EFFICIENT OPEN DOMAIN QUESTION AND ANSWERING SYSTEM THROUGH LLM DRIVEN RETRIEVAL AUGMENTED GENERATION

ABSTRACT

Traditional AI models generate responses based on pre-trained or publicly available data, often leading to issues like hallucinations, irrelevant answers, and security risks when dealing with private or domain-specific information. Users require a system that retrieves and generates responses solely from their uploaded documents, ensuring data privacy and eliminating inaccuracies. A Retrieval-Augmented Generation (RAG) system addresses these concerns by allowing users to upload and process their own files, ensuring that responses strictly adhere to the provided data. The system extracts and stores information locally, minimizing AI hallucinations and enhancing reliability. This approach is particularly beneficial in fields such as enterprise knowledge management, legal analysis, healthcare, finance, and education, where precise and context-aware responses are critical. By integrating retrieval mechanisms with generative AI, RAG ensures that AI-generated content is directly tied to verified sources, reducing misinformation. Furthermore, the system enhances security by keeping sensitive data within a controlled environment rather than relying on external datasets. Unlike traditional AI models that rely on static training data, RAG dynamically retrieves relevant information, making it adaptable to new and evolving knowledge bases. This makes it an ideal solution for businesses and organizations that require real-time, accurate, and secure AI-driven insights.

Keywords:

Traditional AI models, pre-trained data, hallucinations, security risks, data privacy, RAG, uploaded documents,local storage, enterprise knowledge, legal, healthcare, finance, education, context-aware responses, verified sources, misinformation reduction, real-time insights, secure AI.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Artificial Intelligence (AI) has revolutionized various industries by providing automated solutions for information retrieval and content generation. However, traditional AI models rely on pre-trained or publicly available datasets, often leading to issues such as hallucinations, irrelevant responses, and security risks when handling private or domain-specific information. These limitations make it challenging for businesses and professionals in fields like enterprise knowledge management, legal analysis, healthcare, finance, and education to fully trust AI-generated outputs.

To address these challenges, Retrieval-Augmented Generation (RAG) systems have emerged as a reliable solution. Unlike conventional AI models, RAG combines information retrieval with generative AI, ensuring that responses are strictly based on user-provided data. By allowing users to upload and process their own documents, RAG systems eliminate inaccuracies, enhance data privacy, and prevent AI hallucinations. Additionally, the system enables users to upload PDFs and text files, process and store them in a private vector database, and retrieve the most relevant information using AI-driven embeddings and similarity search. The local storage of extracted information ensures secure and contextually relevant responses.

This project explores the significance of RAG systems in overcoming the limitations of traditional AI models. It highlights how RAG enhances AI-driven insights through real-time, dynamic information retrieval while maintaining security and accuracy. The discussion will focus on its applications across various industries and its potential to transform AI-assisted decision-making.

1.1 EXISTING SYSTEM

The current AI systems primarily rely on pre-trained models and external data sources to generate responses. These models are trained on large public datasets, which makes them ineffective for domain-specific or private data processing. Since they do not dynamically retrieve information fro user-provided files, they often generate inaccurate or irrelevant responses when dealing with specialized fields such as legal analysis, finance, or healthcare.

One of the major concerns with the existing AI systems is data privacy and security. Most traditional AI solutions process queries on external servers, where user data is sent, analyzed, and stored. This exposes sensitive information to potential security risks, as these servers may log, retain, or even share the data with third parties without user consent. In industries where confidentiality is critical such as corporate business operations, healthcare records, or legal proceedings this lack of control over data security makes traditional AI solutions unreliable and unsuitable.

Another issue is the lack of transparency and user control over data processing. Once data is uploaded to an AI system, users have no insight into how it is stored, used, or shared. Many cloud-based AI models operate under proprietary frameworks where the specifics of data handling remain unclear.

Furthermore, existing AI systems are heavily dependent on cloud-based infrastructure, requiring a constant internet connection to function. They fetch and process data from external servers, meaning they are inaccessible in remote or offline environments where connectivity is limited or unavailable. This makes them unsuitable for industries and use cases that require AI functionality in offline or restricted-access settings.

These limitations highlight the need for AI frameworks, such as Retrieval-Augmented Generation (RAG) systems, which provide control, security, and contextual accuracy by allowing users to process their own data securely within a controlled environment.

1.2 PROPOSED SYSTEM

The proposed system leverages **Retrieval-Augmented Generation (RAG)**, ensuring secure, domain-specific, and private data processing. Unlike conventional AI models that rely on pre-trained datasets and external sources, this system allows users to upload and process their own files, ensuring that AI-generated responses are based completely on the provided data. This eliminates the risks of irrelevant or hallucinated responses and makes the system highly reliable for industries requiring precise and context-aware AI outputs, such as legal analysis, healthcare, finance, and enterprise knowledge management.

A key feature of the proposed system is its local data processing and private storage mechanism. Instead of relying on external cloud-based servers, all documents are processed and stored in a private vector database, ensuring that user data remains secure and confidential. The system utilizes AI-driven embeddings and similarity search to retrieve the most relevant information from uploaded PDFs and text files, eliminating dependence on external sources. This approach not only enhances security but also ensures that responses remain highly relevant to the user's specific requirements.

Additionally, the proposed system gives users complete control over their data. Unlike existing AI models, which lack transparency in data handling, this system ensures that users have full visibility into how their data is stored, processed, and accessed.

Another significant advantage of the proposed system is its offline capability. Since all processing and retrieval occur locally, it does not require a constant internet connection, making it accessible even in remote or restricted environments. This is particularly beneficial for industries that operate in offline or low-connectivity settings, such as military operations, disaster response, and secure corporate environments.

By integrating RAG-based retrieval, private data storage, enhanced security, and offline functionality, the proposed system provides an efficient, privacy-focused, and contextually accurate AI solution.

1.3 LITERATURE SURVEY

[1]Title: Retrieval-Augmented Generation With Inverted Question Matching for Enhanced QA Performance, 2024.

Authors: Binita Saha

The document outlines a new architecture called QuIM-RAG (Question-to-question Inverted Index Matching) designed to enhance Retrieval-Augmented Generation (RAG) systems for improved Question Answering (QA) performance.

The primary issue addressed in this research is the ineffectiveness of traditional RAG systems in accurately answering domain-specific questions due to challenges such as information dilution and hallucination.

[2] Title: QPaug: Question and Passage Augmentation for Open-Domain Question Answering of LLM, 2024.

Authors: Minsang Kim, Cheoneum Park, and Seungjun Baek

The paper "QPaug: Question and Passage Augmentation for Open-Domain Question Answering of LLMs" introduces a method to enhance Open-Domain Question Answering (ODQA) by improving retrieval and answer extraction. The proposed QPaug framework consists of Question Augmentation (Qaug), which breaks down complex questions into sub-questions for better retrieval, and Passage Augmentation (Pgen), which supplements retrieved passages with self-generated content from LLMs .

However, drawbacks include reliance on LLM knowledge, potential hallucinations, increased computational cost, and limited control over generated passages.

2. SYSTEM ANALYSIS

2.1 FUNCTIONAL REQUIREMENTS

1. Document Upload: Ability to upload documents in various formats, some PDFs are retrievable from URLs, allowing users to process online documents securely.

2. Chunking: Splitting large documents into smaller, meaningful chunks for indexing.

3. Embedding Generation: Compute vector embeddings for document chunks using a pre-trained embedding model.

4. Query Handling: Generate query embeddings, perform similarity searches, and rank results**.**

5. Contextual Prompt Creation: Combine retrieved information with user queries to create prompts for the language model.

6. Answer Generation: Generate context-aware answers using a language model

2.2 NON-FUNCTIONAL REQUIREMENTS

1. Performance: Ensures fast query processing and retrieve relevant information from uploaded documents without relying on external sources.

