Development of a Chatbot using LangChain with Predefined PDFs

Course Title: Artificial Intelligence

Group Members;

> **Amanda Ofori - 10201100146**
>
> **Yineteili Abii - 10201100115**
>
> **Kenneth Tetteh - 10201100073**

Link to Git Repo: https://github.com/Amanda-Ofori/Langchain_Chatbot.git

Overview

In this project, we will investigate the creation of a chatbot with LangChain, a framework for developing language-based apps. The chatbot will be programmed to engage in discussions with users using data taken from a specified set of PDF documents. The key issue in this endeavour is to process and interpret the material within these papers so that the chatbot can respond to user queries in an informed and correct manner. As part of this literature study, we will look at the methodologies used in four additional publications to obtain insight into current practices. Furthermore, we will explain how our project will differ from existing techniques, emphasising the unique features and improvements we intend to incorporate into our chatbot development.

Introduction to the AI Chatbot Documentation

Welcome to the documentation for the AI Chatbot. This system is designed to streamline the process of extracting text from PDF documents and generating context-based answers using advanced natural language processing models. Built with Python, this solution leverages powerful libraries and frameworks including Streamlit for web interfacing, PyMuPDF for handling PDF operations, and Hugging Face's Transformers for deploying state-of-the-art language models.

This documentation provides a comprehensive overview of the components and functionalities of our system, explaining the intricate details of how we handle PDF documents and interactively generate answers to user queries based on the content extracted from these documents.

Literature Review

In this chapter, four articles will be reviewed in order to discover gaps and areas of improvements in various methodologies described to build a chatbot with LangChain. Most of these articles have tutorial guidelines and steps to achieve to building a chatbot with LangChain, and these articles each have specific goals that have been set out to be achieved. These goals obviously affect the overall methods taken but the primary feature that makes each of these articles relevant for the study is the fact that PDFs are being used to acquire data and LangChain package is used in developing the Chatbot based on the extracted data from the PDFs.

[1]Building a Multi-Document Reader and Chatbot With LangChain and ChatGPT

This excerpt describes the creation of chatbots that interact with PDFs and other documents via embeddings, vector storage, and LangChain. Initially, the method entails extracting text from a document and querying a Large Language Model (LLM), such as ChatGPT. For many documents or huge texts, embeddings and vector storage are utilized to manage relevant information and overcome LLM token limitations. LangChain makes this easier by offering out-of-the-box capabilities for document interaction, such as chat history and managing multiple document kinds. The article also discusses prospective enhancements, such as overcoming token constraints and investigating optimizations for specific content kinds.

Gaps and Improvements:

The existing technique has constraints around the LLM token limit, which limits the amount of context that can be transmitted in a single request. This might be addressed in your project by looking into different LLMs with higher token limits or implementing methods for more efficient context selection. Furthermore, more advanced natural language processing techniques could be used to better comprehend document semantics, reducing the need for embeddings and vector storage for relevance determination. Tailoring the chatbot's capabilities to specific document types, such as CVs or user manuals, may improve its effectiveness by allowing for more targeted optimizations.

[2]How To Build A LangChain PDF Chatbot

The article delves at the possibility of generative AI and conversational AI technology for developing chatbots that interact with document files such as PDFs. It presents Large Language Models (LLMs) such as ChatGPT, Llama, Palm2, and Cohere, which train on massive volumes of textual data to anticipate the next sequence of words. However, ChatGPT has limitations, such as restricted knowledge, hallucinations, and inability to access private data. The Langchain framework is proposed to address these restrictions by providing a flexible and context-aware platform for developing LLM-powered apps. The essay walks users through the steps of creating a Langchain PDF chatbot, from installing the required Python libraries to querying the PDF. It emphasizes the use of vector storage, text splitting, and embeddings to effectively handle.

Gaps and Improvements:

The post lays a strong framework for developing a Langchain PDF chatbot, but it might go farther into improving the chatbot's efficiency, particularly for large or complex documents. Improvements could include looking into advanced text splitting techniques to preserve more context or experimenting with different embedding models to improve semantic understanding. Furthermore, incorporating natural language processing (NLP) techniques to increase question understanding and response production could boost the chatbot's accuracy. These improvements could be integrated into your project to build a more robust and intelligent chatbot.

