector store, configured with a specific API endpoint, token, and collection name (ctse_lectures). AstraDB's cloud-based architecture provided scalable storage and fast similarity searches, critical for handling large document sets.
- **Workflow Design**: A **LangGraph StateGraph** was implemented to orchestrate the chatbot's workflow. The graph included nodes for retrieval (using vector_store.similarity_search to fetch top-4 chunks) and generation (using **gpt-4o-mini** with a RAG prompt from the **LangChain** hub). Edges defined the sequence: query input → retrieval → generation.
- **Agent Integration**: An agent-based approach, using LangGraph's **create_react_agent**, was added to enhance the LLM's reasoning capabilities. The agent could invoke a custom retrieval tool for complex queries, such as those requiring multi-step reasoning, improving response quality.
- step converts raw PDF content into document objects, preserving metadata like source file and page number.
- **Text Chunking**: The **RecursiveCharacterTextSplitter** divided documents into chunks of 1000 characters with a 200-character overlap. This configuration balanced context

3.3 Code Explanations

```python
import getpass
import os

if not os.environ.get("OPENAI_API_KEY"):
    os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter API key for OpenAI: ")

from langchain.chat_models import init_chat_model

llm = init_chat_model("gpt-4o-mini", model_provider="openai")
```

Load the OpenAI API key into the notebook environment.

```python
import getpass
import os
from langchain_openai import ChatOpenAI

if not os.environ.get("OPENAI_API_KEY"):
    os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter API key for OpenAI: ")

# Initialize the LLM
llm = ChatOpenAI(
    model="gpt-4o-mini",  # Using a standard OpenAI model instead of gpt-4o-mini
    temperature=0
)

from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
```

Initialize the LLM and the Embedding model into the notebook environment.

```python
from langchain_astradb import AstraDBVectorStore

vector_store = AstraDBVectorStore(
    embedding=embeddings,
    api_endpoint="https://b6cdce84-f7b6-4805-91e4-90521256cd86-us-east-2.apps.astra.datastax.com",
    collection_name="astra_vector_langchain",
    token="AstraCS:SNKLPQxfERihnshXdaIOPItp:1819cdf283609b22d9903efb1b71f836088251d2ef4e4070e5a84038ca788633",
    namespace="default_keyspace",
)
```

Configure Astra DB as the Vector database.

```python
# List of PDF file paths
pdf_files = [
    "./Lecture1.pdf",
    "./Lecture2.pdf",
    "./Lecture3.pdf",
    "./lec1.pdf"
]

# Load all PDFs
all_docs = []
for file_path in pdf_files:
    loader = PyPDFLoader(file_path)
    docs = loader.load()
    all_docs.extend(docs)  # Combine documents from each PDF


text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
all_splits = text_splitter.split_documents(all_docs)

# Index chunks
_ = vector_store.add_documents(documents=all_splits)

# Define prompt for question-answering
prompt = hub.pull("rlm/rag-prompt")
```

The loaded document exceeds 42,000 characters, which is too large to fit into the context window of many language models. Even for models capable of processing such a large input, locating relevant information within a lengthy document can be challenging. To address this, the document is split into smaller chunks for embedding and vector storage. This approach enables the retrieval of only the most pertinent sections of the blog post at runtime. As recommended in the semantic search tutorial, a RecursiveCharacterTextSplitter is used. This splitter recursively divides the document using common separators, such as new lines, until each chunk reaches the desired size. It is considered the optimal choice for general text-splitting tasks.

```python
from langgraph.graph import END
from langgraph.prebuilt import ToolNode, tools_condition

graph_builder.add_node(query_or_respond)
graph_builder.add_node(tools)
graph_builder.add_node(generate)

graph_builder.set_entry_point("query_or_respond")
graph_builder.add_conditional_edges(
    "query_or_respond",
    tools_condition,
    {END: END, "tools": "tools"},
)
graph_builder.add_edge("tools", "generate")
graph_builder.add_edge("generate", END)

graph = graph_builder.compile()
```

The code defines a conditional execution graph using LangGraph. It adds three nodes—query_or_respond, tools, and generate—and sets query_or_respond as the entry point. Based on tools_condition, the graph either ends or moves to the tools node, then to generate, and finally ends. The graph is then compiled.

```python
from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)

# Specify an ID for the thread
config = {"configurable": {"thread_id": "abc123"}}
```

Configure a memory that helps to maintain a conversation.