2. Scalability: Supports multiple document uploads, PDF retrieval from URLs, and efficient vector-based indexing to handle large datasets.

3. Usability: Provides a user-friendly interface for uploading files, retrieving PDFs from URLs, and querying stored data, ensuring smooth and interactive usage.

4. Accuracy: Generates precise, context-aware responses strictly based on uploaded documents, eliminating AI hallucinations.

5. Error Handling: Implements robust error management, returning appropriate messages for issues like unsupported file formats, missing data, or failed URL retrievals, ensuring the system remains stable without crashing

2.3 HARDWARE REQUIREMENTS

CPU: Provided by Google Colab.

RAM: A minimum of 8 GB RAM (Google Colab provides sample memory for standard operations).

Storage:Harddisk

2.4 SOFTWARE REQUIREMENTS

1. Operating System: Windows 10/11:

- The specified operating system is Windows 11. This indicates that the software is intended to run on the Windows 11 platform. The choice of the operating system can impact compatibility and performance**.**

2. Development Tools: Google Colab or Jupyter Notebook

Google Colab:

- Cloud-based: No need for local installation.
- Pre-configured GPU Support: Useful for models like Whisper and SentenceTransformers.
- Google Drive Integration: Helps store processed files and embeddings persistently

Jupyter Notebook:

- Local Execution: Runs on your local machine for offline processing.
- Interactive Coding: Suitable for debugging and iterative development.
- Works with Anaconda: Can be easily set up with conda for managing dependencies.

3. Programming Language: Python

- Python is the core language used for writing and executing the code.
- It supports AI/ML libraries like FAISS, SentenceTransformers, and Whisper.
- It has strong text-processing capabilities with libraries like LangChain.

4. Libraries and Packages: FAISS, yt_dlp, fpdf, whisper

FAISS (Facebook AI Similarity Search)

- Efficient vector search and similarity matching.
- Stores and retrieves text embeddings using a vector index.

yt_dlp (YouTube Downloader)

- Creates PDF reports from extracted text.
- Converts processed **text-based answers** into a **PDF report**.

Whisper (Speech-to-Text Transcription)

- Converts speech (audio/video) into text

AI Model: all-MiniLM-L6-v2 (SentenceTransformers)

- Converts text into numerical vectors (embeddings)**.**

5. Frameworks: Gradio, LangChain

Gradio (For User Interface)

- Creates a web-based UI for the chatbot.

LangChain (For Document Processing & Retrieval)

- Helps in loading, processing, and retrieving documents.

3. METHODOLOGY

This system implements an AI-powered document-based Q&A chatbot that allows users to upload PDFs and text files, index them using embeddings, and **retrieve relevant** information based on user queries.

Embedding-Based Retrieval (Semantic Search)

- Embedding-based retrieval converts both the document text and the user's question into mathematical vectors (numerical representations) that encode semantic meaning.
- This allows the system to retrieve the most relevant document chunk even if the exact words are different.

Text Embeddings: The system uses a pre-trained sentence transformer (all-MiniLM-L6-v2) to convert text into high-dimensional vectors.

Semantic Similarity Search: When a user asks a question, the system converts it into a vector and finds the most similar document chunk in the FAISS vector store.

Ranking Results: The most relevant chunk is returned as an answer if its similarity score meets the threshold.

Libraries Used:

sentence-transformers**:**The sentence-transformers library is a powerful NLP tool designed to convert sentences, paragraphs, or documents into high-dimensional embeddings that capture semantic meaning. Built on top of Hugging Face's transformers and PyTorch/TensorFlow, it enables tasks like semantic search, text similarity, clustering, and retrieval-based question answering. It supports models such as all-MiniLM-L6-v2, optimized for fast and efficient embeddings. The library integrates seamlessly with FAISS, making it ideal for AI-driven applications like chatbots, recommendation systems, and document retrieval.

Model:all-MiniLM-L6-v2

The all-MiniLM-L6-v2 model is a compact and efficient sentence embedding model from Sentence-Transformers, designed for semantic search, text similarity, and information retrieval. Based on Microsoft's MiniLM, it has 6 Transformer layers and generates 384-

dimensional embeddings, balancing speed and accuracy for real-time applications.Standard transformer models like BERT-base have 12 layers, while BERT-large has 24 layers. MiniLM reduces this to 6 layers, making it much lighter and faster. The model has been fine-tuned on millions of sentence pairs It is widely used in chatbots, recommendation systems, FAQ retrieval, and clustering tasks, integrating seamlessly with FAISS and Elasticsearch for fast similarity searches. While it may not capture as much context as larger transformer models. Easy to use and highly optimized, all-MiniLM-L6-v2 is a powerful tool for developers building AI-driven NLP applications.

Natural Language Processing (NLP):

NLP (Natural Language Processing) helps the system understand and process human language. In this system, NLP techniques are used to:

- Extract text from PDFs and text files.
- Split large text files into smaller, meaningful chunks.
- Search and retrieve relevant answers from stored documents.

Modules:

langchain_community.document_loaders : Thelangchain_community.document_loader s library in LangChain provides various tools for loading and extracting text from different document formats such as PDFs, plain text, Word documents, and more. It includes specialized loaders like PyMuPDFLoader for PDFs and TextLoader for .txt files, allowing seamless integration of external documents into AI applications. These loaders convert raw files into structured LangChain documents, which can then be processed for semantic search, AI-powered Q&A, and knowledge retrieval. By using langchain_community.document_loaders, developers can efficiently ingest and manage text-based data for AI-driven applications.

langchain_text_splitters: The langchain_text_splitters module in **LangChain** is designed to break large texts into **smaller, structured chunks** for better processing by AI models. Since large documents can overwhelm **language models and vector databases**, this

module ensures that information is split into **manageable segments** while preserving context through overlapping text. A key component is RecursiveCharacterTextSplitter, which intelligently divides text based on sentence structure and character limits. This is essential for semantic search, document retrieval, and chatbot applications, ensuring AI models can efficiently understand and respond to queries without losing important details.

PyMuPDFLoader : PyMuPDFLoader is a powerful tool in LangChain designed to extract text from PDF files. Built on the PyMuPDF (Fitz) library, it efficiently reads both text-based and scanned PDFs, making it ideal for document processing in AI applications. It allows seamless extraction of structured content, which can then be processed for semantic search, AI-powered chatbots, and question-answering systems. This is particularly useful in legal, academic, and corporate environments where PDFs are a primary format for storing information.

TextLoader : TextLoader is another essential module in LangChain, used for processing plain text (.txt) files. Unlike PDFs, text files store unstructured information, which TextLoader reads and converts into LangChain-compatible documents. These extracted texts can then be used for AI-driven search, content analysis, or chatbot interactions. Since AI models require structured inputs, TextLoader plays a crucial role in preparing raw text for further processing, embedding, and retrieval.

RecursiveCharacterTextSplitter: RecursiveCharacterTextSplitter helps AI systems handle large documents efficiently by breaking them into smaller, context-preserving chunks. AI models perform better with shorter, structured inputs, and this module ensures that important context is not lost by introducing overlapping text between chunks. This is especially useful when integrating with vector search databases like FAISS, allowing more accurate search results and AI-driven responses. By combining PyMuPDFLoader, TextLoader, and RecursiveCharacterTextSplitter, developers can build a robust document processing pipeline, enabling AI models to understand and retrieve information effectively.

Vector Database (FAISS) for Efficient Search:

FAISS (Facebook AI Similarity Search) is an open-source library developed by Meta AI for efficient similarity search and clustering of dense vectors. It is widely used for fast nearest-neighbor searches in large-scale datasets, making it ideal for applications like semantic search, and AI-driven document retrieval. FAISS enhances AI responses by efficiently storing and retrieving high-dimensional embeddings. Documents are converted into vector embeddings and indexed in FAISS for fast similarity searches. When a user queries, FAISS retrieves the most relevant documents, which provide context for the LLM to generate accurate and context-aware answers. This enables AI systems to access up-to-date, domain-specific knowledge, improving search accuracy and response relevance in applications like chatbots and document retrieval systems.