[3]Multiple-PDF Chatbot using Langchain

The project's goal is to develop a question-answering system based on information retrieval, with PDF documents as a source. The system will be deployable through several interfaces, including Gradio UI, Streamlit UI, FastAPI, and Azure Endpoint. The method includes installing the required libraries, such as langchain, torch, sentence_transformers, and faiss-cpu. The system uses langchain for text production, HuggingFaceEmbeddings for text embeddings, LlamaCpp for neural language modeling, and FAISS to provide a vector index for efficient searches. PyPDFDirectoryLoader loads PDF files, while RecursiveCharacterTextSplitter splits text into chunks. The system extracts relevant text chunks for a given query and uses a neural language model to provide a response. The final phase entails creating an interactive loop where the user can input inquiries and receive answers.

Gaps and Improvements:

While the project provides a solid framework for a question-and-answer system, there is room for development. Incorporating more powerful natural language processing algorithms could help the system better understand difficult inquiries and respond more accurately. Additionally, investigating various embedding and language models could improve the system's performance. Integrating a user feedback mechanism to fine-tune the system's answers over time may also improve its effectiveness. These enhancements could be applied into your project to create a more robust and intelligent question-answering mechanism.

[4]Building A Custom Chatbot To Query PDF Documents With Langchain

This tutorial shows how to build a Langchain chatbot that can answer queries based on the content of PDF documents. The process entails importing required libraries, loading documents with the Directory Loader, splitting documents into manageable chunks, generating embeddings for text data, initializing the language model (OpenAI's GPT-3), configuring a Question-Answer (QA) system, and asking the chatbot questions. The tutorial emphasizes the advantages of better data accessibility, faster insights, targeted solutions, and increased efficiency. Langchain's AI-powered chatbots allow users to swiftly extract information from PDFs and make data-driven decisions with ease.

Gaps and Improvements:

The training could be more specific in explaining how to handle different types of requests and improve the chatbot's accuracy. Using advanced natural language processing techniques and fine-tuning the language model may increase the chatbot's comprehension of complex inquiries. Furthermore, investigating ways to scale the chatbot for huge datasets and integrating it into existing systems or workflows could increase its usefulness in real-world applications. These enhancements could be integrated into your project to build a more sophisticated and adaptable chatbot.

Summary of Findings:

Chatbot Development with Langchain

- The articles collectively emphasize the development of chatbots using Langchain, a toolkit that leverages OpenAI's GPT models for natural language understanding.
- The chatbots are designed to interact with users based on information extracted from PDF documents, making them suitable for various applications such as knowledge retrieval and document querying.

Step-by-Step Implementation:

- The tutorials provide a detailed step-by-step process for chatbot development, including importing libraries, loading documents, splitting text into manageable chunks, generating embeddings, and setting up a Question-Answer (QA) system.
- Key components such as Langchain, OpenAI's GPT-3 model, text embeddings, vector stores, and text splitters are integral to the implementation.

Question Answering System:

- A significant focus is on creating a question answering system that can retrieve relevant information from PDF documents based on user queries.
- The system involves processing text data, generating embeddings for efficient querying, and utilizing a neural language model to generate responses.

Deployment and Interaction:

- The chatbots can be deployed via various interfaces such as Gradio UI, Streamlit UI, FastAPI, and Azure Endpoint, providing flexibility in how users interact with the system.
- The tutorials highlight the importance of attributing sources to the answers, enhancing the transparency and reliability of the information provided.

Benefits and Applications:

- The articles highlight the benefits of using Langchain for chatbot development, including enhanced data accessibility, instant insights, tailored solutions, and improved efficiency.
- Chatbots have potential applications in areas such as research, knowledge management, and data-driven decision-making.

Gaps and Areas for Improvement:

- While the tutorials provide a comprehensive guide, there is room for improvement in handling complex queries and refining the chatbot's responses for greater accuracy.
- Incorporating advanced natural language processing techniques and fine-tuning the language model could enhance the chatbot's understanding of user queries.

- Exploring scalability and integration with existing systems could broaden the applicability of the chatbots in real-world scenarios.

Conclusion:

Finally, the literature analysis looked at several techniques and innovations for developing chatbots with Langchain, with a focus on their use in extracting and processing information from PDF documents. The step-by-step instructions and frameworks mentioned are extremely useful for implementing natural language processing techniques and integrating complex AI models such as OpenAI's GPT-3. These chatbots present intriguing solutions for improved data accessibility and quick knowledge retrieval, with potential applications across a wide range of areas. As we move forward, improving these systems' ability to handle complex queries and integrating them into larger workflows will be critical to maximizing their impact and utility in real-world circumstances.