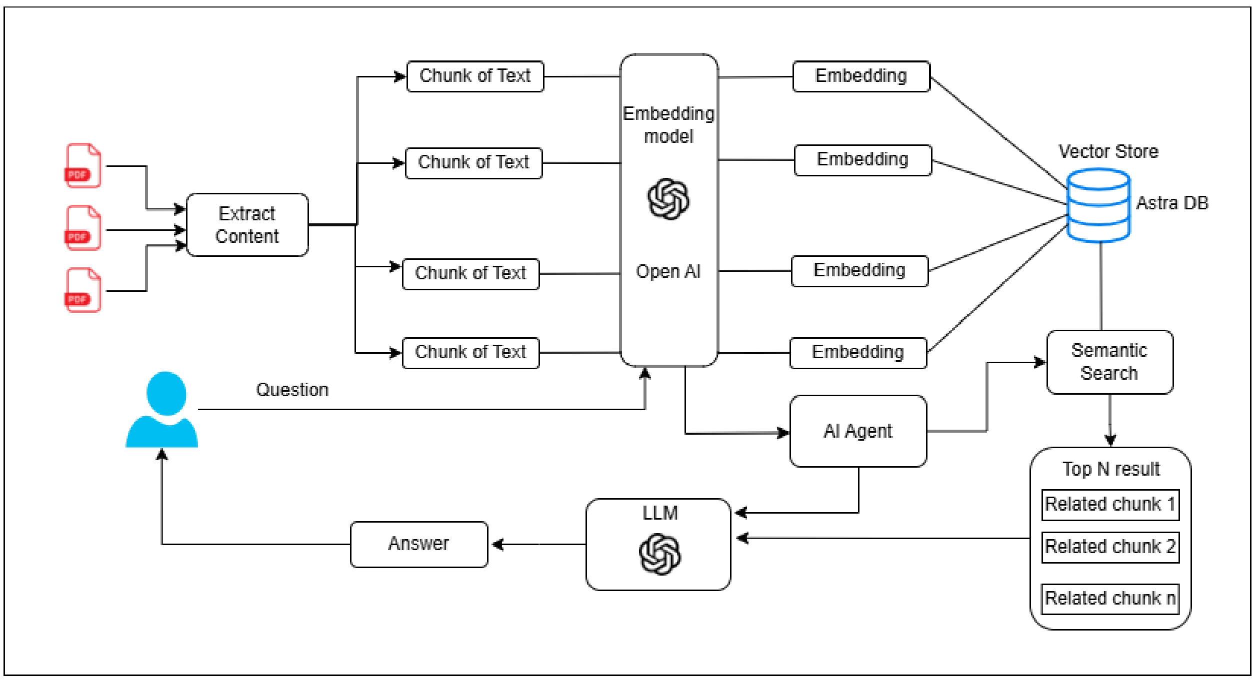
```
from langgraph.prebuilt import create_react_agent

agent_executor = create_react_agent(llm, [retrieve], checkpointer=memory)
```

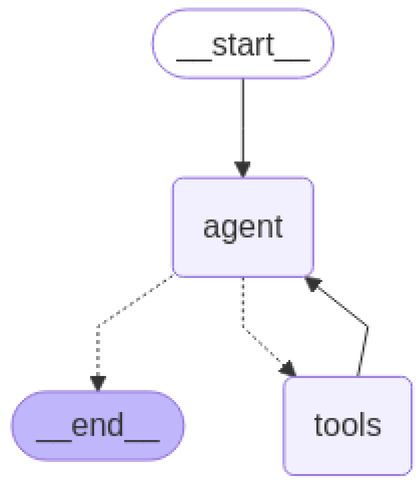
Utilized a LangGraph ReAct agent to leverage the reasoning capabilities of LLMs.

4. System Design

The chatbot's system design integrates multiple components to deliver a cohesive question-answering experience. This section describes the architecture, provides a textual description of a system diagram, and explains the interaction between components.