4. SYSTEM DESIGN

4.1 SYSTEM ARCHITECTURE:

The architecture followed in this system is the Retrieval-Augmented Generation **(RAG)** pattern, a widely used approach in AI-driven search and retrieval systems. This pattern enhances response accuracy by integrating document retrieval (Vector DB) and language generation (LLM), ensuring context-aware answers. The RAG architecture serves as the foundation for structuring the system, enabling efficient knowledge retrieval, modularity, and scalability.

Query :

- This is the user's input question or request.
- It is converted into vector embeddings using an embedding model before being processed.

Embedding Model:

- A machine learning model that converts text into vector representations (embeddings).
- Used for both document storage and query processing to enable similarity searches.

Embeddings:

- Numerical vector representations of text generated by the embedding model.
- These vectors help in searching and retrieving semantically similar content

Vector DB (Vector Database):

- A specialized database that stores and retrieves vector embeddings.
- It is optimized for similarity searches, allowing the system to fetch relevant content efficiently.
- It stores PDF embeddings and retrieves context relevant to a query.

Fetch Relevant Context:

- The process of retrieving the most relevant document chunks from the Vector DB based on a query.
- Helps the LLM generate accurate and informed responses.

Query + Context:

- Once relevant text chunks are retrieved, they are combined with the user's query.
- This enriched input helps the LLM generate a more accurate response.

LLM (Large Language Model):

- A powerful AI model used for natural language understanding and generation.
- It takes in the query + context and produces a human-like response.

Response:

- The final answer generated by the LLM, based on the provided query and retrieved context.
- This response is returned to the user via a chatbot or interface.

PDF:

- Represents documents uploaded by the user for processing and searching.
- These documents are converted into vector embeddings and stored in the Vector DB.

System Architecture Diagram for the Retrieval-Augmented Generation (RAG) System:

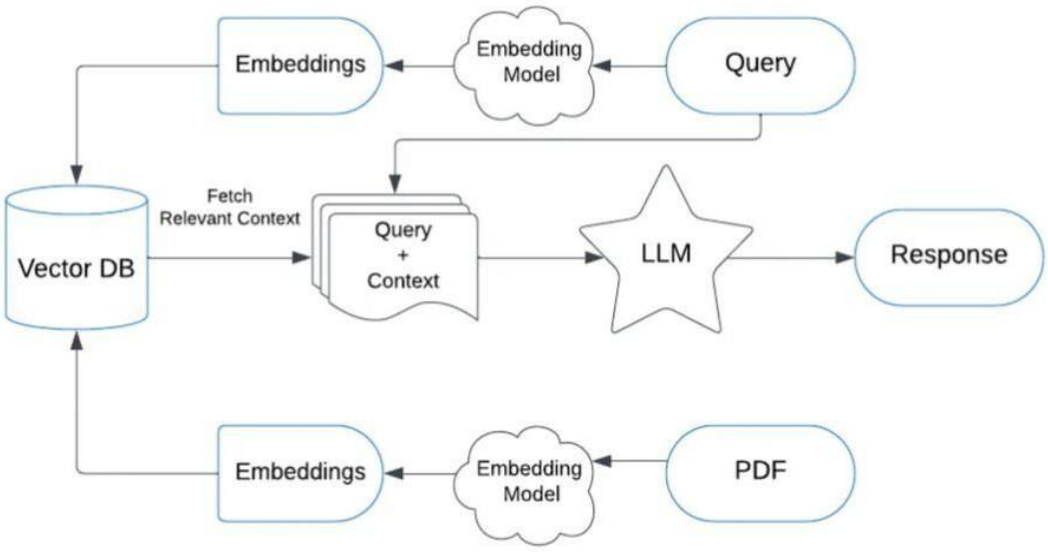

Fig No:4.1 System Architecture for Retrieval-Augmented Generation

5. UML DIAGRAMS

UML is simply anther graphical representation of a common semantic model. UML provides a comprehensive notation for the full lifecycle of object-oriented development.

ADVANTAGES

- To represent complete systems using object oriented concepts
- To establish an explicit coupling between concepts and executable code
- To take into account the scaling factors that are inherent to complex and Critical end.

UML defines several models for representing systems

USECASE DIAGRAM :

The use case model is essential for capturing the functional requirements of the system from the perspective of the end-user. The provided diagram exemplifies a use case model where a user interacts with the system by submitting a query, triggering document retrieval, response generation, and response delivery. It also highlights external components such as the document Store and LLM, which contribute to fulfilling the request.

CLASS DIAGRAM:

The class model represents the static structure of the system, defining the objects, classes, attributes, and relationships between various entities. It is crucial for understanding the data structure and how different components interact within the system. In the given use case, the class model would include representations of the User, Document Store, LLM, Query Processor, and Response Generator, showing their attributes and relationships.

SEQUENCE DIAGRAM:

The sequence **model** represents the interaction between different system components over time, showing how objects communicate through a series of messages. It provides a clear step-by-step flow of operations, illustrating the order of execution and dependencies between components. This model is useful for understanding real-time processing and request-response interactions.

ACTIVITY DIAGRAM:

The activity model focuses on the workflow and overall process rather than individual object interactions. It represents the sequence of actions performed to achieve a specific goal, using control flows to define decision points and parallel tasks. This model helps in visualizing the logical flow of operations, making it useful for understanding system behavior, automation, and process optimization.

5.1 USECASE DIAGRAM

This use case diagram outlines the interaction between a User and a PDF & Text File Chatbot System. The process starts with the user uploading files, which triggers the system to process the files. The system then splits the content into manageable chunks and stores them in the Vector Store for efficient retrieval. When the user submits a query, the system retrieves relevant documents from the Document Store and forwards the data to the LLM (Large Language Model) for generating a suitable response. The generated response is then returned to the user. Each key function uploading files, processing data, storing chunks, and generating responses is designed to ensure smooth document handling, fast data retrieval, and accurate query resolution.