AI Chatbot Documentation

Installation of Packages

This code snippet imports the neccessary packages needed for the project. The overall imports suggest that the script is likely involved in text processing or manipulation tasks, possibly involving the handling of PDF files, performing natural language processing, and handling text data in different character encodings.

1. `import os`
    - The `os` module in Python is a standard utility module that provides functions to interact with the operating system. It includes functionality for creating and removing a directory (folder), fetching its contents, managing paths, reading and writing files, etc.
2. `import json`
    - The `json` module allows you to encode and decode JSON data. JSON (JavaScript Object Notation) is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript).
3. `import fitz  # PyMuPDF`
    - `fitz` is a commonly used nickname for the PyMuPDF library, which is a Python binding for MuPDF – a lightweight PDF and XPS viewer. PyMuPDF provides functionalities for accessing and modifying PDF files, extracting information from them, and even creating new PDF files.
4. `from langchain.text_splitter import RecursiveCharacterTextSplitter`
    - This import statement brings in the `RecursiveCharacterTextSplitter` class from the `langchain.text_splitter` module, which is part of the LangChain library. This specific class is likely used for splitting text into smaller chunks, potentially using a recursive method that breaks down text to a certain character limit, useful in handling large texts in processing tasks such as language modeling or translation.
5. `from transformers import GPT2Tokenizer, GPT2LMHeadModel`
    - The `transformers` library, developed by Hugging Face, is popular for working with state-of-the-art machine learning models, specifically those designed for natural language processing (NLP). The `GPT2Tokenizer` is used for tokenizing text into a format suitable for input into the GPT-2 model, and the `GPT2LMHeadModel` is an actual pre-trained GPT-2 model designed for language modeling tasks (generating text).
6. `import glob`
    - The `glob` module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell. It is used for file handling in Python, allowing you to list files within a directory that match a certain pattern.

you to list files within a directory that match a certain pattern.

7. `import chardet  # Import chardet for encoding detection`

    - The `chardet` module is used for character encoding detection. This is crucial in data processing tasks where the input data's encoding might not be known beforehand. It helps determine the correct encoding to use for reading files, ensuring that text data is correctly processed and avoiding issues like misinterpretation of characters.

8. `import sys`

    - The `sys` module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is commonly used to manipulate the Python runtime environment. For instance, it can be used to retrieve system-specific parameters and functions, handle command-line arguments, or exit from the program.

In [ ]:
```python
import os
import json
import fitz  # PyMuPDF
from langchain.text_splitter import RecursiveCharacterTextSplitter
from transformers import GPT2Tokenizer, GPT2LMHeadModel
import glob
import chardet  # Import chardet for encoding detection
import chardet
import sys
```

Explanation of the PDFDocumentProcessor Class and Auxiliary Function

The `PDFDocumentProcessor` class and the auxiliary function `read_file_safely` defined in the script work together to manage and process PDF documents within a specified directory. The class is structured to extract text from PDFs, split the text into manageable chunks, and then save the processed text into a separate directory as text files. The auxiliary function provides a safe method to read files with unknown character encodings. Below is a detailed explanation of each part:

PDFDocumentProcessor Class

1. **Initialization (`__init__` Method):**
   - The constructor takes two parameters: `directory` and `storage_directory`. `directory` is where the PDF files are located, and `storage_directory` is where the processed files will be saved.
   - An instance of `RecursiveCharacterTextSplitter` is created and stored as an instance variable. This splitter is likely used to handle large blocks of text that need to be divided into smaller, more manageable pieces.
   - The `ensure_directory_exists` method is immediately called with `storage_directory` to make sure the target directory for storing processed files exists. If it doesn't, it is created.
2. **Ensure Directory Exists (`ensure_directory_exists` Method):**
   - This method checks if the specified `directory` exists on the file system. If not, it creates the directory using `os.makedirs(directory)`, ensuring that all intermediate-level directories needed to contain the leaf directory are also created.
3. **Extract Text (`extract_text` Method):**
   - This method processes each PDF file in the specified `directory`. It looks for files that end with a `.pdf` extension.
   - For each PDF file found, it opens the file using `fitz.open` and extracts all text from each page, concatenating them into a single string.
   - The complete text of the PDF is then passed to `text_splitter.split_text`, which presumably splits the text into segments suitable for further processing or storage.
   - The segmented text is passed along with the filename to the `store_document` method to handle the actual file writing.
4. **Store Document (`store_document` Method):**
   - This method takes the processed text (`document`) and the original filename, modifies the filename by replacing the `.pdf` extension with `.txt`, and writes the text segments to a new file in the `storage_directory`. Each segment of text is written on a new line.

written on a new line.