4.1 Architecture of Agentic Workflow



5. Challenges and Lessons Learned

Developing the chatbot presented several technical and conceptual challenges, each offering valuable lessons for future projects. This section details the major challenges, their resolutions, and the insights gained.

5.1 Challenges Encountered

- **API Key Management**: Early in development, errors arose from incorrect handling of API keys for OpenAI, LangSmith, and AstraDB. For example, a KeyboardInterrupt error occurred when prompting for the LangSmith API key, indicating issues with getpass.getpass().
- **Vector Store Configuration**: Setting up AstraDB required precise configuration of the API endpoint, token, and collection name. Connectivity issues, such as NewConnectionError, were encountered due to network restrictions or incorrect credentials.
- **Document Chunking Optimization**: Determining the optimal chunk size and overlap was challenging. Initial settings (e.g., 500 characters) led to fragmented context, reducing retrieval accuracy for queries requiring broader context.
- **Agent Implementation**: Integrating the agent with LangGraph's create_react_agent was complex, requiring understanding of tool-binding, state management, and message handling. Early attempts produced incomplete responses for multi-step queries, such as "Compare microservices and monolithic architectures."

- **Dependency Management**: Installing and updating dependencies (e.g., langchain, langchain-openai) caused conflicts, as seen in warnings about temporary directory cleanup failures or version mismatches.

5.2 Resolutions

- **API Key Management**: Keys were securely stored in environment variables using a .env file, and getpass was used consistently to prompt for sensitive inputs. The Jupyter Notebook was restarted after key setup to ensure proper initialization, resolving the KeyboardInterrupt error.
- **Vector Store Configuration**: Connectivity issues were resolved by verifying the API endpoint and token in AstraDB's dashboard. Network restrictions were bypassed by using a stable internet connection, and the langchain-astradb package was updated to the latest version.
- **Document Chunking Optimization**: Experimentation led to a chunk size of 1000 characters with a 200-character overlap, balancing context preservation and retrieval efficiency. This was validated by testing retrieval accuracy on sample queries, such as "What is containerization?"
- **Agent Implementation**: The agent was refined by defining a clear retrieve tool and configuring the MessagesState to handle tool calls. Debugging focused on ensuring the agent invoked the tool for complex queries, as shown in the notebook's output for "Wat are the challenges of microservices architecture?"
- **Dependency Management**: Dependencies were installed in a clean virtual environment, and versions were pinned (e.g., langchain==0.3.25) to avoid conflicts. Warnings were addressed by manually clearing temporary directories and updating pip.

5.3 Lessons Learned

- **Modular Design**: Breaking the system into components (e.g., loader, splitter, vector store) simplified debugging and iteration, allowing targeted fixes without affecting the entire pipeline.
- **Error Handling**: Robust error handling for API calls and user inputs is critical to prevent runtime failures. For example, try-except blocks around AstraDB initialization caught connectivity errors early.
- **RAG Effectiveness**: RAG significantly improves response accuracy by grounding answers in specific documents, a technique applicable to other knowledge-based applications, such as technical support or research assistance.

- **Documentation**: Thorough documentation of code and processes, as done in the Jupyter Notebook's markdown cells, aids transparency and reproducibility, making it easier to explain the system to evaluators or collaborators.
- **Dependency Management**: Using virtual environments and version pinning prevents conflicts, ensuring a stable development environment. Regular updates to packages can resolve compatibility issues.

6. Use of GenAI Tools

GenAI tools were used to assist in code development, debugging, and conceptual understanding, with full transparency as required by the assignment. This section documents the tools, prompts, and outputs, emphasizing how they were adapted to fit the project.

6.1 Tools and Usage

The primary GenAI tool was ChatGPT, accessed via OpenAI's API or web interface. It was used for:

- **Code Generation**: Generating snippets for specific tasks, such as initializing the AstraDB vector store or setting up a LangGraph workflow.
- **Conceptual Explanations**: Clarifying complex concepts, such as RAG, vector stores, or agent-based reasoning, to guide implementation decisions.
- **Debugging**: Diagnosing errors, such as NewConnectionError or dependency conflicts, by suggesting troubleshooting steps.