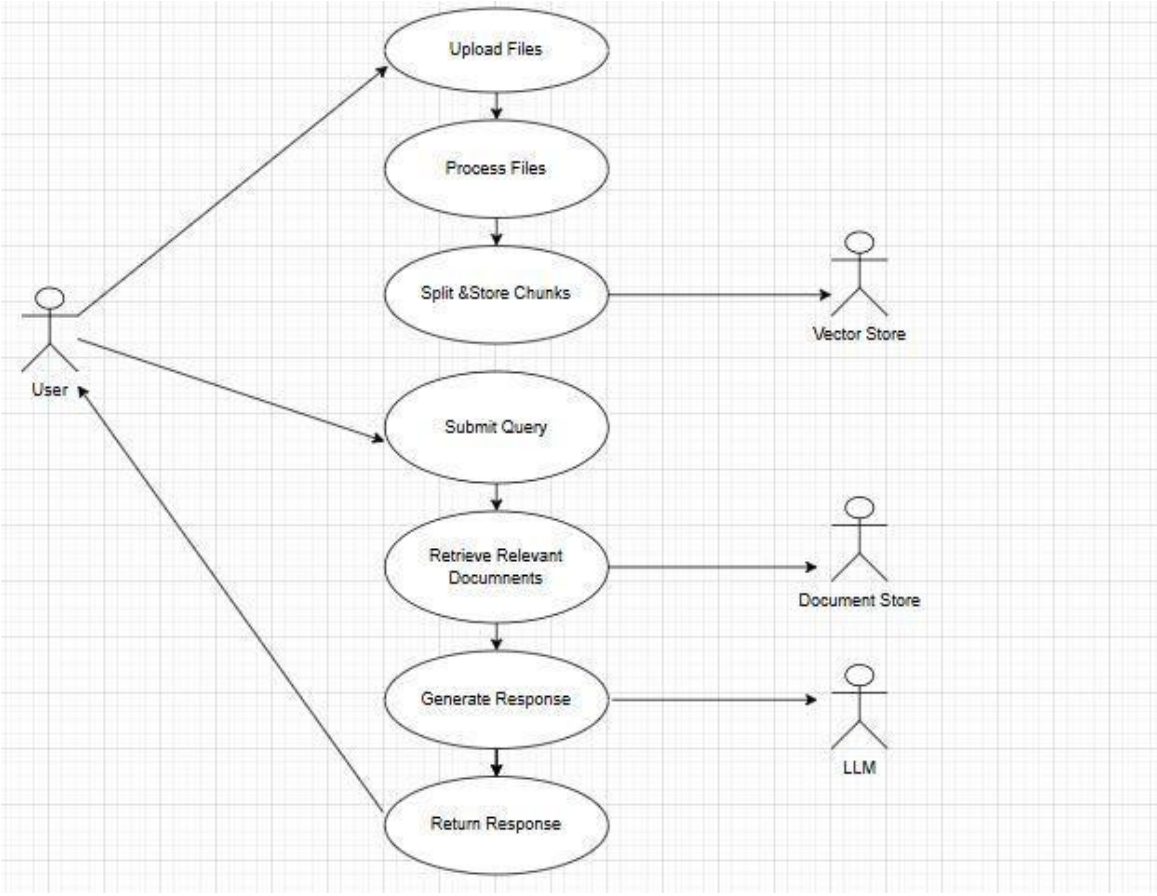

Fig No:5.1 Usecase Diagram

5.2 CLASS DIAGRAM

The class diagram represents a PDF & Text File Chatbot System designed for efficient document management and query response. The **User** interacts with the system by uploading PDF/text files, submitting queries, and receiving responses. The core system, PDFTextFileChatbotSystem, processes files, stores documents in the DocumentStore, and saves data chunks in the VectorStore for faster retrieval. When a query is submitted, the system fetches relevant information from the VectorStore and forwards it to the LLM (Large Language Model) for generating a response. The DocumentStore handles complete document storage, while the VectorStore manages smaller data chunks for improved search performance. The LLM generates accurate answers based on the retrieved data. The chatbot system integrates these components to ensure smooth information flow.

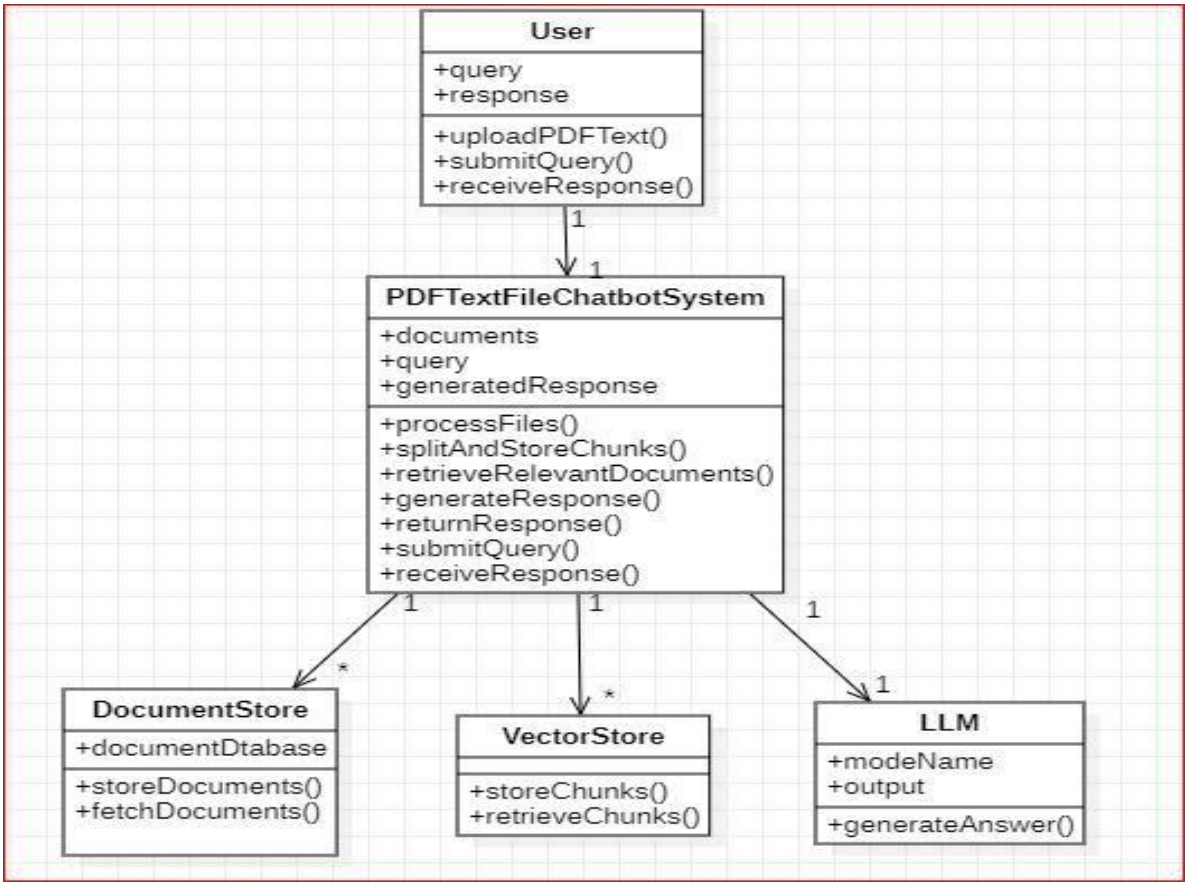

Fig No:5.2 Class Diagram

5.3 SEQUENCE DIAGRAM

This sequence diagram outlines the workflow of a PDF & Text File Chatbot System. The process begins when the user uploads a PDF or text file. The chatbot system processes the uploaded file and stores its content in the Document Store. After successful storage, an acknowledgment is sent back to the user. When the user submits a query, the chatbot system retrieves relevant documents from the Vector Store, which likely contains indexed and structured information for efficient searching. The system fetches the required data and returns it to the chatbot system. Next, the chatbot system generates a response by interacting with the LLM Model (Large Language Model). The generated response is then returned to the chatbot system, which ultimately displays the response to the user. This structured flow ensures efficient document management, query processing, and accurate text generation.