- The text file is saved with UTF-8 encoding, ensuring wide compatibility and proper handling of various characters and symbols present in the text.

Auxiliary Function: read_file_safely

- **Purpose:** The function is designed to read files safely, handling different character encodings robustly. This is important when dealing with text data where the encoding is not known beforehand.
- **Process:**
  - It reads the file as binary data.
  - Uses the `chardet.detect` function to determine the likely character encoding of the binary data.
  - Attempts to decode the binary data using the detected encoding. If decoding fails (e.g., due to a `UnicodeDecodeError`), it falls back to UTF-8 encoding and ignores any errors that occur during decoding.

This class and function together provide a robust framework for processing and handling text data extracted from PDF files, ensuring data integrity and handling potential issues with file encoding gracefully. The use of the `RecursiveCharacterTextSplitter` suggests an application where handling large volumes of text efficiently and reliably is a key requirement.

In [ ]:
```
class PDFDocumentProcessor:
    def __init__(self, directory, storage_directory):
        self.directory = directory
        self.storage_directory = storage_directory
        self.text_splitter = RecursiveCharacterTextSplitter()
        self.ensure_directory_exists(self.storage_directory)

    def ensure_directory_exists(self, directory):
        """Ensure the storage directory exists."""
        if not os.path.exists(directory):
            os.makedirs(directory)

    def extract_text(self):
        """Extract text from PDF files in the directory and store it."""
        for filename in os.listdir(self.directory):
            if filename.endswith('.pdf'):
                path = os.path.join(self.directory, filename)
                with fitz.open(path) as doc:
                    text = " ".join(page.get_text() for page in doc)
                tokenized_text = self.text_splitter.split_text(text)
                self.store_document(tokenized_text, filename)

    def store_document(self, document, filename):
        """Store the tokenized document in a text file using UTF-8 encoding."""
        file_path = os.path.join(self.storage_directory, filename.replace('.pdf', '
        with open(file_path, 'w', encoding='utf-8') as file:
            for item in document:
                file.write("%s\n" % item)
```

```
def read_file_safely(file_path):
    # Read the file as binary data for detection
    with open(file_path, 'rb') as file:
        raw_data = file.read()

    # Detect the encoding
    detected = chardet.detect(raw_data)
    encoding = detected['encoding'] if detected['encoding'] is not None else 'utf-8

    try:
        # Attempt to read with detected encoding
        return raw_data.decode(encoding)
    except UnicodeDecodeError:
        # Fallback to UTF-8 and ignore errors
        return raw_data.decode('utf-8', errors='ignore')
```

- This method initializes the `CustomRAGProcessor` instance with a specified model name (for the GPT-2 model) and a storage directory where text documents are stored.
- It loads a GPT-2 tokenizer and model using `GPT2Tokenizer.from_pretrained` and `GPT2LMHeadModel.from_pretrained`, respectively. These are essential components for encoding text into a format that the GPT-2 model can process and for generating text based on input prompts.

2. **Detect Encoding (`detect_encoding` Method):**

   - This method opens a file in binary mode, reads the raw data, and uses the `chardet.detect` function to determine the probable character encoding of the file. This functionality is crucial for correctly processing files that may not use standard UTF-8 encoding.

3. **Retrieve Documents (`retrieve_documents` Method):**

   - This method searches for text files within the specified storage directory and checks if they contain the query string. It leverages `glob.glob` to find all `.txt` files in the directory.
   - Each file's content is read safely using the `read_file_safely` function (defined elsewhere, as seen in previous explanations), which handles various encodings and potential decoding errors.
   - If the query is found in a document, that document is added to a list of relevant documents, which is then returned.

4. **Generate Answer (`generate_answer` Method):**

   - This method takes a user's query and optionally a maximum length for the generated answer. It retrieves documents related to the query using `retrieve_documents`.
   - The retrieved documents are concatenated into a single string, which serves as the context for the query.
   - To ensure the total input length does not exceed the model's limits, the context is truncated if necessary.

- The full input to the model is structured as "Context: [context] Question: [query]", which is then encoded into model-ready tokens.
- Using the GPT-2 model, it generates an answer by setting specific parameters like `max_length`, `max_new_tokens`, `top_p`, `top_k`, and `temperature` to control the randomness and focus of the answer generation.
- Finally, the generated tokens are decoded back into a readable string, stripping away any special tokens that are typically used by the model for internal processing.