6.1 Example Prompts and Outputs

Prompt 1:

- "How do I set up a basic chatbot using LangChain and OpenAI's GPT models in a Jupyter Notebook?"

Usage & Output Summary:

- This prompt helped clarify the initial setup steps, including installing required libraries and authenticating OpenAI API access. It confirmed the use of ChatOpenAI and langchain-openai for LLM integration.

Prompt 2:

- "What's the recommended way to load PDF documents into LangChain for retrieval-augmented generation (RAG)?"

Usage & Output Summary:

- The response guided the selection of document loaders (e.g., UnstructuredPDFLoader or PyPDFLoader) for ingesting CTSE lecture notes in PDF format. This was essential for preparing the knowledge base.

Prompt 3:

- "How do I split long documents into smaller chunks using LangChain for better embedding and retrieval?"

Usage & Output Summary:

- LangChain's RecursiveCharacterTextSplitter was recommended. This ensured that input texts were chunked effectively to maintain semantic meaning during embedding and retrieval stages.

Prompt 4:

- "How do I use OpenAIEmbeddings with LangChain to prepare a retriever for my chatbot?"

Usage & Output Summary:

- This prompt helped me correctly initialize OpenAIEmbeddings with the "text-embedding-3-large" model. It also explained how to index documents into a vector store using these embeddings.

Prompt 5:

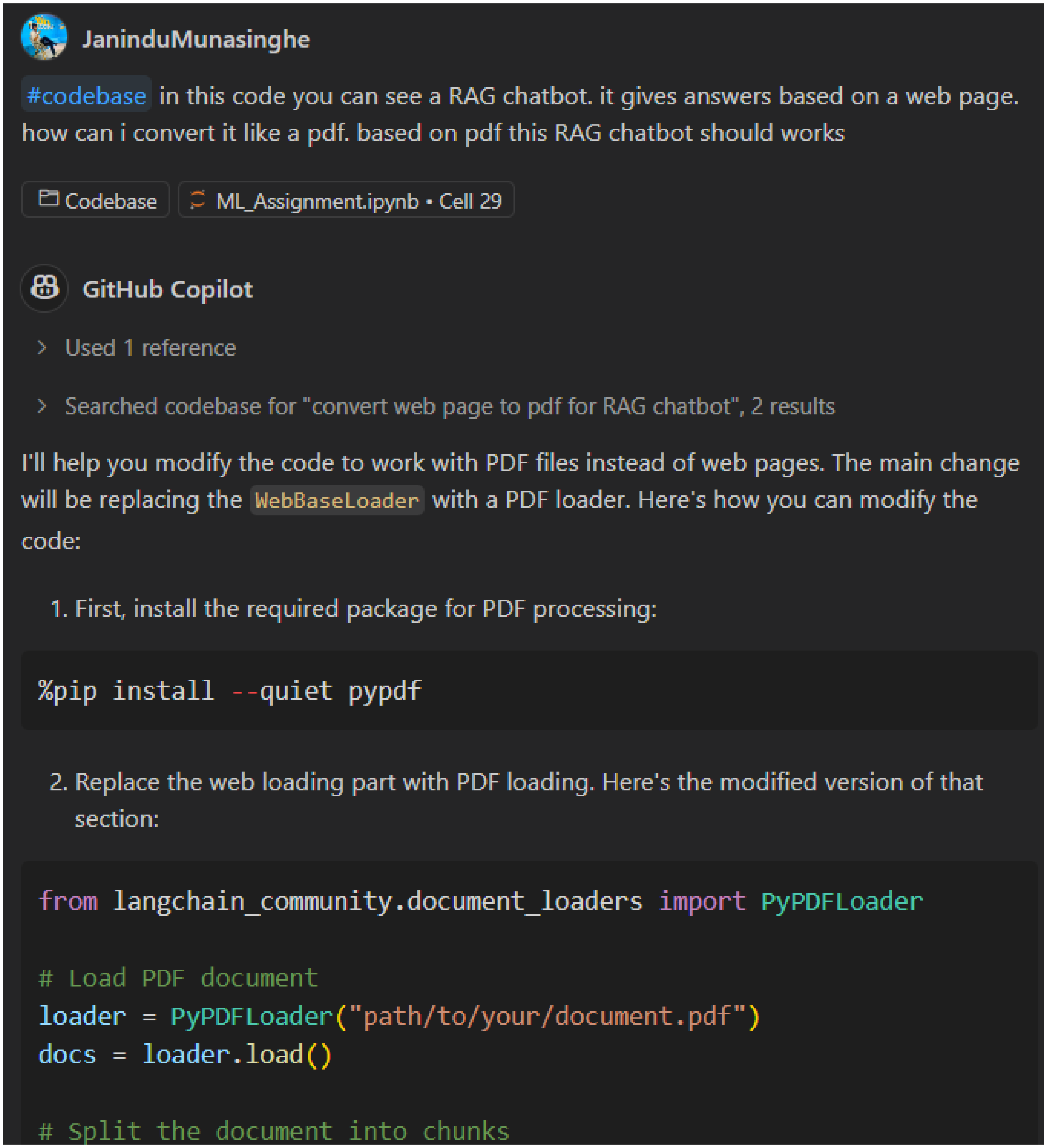
Prompt:

- "How can I create a simple retrieval-based QA chatbot using LangChain with OpenAI?"

Usage & Output Summary:

- The response provided a working example of how to build a RetrievalQA chain that combines the retriever and LLM into a single interface for answering user queries.

6.2 Sample Screenshots of Prompts



JaninduMunasinghe

#codebase in this code you can see a RAG chatbot. it gives answers based on a web page. how can i convert it like a pdf. based on pdf this RAG chatbot should works

Codebase | ML_Assignment.ipynb • Cell 29

GitHub Copilot

> Used 1 reference

> Searched codebase for "convert web page to pdf for RAG chatbot", 2 results

I'll help you modify the code to work with PDF files instead of web pages. The main change will be replacing the `WebBaseLoader` with a PDF loader. Here's how you can modify the code:

1. First, install the required package for PDF processing:

```
%pip install --quiet pypdf
```

2. Replace the web loading part with PDF loading. Here's the modified version of that section:

```
from langchain_community.document_loaders import PyPDFLoader

# Load PDF document
loader = PyPDFLoader("path/to/your/document.pdf")
docs = loader.load()

# Split the document into chunks
```

JaninduMunasinghe

is this code has coversational memory

ML_Assignment.ipynb • Cell 29

GitHub Copilot

> Used 1 reference

Looking at the code, it does have a form of memory implemented through the `MemorySaver` class, but it's not a true conversational memory that maintains the context of previous conversations. Here's how we can add conversational memory:

1. First, let's add the required imports and create a conversation memory:

```
from langchain.memory import ConversationBufferMemory
from langchain_core.chat_history import BaseChatMessageHistory
from langchain_core.prompts import MessagesPlaceholder

# Create conversation memory
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True
)
```

instaed of loading content from blog post, how can i load content from pdf

To load content from a **PDF** instead of a blog post, you can use `PyPDFLoader` or another PDF loader from the `langchain_community.document_loaders` module. Here's how you can update your code:

✅ Replace the web loader section with PDF loading:

python Copy Edit

```python
from langchain_community.document_loaders import PyPDFLoader

# Load and chunk contents of the PDF
loader = PyPDFLoader("path/to/your/file.pdf")  # Provide your local or remote PDF path
docs = loader.load()
```

how can we provide multiple pdfs

To provide **multiple PDFs**, you can load them in a loop and combine the documents into a single list before splitting and indexing. Here's how you can modify your code to handle multiple PDFs:

✅ Step-by-step approach:

python Copy Edit

```python
from langchain_community.document_loaders import PyPDFLoader

# List of PDF file paths
pdf_files = [
    "path/to/first.pdf",
    "path/to/second.pdf",
```

7. References

- LangChain Documentation: RAG Tutorial. https://python.langchain.com/docs/tutorials/rag/
- OpenAI API Documentation. https://platform.openai.com/docs/overview
- AstraDB Documentation. https://docs.datastax.com/en/astra-db-serverless/index.html

8. Video Link

Video –
https://drive.google.com/file/d/1lNLGFvhJODmHcgHIdqDmqg6tmFndem3X/view?usp=sharing