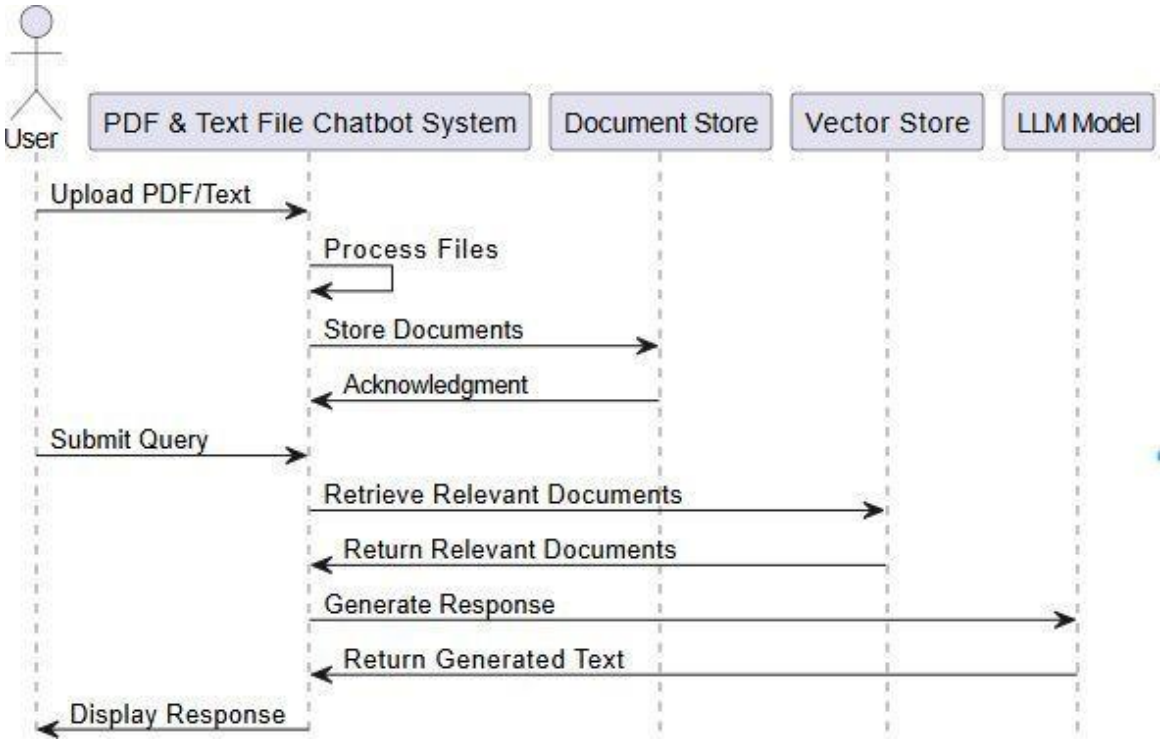

Fig No:5.3 Sequence Diagram

5.4 ACTIVITY DIAGRAM

This activity diagram illustrates the workflow of a PDF & Text File Chatbot System. The process begins when the User uploads a PDF or text file. The system processes the file and stores its contents in the Document Store. After successful storage, the system sends an acknowledgment back to the user. When the user submits a query, the chatbot system retrieves relevant documents from the Vector Store, which holds indexed data for faster access. The retrieved documents are then passed to the LLM Model, which generates a suitable response. The generated text is returned to the system, which then displays the response to the user.

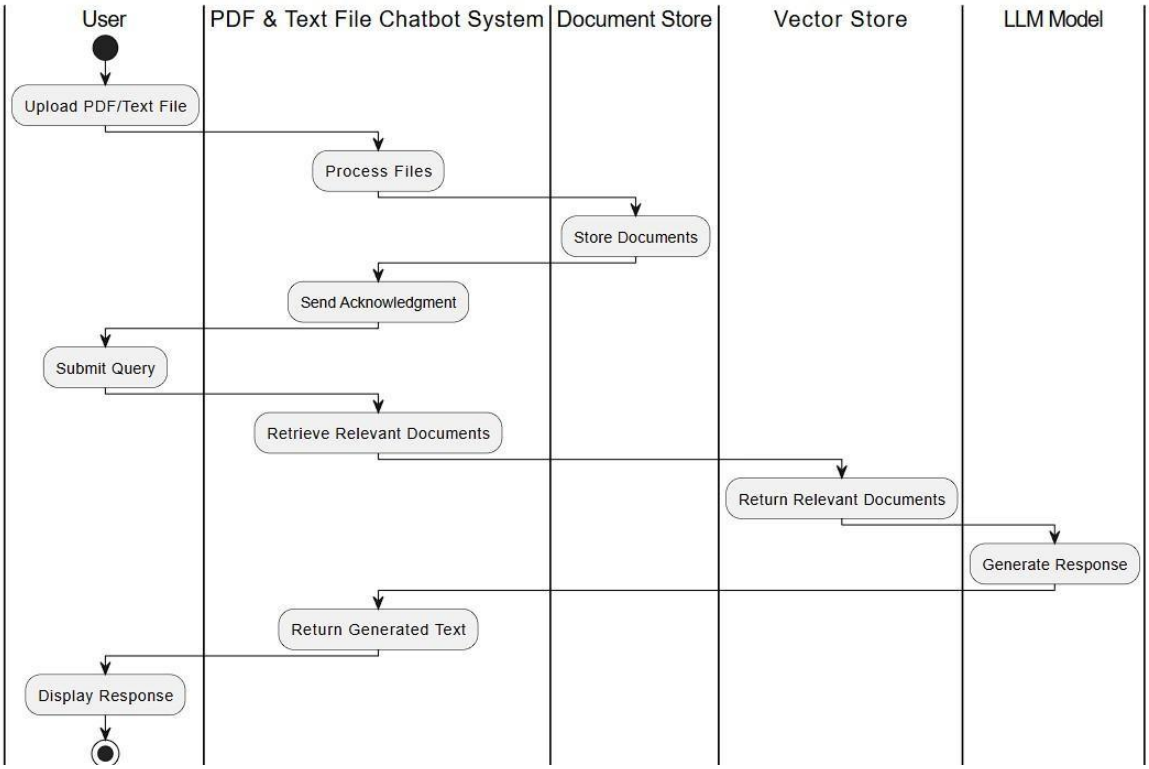

Fig No:5.4 Activity Diagram

6. IMPLEMENTATION

6.1 System Initialization:- The system initialization process sets up the necessary components for efficient document retrieval and query processing. It begins by loading a Sentence Transformer model (all-MiniLM-L6-v2), which converts text into high-dimensional vector embeddings for semantic search. Next, it initializes a FAISS (Facebook AI Similarity Search) index, specifically using an IndexFlatL2 structure with a 384-dimensional space, matching the embedding size of the transformer model. Additionally, an in-memory document store is created to store the actual document content, while a mapping between vector indices and document IDs is maintained. This setup enables the system to quickly process and retrieve relevant information by comparing query embeddings with stored document vectors, forming the foundation for fast and accurate response generation.

```
def initialize_system():
    global model, vector_store

    # Use HuggingFace Sentence Transformer
    model = SentenceTransformer('all-MiniLM-L6-v2')

    # Create FAISS Index
    index = faiss.IndexFlatL2(384)  # Dimension of embeddings
    vector_store = FAISS(
        embedding_function=model.encode,
        index=index,
        docstore=InMemoryDocstore(),
        index_to_docstore_id={}
    )

initialize_systeem()
```

6.2 File Upload and Processing:- User uploads PDF or text files, the system identifies the file type and uses the appropriate loader—PyMuPDFLoader for PDFs and TextLoader for plain text files—to extract content. The extracted text is then split into smaller chunks using the RecursiveCharacterTextSplitter**,** ensuring efficient storage and retrieval by maintaining contextual integrity. Each chunk is then converted into vector embeddings using the Sentence Transformer model and stored in the FAISS index for similarity-based search. This structured approach enables efficient retrieval of relevant document segments, enhancing the accuracy of the responses generated from the stored knowledge.

```
def upload_files(files):
    try:
        if not files:
            return "No files uploaded."
        for file in files:
            if file.name.endswith(".pdf"):
                loader = PyMuPDFLoader(file.name)
            elif file.name.endswith(".txt"):
                loader = TextLoader(file.name)
            else:
                continue  # Skip unsupported file types
            docs = loader.load()
            if not docs:
                return f"No text extracted from {file.name}."
            # Split into chunks
            splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
            chunks = splitter.split_documents(docs)
            if not chunks:
                return f"Failed to split {file.name} into chunks."
            # Add to vector store
            vector_store.add_documents(documents=chunks)
```