Example Usage

- **PDFDocumentProcessor Initialization and Text Extraction:**
  - An instance of `PDFDocumentProcessor` is created with directories for source documents and storage.
  - The `extract_text` method is called to process all PDF files in the specified directory, extracting text and storing it as `.txt` files.
- **CustomRAGProcessor Initialization and Answer Generation:**
  - An instance of `CustomRAGProcessor` is created with the model name 'gpt2' and the same storage directory.
  - The `generate_answer` method is invoked with a specific query ("What is the topic?"), generating an answer based on the text documents that contain relevant content.
- **Print the Generated Answer:**
  - The answer returned by `generate_answer` is printed, showcasing the application of the GPT-2 model in generating context-aware responses based on the content found in the processed documents.

This system demonstrates an advanced use of machine learning models for NLP tasks, combining document retrieval and AI-driven text generation to provide meaningful answers to user queries based on stored textual data.

In [ ]:
```
class CustomRAGProcessor:
    def __init__(self, model_name, storage_directory):
        self.tokenizer = GPT2Tokenizer.from_pretrained(model_name)
        self.model = GPT2LMHeadModel.from_pretrained(model_name)
        self.storage_directory = storage_directory

    def detect_encoding(self, file_path):
        with open(file_path, 'rb') as file:
            raw_data = file.read()
        result = chardet.detect(raw_data)
        return result['encoding']

    def retrieve_documents(self, query):
        documents = []
        for file_path in glob.glob(os.path.join(self.storage_directory, '*.txt')):
            document = read_file_safely(file_path)
            if query.lower() in document.lower():
```

```
                documents.append(document)
        return documents

    def generate_answer(self, query, max_length=100):
        documents = self.retrieve_documents(query)
        combined_context = " ".join(documents)
        # Truncate context to fit within model's maximum input size if necessary
        max_context_length = max_length - len(query) - 50  # reserve space for quer
        if len(combined_context) > max_context_length:
            combined_context = combined_context[:max_context_length]

        input_text = f"Context: {combined_context} Question: {query}"
        input_ids = self.tokenizer.encode(input_text, return_tensors="pt", max_leng

        # Adjust generation parameters for more coherent generation
        output_ids = self.model.generate(
            input_ids,
            max_length=max_length,
            max_new_tokens=150,  # adjust as needed
            num_return_sequences=1,
            top_p=0.92,  # nucleus sampling, less randomness
            top_k=50,  # top-k sampling
            temperature=0.7  # lower temperature makes outputs less random
        )
        return self.tokenizer.decode(output_ids[0], skip_special_tokens=True)

# Example usage
processor = PDFDocumentProcessor('Documents', 'path/to/storage')
processor.extract_text()

rag_processor = CustomRAGProcessor('gpt2', 'path/to/storage')
answer = rag_processor.generate_answer("What is the topic?")
print(answer)
```

2. **Input Settings**:

   - `directory`: A text input field in the sidebar where users can specify the directory containing the PDF files.
   - `storage_directory`: A text input field in the sidebar for specifying the directory where processed files will be stored.
   - `model_name`: A text input field in the sidebar to specify the name of the model to use for the Q&A system (e.g., 'gpt2').

3. **Process PDFs**:

   - `process_button`: A button in the sidebar labeled "Process PDFs". When clicked:
     - An instance of `PDFDocumentProcessor` is created with the user-specified `directory` and `storage_directory`.
     - The `extract_text` method of `PDFDocumentProcessor` is called to process the PDFs and store the extracted text.
     - A success message is displayed using `st.success` indicating that the PDFs have been processed and stored successfully.

4. **Generate Answer**:

   - `query`: A text input field where users can type a question.

- A button labeled "Generate Answer". When clicked:
  - If a query is provided, an instance of `CustomRAGProcessor` is created using the user-specified `model_name` and `storage_directory`.
  - The `generate_answer` method of `CustomRAGProcessor` is called to generate an answer based on the query and the content of the processed PDFs.
  - The answer is displayed on the main page under the label "Answer:".
  - If no query is entered, a message prompts the user to enter a question.

Execution Trigger

- The script checks if it is run as the main program (`if __name__ == "__main__":`). If true, it calls the `main()` function to start the application.

Usage and Interaction

When run, this Streamlit application provides a user-friendly web interface allowing users to:

- Enter the directory paths for PDFs and storage.
- Process PDF documents to extract and store text.
- Enter a question and generate an answer based on the extracted text using a machine learning model.