6.3 Query Processing and Answer Generation:- User submits a query, the system converts it into a numerical representation using the same embedding model applied to the stored document chunks. It then searches for the most relevant document by comparing the query embedding with the precomputed embeddings in the index. The system retrieves the closest matching document along with a similarity score, which indicates how well the document aligns with the query. If the similarity score meets a predefined threshold, the corresponding document content is extracted and returned as the response. This approach ensures that the answer is contextually relevant and based on the most suitable information available in the stored documents.

```
# Function to retrieve answers based on user queries
def generate_answer(question):
    try:
        # Retrieve the most relevant document with similarity scores
        docs = vector_store.similarity_search_with_score(query=question, k=1)

        if not docs:  # No documents found
            return "No relevant information found."

        # Extract best matching document and its similarity score
        doc, score = docs[0]

        # FAISS returns distances, where lower values indicate better matches
        if score > 1.2:  # If the match isn’t close enough, it’s not relevant
            return "No relevant information found."

        # Return the best-matching document content
        return doc.page_content

    except Exception as e:
        return f"Error generating response: {str(e)}"
```

6.4 Gradio Web Interface :- The Gradio Web Interface provides an interactive and user-friendly platform for uploading documents and querying stored information. It consists of two main sections: a file upload panel, where users can add PDF or text files for processing, and a chat interface, where users can enter questions and receive responses. The system allows multiple file uploads, processes them in the background, and displays the status of the operation. Once the documents are stored, users can type queries into the input box, and the system retrieves the most relevant information, displaying the response in real time. With intuitive buttons for file processing and query submission, the interface ensures a seamless experience for users to interact with the document-based AI system effortlessly.

```
import gradio as gr
# Create Gradio interface
with gr.Blocks() as app:
   gr.Markdown("# PDF & Text File Chatbot with AI Embeddings")

   # Sidebar for Uploading Files
   with gr.Row():
      with gr.Column(scale=1):
         gr.Markdown("## Sources")
         file_input = gr.File(label="Upload PDFs or Text Files", file_types=[".pdf", ".txt"],
file_count="multiple")
         upload_button = gr.Button("Process Files")
         upload_output = gr.Textbox(label="Status")

      # Main Chat Interface
      with gr.Column(scale=3):
         gr.Markdown("## Chat")
         question_input = gr.Textbox(label="Ask a Question")
         answer_output = gr.Textbox(label="Answer")
         submit_button = gr.Button("Get Answer")

   # Link buttons to functions
   upload_button.click(upload_files, inputs=[file_input], outputs=[upload_output])
   submit_button.click(generate_answer,inputs=[question_input],
outputs=[answer_output])
 app.launch()
```

6.5 Downloading Audio from YouTube :- The process of downloading audio from YouTube begins with extracting the best available audio stream from the given video URL. The system utilizes yt-dlp, an advanced YouTube downloader, to fetch the highest-quality audio while ensuring efficiency. The extracted audio is then converted into MP3 format using FFmpeg, a powerful multimedia framework. To optimize speed and storage, the audio quality is set to 32 kbps, which balances clarity and file size. The resulting MP3 file is saved with a predefined name and returned for further processing. This streamlined approach ensures that only the necessary audio is retrieved, making transcription faster and more efficient.

```
import yt_dlp as youtube_dl
# Function to download audio from a YouTube video (Limited to 2 minutes for speed)
def download_audio(url, output_path="audio.mp3", duration=120):
    ydl_opts = {
        'format': 'bestaudio/best',
        'outtmpl': output_path.rstrip('.mp3'),  # Ensure correct filename
        'postprocessors': [{
            'key': 'FFmpegExtractAudio',
            'preferredcodec': 'mp3',
            'preferredquality': '32',  # Lower quality for faster processing
        }],
        'postprocessor_args': ['-t', str(duration)],  # Limit duration to first 2 minutes
        'quiet': True
    }
    with youtube_dl.YoutubeDL(ydl_opts) as ydl:
        ydl.download([url])

    return output_path
```

6.6 Transcribing Audio Using Whisper and Convert Into Pdf:-

Transcribing audio using Whisper involves converting speech from the downloaded MP3 file into text. The system loads the Whisper "tiny" model, which is optimized for speed and efficiency. It then processes the audio to generate a transcript without word timestamps, keeping the output clean and readable. The transcribed text is then formatted and saved as a PDF using the FPDF library. This approach ensures a fast and accurate transcription process, making it easy to store and share the generated text.

```
import whisper
from fpdf import FPDF
import os

# Function to transcribe audio and save as a PDF
def transcribe_audio(audio_file, output_pdf_file="transcription.pdf"):
    model = whisper.load_model("tiny")  # Use the fastest Whisper model
    result = model.transcribe(audio_file, fp16=False, word_timestamps=False)

    # Save transcription to a PDF
    pdf = FPDF()
    pdf.set_auto_page_break(auto=True, margin=15)
    pdf.add_page()
    pdf.set_font("Arial", size=12)
    pdf.multi_cell(0, 10, result["text"])
    pdf.output(output_pdf_file)

    return output_pdf_file
```

7. DEPLOYMENT

This provides a step-by-step process to deploy and manage the Retrieval-Augmented Generation (RAG) system in Google Colab. The system allows users to upload PDFs and text files, process them into vector embeddings using FAISS and Sentence Transformers, and retrieve relevant information via a chatbot built with Gradio.

7.1 Deployment Environment

The RAG system is deployed in **Google Colab**, a cloud-based Jupyter Notebook environment that provides access to computational resources without requiring local installation. Google Colab allows users to execute Python-based AI applications using cloud-hosted CPUs, GPUs, and TPUs. Since it runs entirely in a web browser, no additional hardware setup is needed, making it a convenient platform for deploying and testing AI models.

For hardware requirements, the system requires a minimum of 4GB RAM for handling small datasets and lightweight document processing. However, for processing multiple PDFs , at least 8GB RAM is recommended, which is available in **Google Colab Pro**. The platform provides a cloud-based CPU by default, but users can switch to a **GPU** (if available) by changing the runtime settings. Google Colab also offers around 30GB of temporary cloud storage, which is sufficient for storing and processing document embeddings.

Regarding software requirements, the system is implemented in Python 3.8+, which comes pre-installed in Google Colab. Several Python libraries are required, including Gradio for the user interface, FAISS for efficient vector search, LangChain for document processing, Sentence Transformer**s** for generating text embeddings, and PyMuPDF for handling PDFs.

7.2 Deployment Process

The first step is **installation,** where all necessary dependencies must be installed in the Colab environment since they are not pre-installed. This includes libraries such as Gradio (for the user interface), FAISS (for vector-based similarity search), LangChain (for document processing), Sentence Transformers (for text embedding), and PyMuPDF (for extracting text from PDFs).

The next step is **configuration**, where the system initializes its core components. A Sentence Transformer model is loaded to generate embeddings, while FAISS is set up as the vector store to store and retrieve processed document embeddings efficiently. The system also ensures that uploaded files (PDFs and text documents) are properly read, split into smaller text chunks, and embedded into FAISS for fast retrieval. Configuration also involves defining the chatbot interface using Gradio, which allows users to upload documents and ask questions dynamically.

The system moves to **execution**, where the Gradio application is launched in Colab. When the script is run, Colab provides a public URL where users can interact with the chatbot in real time. Users can upload documents, process them, and ask questions, with the system retrieving and returning the most relevant information based on the embeddings stored in FAISS.

The final stage is **testing**, which ensures that the system functions correctly and efficiently. Testing involves uploading different file formats (PDF, TXT) to verify that text extraction is working as expected. Users then input various queries to assess whether the retrieval system correctly finds the most relevant information. The system's performance is also evaluated by uploading multiple documents and running multiple queries to observe processing time and retrieval efficiency. If any issues arise, debugging can be done by analyzing error messages and adjusting configurations accordingly. Through this structured deployment process, the RAG System is successfully implemented in Google Colab, providing a secure and efficient AI-powered document retrieval and query system.

7.3 Deployment Considerations

When deploying the RAG System in Google Colab, several key factors must be considered to ensure smooth operation and efficiency. Security and data privacy are also important, as Colab operates in a cloud environment. Since the system is designed for private document retrieval, users should avoid storing sensitive data persistently and ensure that files are deleted after processing. Dependency management should be handled carefully by installing the required libraries at the start of each session, as Colab does not retain installed packages after disconnection. Additionally, runtime limitations in Colab, such as automatic session timeouts, may affect long-running tasks, so frequent checkpoints or saving vector stores externally can help retain progress. Lastly, scalability and user load should be considered while Colab is suitable for small-scale deployments, hosting on a dedicated server or local machine may be needed for continuous or large-scale use.

7.4 Post-Deployment Maintenance

After deploying the RAG System in Google Colab, continuous maintenance is essential to ensure long-term efficiency, accuracy, and security. Regular updates of dependencies such as Gradio, FAISS, LangChain, and Sentence Transformers are necessary to keep the system compatible with the latest features and improvements. Since Colab sessions reset after disconnection, reinstalling dependencies and reloading the model and vector store in every session is required. Performance monitoring should be conducted by testing retrieval accuracy and response time, optimizing chunk sizes and embeddings if needed. By implementing these maintenance strategies, the RAG System can operate smoothly, providing reliable and secure document-based AI retrieval over time.

8. TESTING

Testing is the process of evaluating a system or software to ensure it functions correctly, meets requirements, and is free from defects. It involves systematically checking various components, identifying issues, and verifying that the system performs as expected. Testing helps improve reliability, security, and efficiency before deploying the system for real-world use.

8.1 BLACK BOX TESTING

Black Box Testing is a software testing approach where the system is tested without knowing its internal workings. In the context of Retrieval-Augmented Generation (RAG), black box testing evaluates the model's performance purely based on its inputs and outputs, ensuring that it retrieves and generates accurate and meaningful responses. For your chatbot system, black box testing would involve:

Functional Testing – Testing:

- Whether PDFs and text files are correctly uploaded and processed.
- If user queries return relevant responses from the stored documents.
- The UI interactions in the Gradio interface.

Boundary Value Analysis – Providing edge case inputs, such as:

- Uploading an empty file.
- Submitting a very large document.
- Asking extremely short or long queries.

Usability Testing :

- Checking if the chatbot is user-friendly and displays clear error messages.

Integration Testing :

- Ensuring that different components (file processing, embedding model, FAISS indexing, and chatbot interface) work together seamlessly.

8.2 WHITE BOX TESTING

White Box Testing is a software testing method where the internal code, logic, and structure of the system are tested. Unlike Black Box Testing, which focuses only on inputs and outputs, White Box Testing examines the internal workings of the software. For your code, white box testing would focus on:

Code Coverage Analysis :

- Ensuring all functions (initialize_system(), upload_files(), generate_answer()) are executed at least once.

Unit Testing – Writing test cases for:

- initialize_system() to verify if the FAISS index and embedding model are initialized correctly.
- upload_files(files) to check if PDFs and text files are processed properly and added to the vector store.
- generate_answer(question) to validate correct retrieval and ranking of relevant documents.

Exception Handling Tests :

- Checking how the system handles empty inputs, unsupported file formats, or errors in embedding processing.

Performance Testing :

- Evaluating the efficiency of the FAISS index in retrieving relevant results.

8.3 TEST CASES

Test case1:Uploading Files

Output:All files loaded and processed successfully

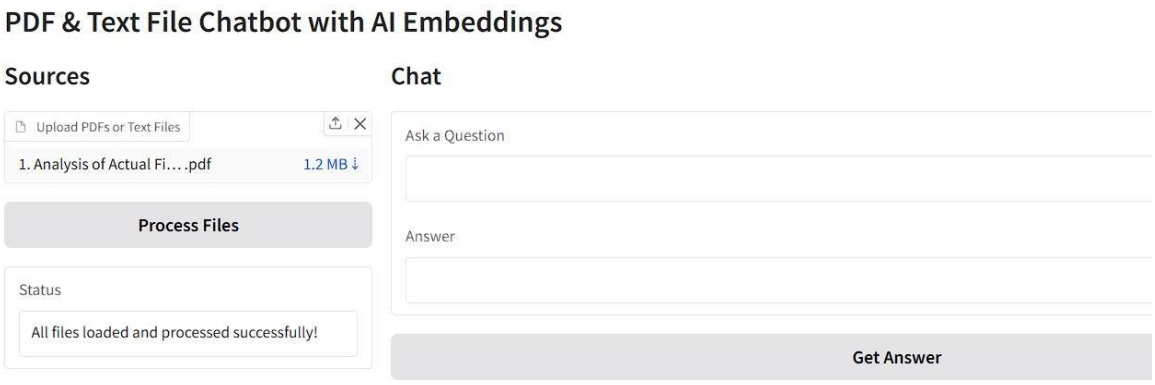

Fig No:8.3 Uploading Files

Test Case2:Files are not uploaded

Output:No files uploaded

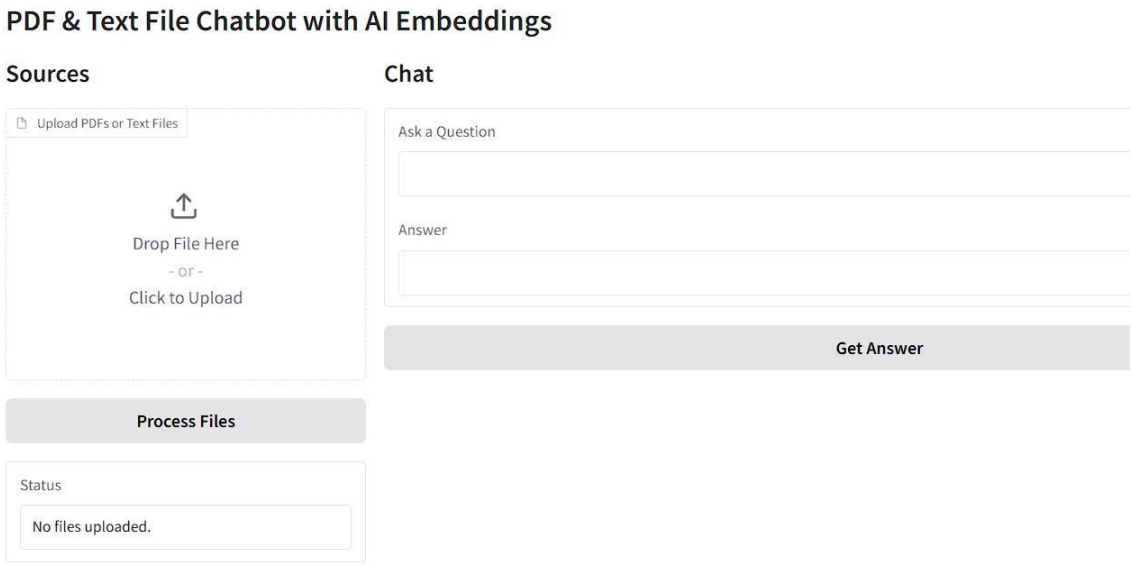

Fig No:8.4 Files are not uploaded

Test Case 3: Query a question that has relevant content in uploaded documents

Output: Returns a relevant text snippet from the document.