This setup is practical for scenarios where users need to interactively process documents and obtain information without delving into command-line tools or complex interfaces, leveraging the capabilities of modern NLP models directly from a web application.

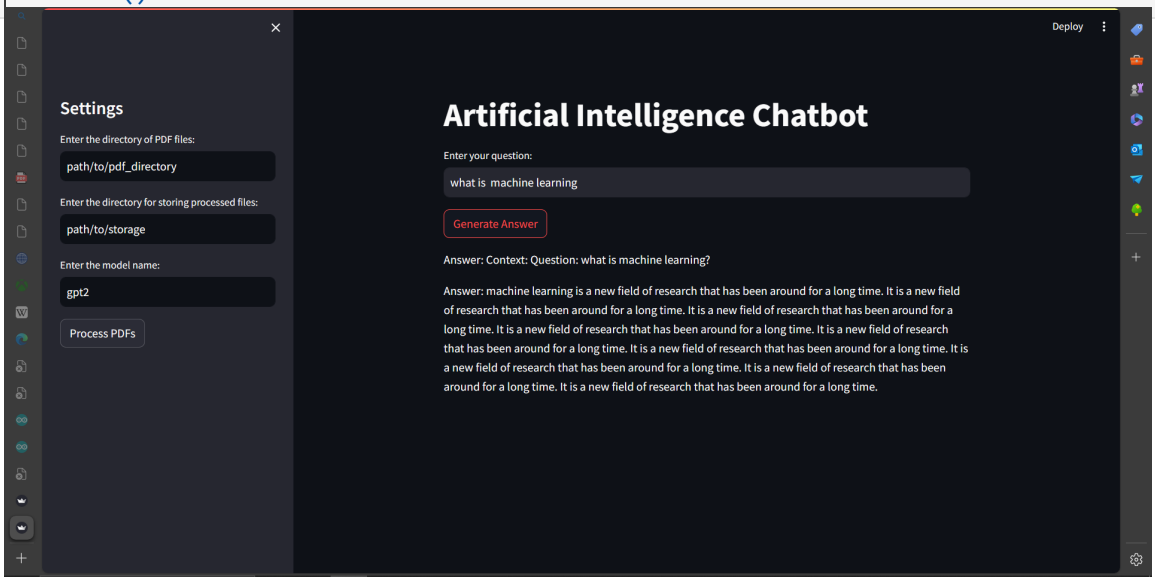
In [ ]:
```
import streamlit as st
from langchainchatbot_backend import PDFDocumentProcessor, CustomRAGProcessor

def main():
    st.title('PDF Document Processor and Q&A System')
    st.sidebar.title("Settings")
    directory = st.sidebar.text_input("Enter the directory of PDF files:", "path/to
    storage_directory = st.sidebar.text_input("Enter the directory for storing proc
    model_name = st.sidebar.text_input("Enter the model name:", "gpt2")

    process_button = st.sidebar.button("Process PDFs")
    if process_button:
        processor = PDFDocumentProcessor(directory, storage_directory)
        processor.extract_text()
        st.success("PDFs processed and stored successfully.")

    query = st.text_input("Enter your question:")
    if st.button("Generate Answer"):
        if query:
            rag = CustomRAGProcessor(model_name, storage_directory)
            answer = rag.generate_answer(query)
            st.write("Answer:", answer)
        else:
            st.write("Please enter a question to generate an answer.")
```

```
if __name__ == "__main__":
    main()
```



References:

[1] Sami Maameri, “Building a Multi-Document Reader and Chatbot With LangChain and ChatGPT,” Medium, May 20, 2023. https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339

[2] Pieces 🌟, “How to Build a Langchain PDF Chatbot,” Pieces for Developers, Oct. 31, 2023. https://medium.com/getpieces/how-to-build-a-langchain-pdf-chatbot-b407fcd750b9 (accessed Mar. 20, 2024).

[3] C. C, “Multiple-PDF Chatbot using Langchain,” **AI monks.io**, Oct. 22, 2023. https://medium.com/aimonks/multiple-pdf-chatbot-using-langchain-b3ee2296b1a7 (accessed Mar. 20, 2024).

[4]“Build Custom Chatbot For PDFs With Langchain,” Bluebash Blog, Oct. 30, 2023. https://www.bluebash.co/blog/custom-chatbot-to-query-pdf-documents-with-langchain/amp/ (accessed Mar. 20, 2024).