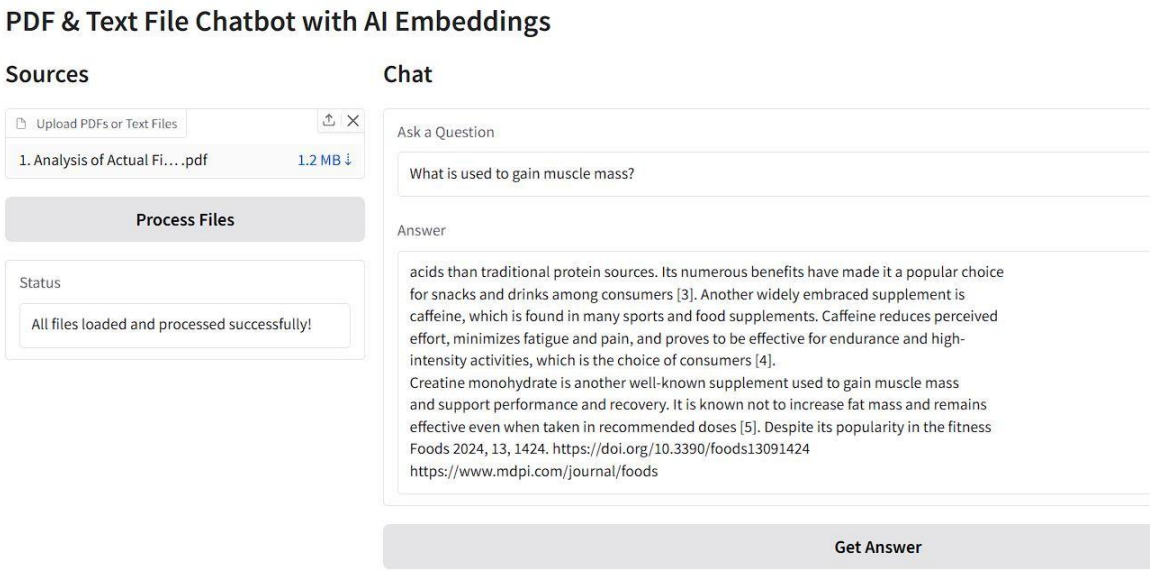

Fig No:7.5 Query a question that has relevant content in uploaded documents

Test case 4: Query a question that has no relevant content in uploaded documents

Output: No relevant information found

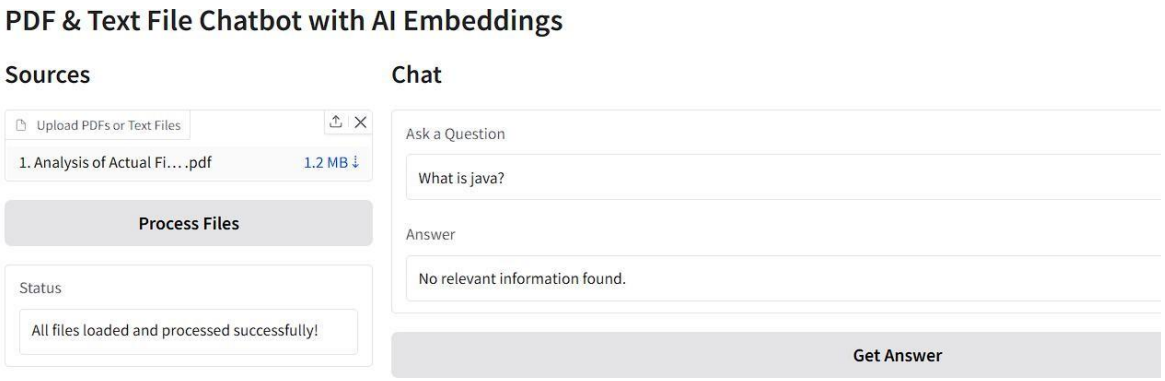

Fig No:7.6 Query a question that has no relevant content in uploaded documents

9. OUTPUT SCREENS

9.1 PDF & Text File Chatbot with AI Embeddings

The output screen consists of two main sections: Source Upload Panel and Chat Panel. The Source Upload Panel allows users to upload PDF or text files, process them, and store them for AI-based searching. The Chat Panel enables users to ask questions related to the uploaded documents, retrieving relevant answers based on semantic similarity. The system processes queries using FAISS vector search and returns the most relevant text from the stored files. If no relevant content is found, it informs the user accordingly. This setup provides a seamless way to interact with and extract information from documents.

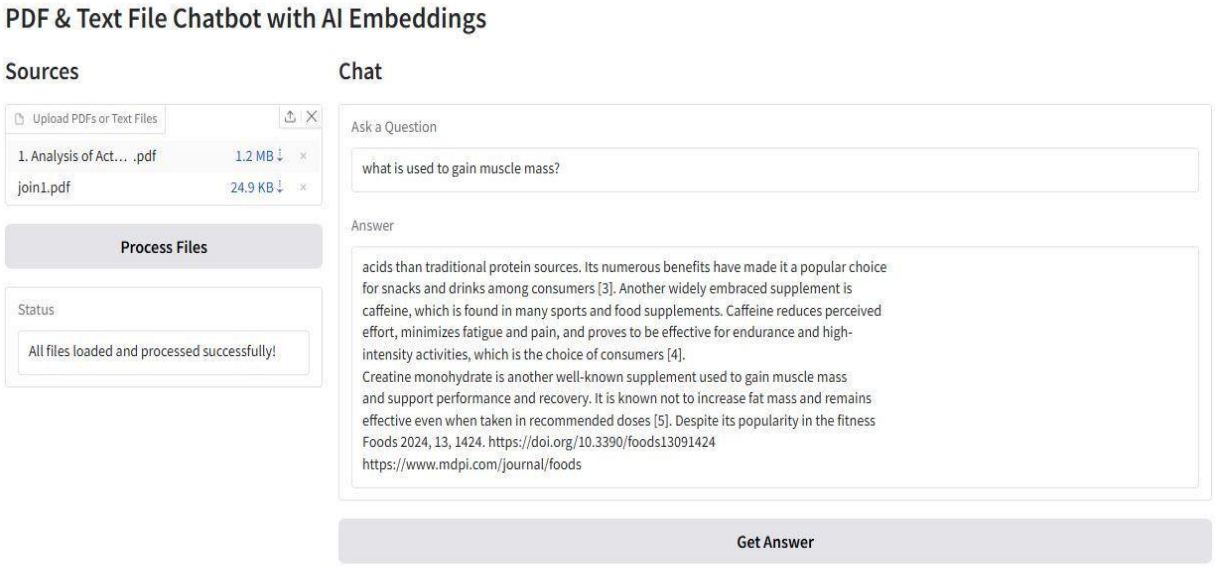

Fig No:9.1 PDF & Text File Chatbot with AI Embeddings

10. CONCLUSION

A Retrieval-Augmented Generation (RAG) system offers a powerful solution to the limitations of traditional AI models by ensuring that responses are generated exclusively from user-uploaded documents. This approach eliminates hallucinations and enhances data privacy by extracting and storing information locally. It is particularly valuable in fields such as enterprise knowledge management, legal analysis, healthcare, finance, and education, where domain-specific accuracy is essential. By preventing misinformation and enabling secure, controlled access to private data, the RAG system empowers context-aware decision-making, making it a crucial tool for handling sensitive and specialized information.

REFERENCES

[1] Binita Saha, " Retrieval-Augmented Generation With Inverted Question Matching for Enhanced QA Performance ", 2024

[2] Minsang Kim**,** Cheoneum Park, and Seungjun Baek ,QPaug: " Question and Passage Augmentation for Open-Domain Question Answering of LLM " , 2024

[3] Shamiha Binta Manir ,K.M.Sajjadul Islam2 ," LLM-Based Text Prediction and Question Answer Models for Aphasia Speech ", 27 August 2024

[4] OpenAI. (2023). *Whisper: Robust speech recognition model*. Retrieved from https://openai.com/research/whisper

[5] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[6] https://python.langchain.com/en/latest/modules/data_connection/document_transformers.html

GITHUB LINK

https://github.com/keerthanapogiri/Retrieval-Augmented-Generation