RETRIEVAL AUGMENTEDGENERATION

A PROJECT REPORT

Submitted by

S.KALAI SELVI	812821104029
C.KAVIYA	812821104032
N.MEHAR JABEEN	812821104040

in partial fulfillment for the award of the degree

*o*f

BACHELOR OF ENGINEERING

*i*n

COMPUTER SCIENCE AND ENGINEERING

MOOKAMBIGAI COLLEGE OF ENGINEERING,PUDUKKOTTAI

ANNA UNIVERSITY: CHENNAI 600 025

MAY 2025

ANNAUNIVERSITY: 600 025

BONAFIDE CERTIFICATE

Certified that report "**RETRIEVAL AUGMENTED GENERATION"** is the Bonafide work of "**MEHAR JABEEN, KALAISELVI, KAVIYA**" who carried out the project work under my supervision.

SIGNATURE	**SIGNATURE**
Dr.P.VALARMATHI, M.E., M.B.A., Ph.D.,	Mrs.S.Sigappi, M.E.,
HEAD OF THE DEPARTMENT,	**SUPERVISOR,**
Professor,	Assistant Professor,
Computer Science & Engg,	Computer Science & Engg,
Mookambigai College of Engg,	Mookambigai College of Engg,
Pudukkottai-622 502	Pudukkottai-622 502

Submitted for the University project review held on ____________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

Understanding complex PDF documents such as research papers, legal contracts, and technical manuals can be time-consuming and challenging, especially when users need to locate key information quickly. This offline AI-powered tool is developed to simplify this process by automatically extracting and organizing valuable content, including text, tables, figures, and code. Leveraging advanced language models and layout analysis techniques, the system transforms dense, unstructured documents into searchable and structured data that users can interact with. Through a simple and intuitive local web interface, users can upload PDF files and ask natural language questions about the content. The system provides clear, accurate answers with references to the original document, enabling faster comprehension and deeper insights. It supports a wide range of use cases, including academic research, legal document review, technical support, and training materials analysis. A core feature of this tool is its complete offline functionality. All processing is done locally, without requiring an internet connection, ensuring data confidentiality and security. This makes it particularly suitable for privacy-sensitive environments such as schools, universities, law firms, government institutions, and engineering teams. Users retain full control of their data, and no information is sent to external servers. In summary, the tool offers an effective solution for navigating and understanding complex documents without compromising privacy. It combines powerful AI with local accessibility to deliver fast, accurate, and user-friendly document analysis. Its ability to run entirely offline, while maintaining high performance, makes it a practical choice for professionals and organizations seeking secure and efficient ways to work with intricate PDFs.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In today's digital world, PDF documents have become a standard format for sharing information across academic, legal, technical, and business domains. However, understanding and extracting meaningful data from complex PDFs such as research papers, contracts, technical manuals, or financial reportsremains a challenging task. These documents often contain dense text, structured tables, code snippets, and specialized formatting that are not easily searchable or interpretable by non-expert users. Traditional PDF readers offer limited functionality for advanced analysis, often requiring manual effort and significant time.

To address these challenges, we introduce an offline AI-powered tool designed to simplify PDF document understanding. The tool utilizes artificial intelligence to extract key content such as text, tables, and embedded code, converting it into searchable, structured data. It offers a user-friendly local web interface where users can upload documents and ask questions in natural language. In response, the system provides relevant, accurate answers along with references to the source content, significantly reducing the time required to locate and comprehend information.

A major advantage of this tool is its ability to operate entirely offline, ensuring data privacy and security. Unlike cloud-based alternatives, this solution does not require an internet connection and does not send any data to external servers. This feature makes it particularly valuable for use in environments where confidentiality is critical, such as law firms, educational institutions, research labs, and corporate offices.

CHAPTER 2

SYSTEM ANALYSIS

2.1. EXISTING SYSTEM

Currently, most PDF document analysis tools either rely on basic keyword search or require constant internet connectivity to function, often depending on cloud-based services. These systems are limited in handling complex documents such as research papers, legal contracts, or technical manuals, especially when the information is deeply embedded in tables, code blocks, or nested paragraphs. Moreover, many existing solutions do not prioritize data privacy, as documents are processed on remote servers. This raises concerns in environments where confidentiality is crucial, such as legal, academic, and government institutions. Additionally, non-technical users often struggle to extract meaningful insights without specialized knowledge, making the process slow and inefficient. There is a clear need for an intelligent, user-friendly, and offline system that ensures secure, accurate, and accessible PDF content analysis.

1. **Humata.ai**

- Upload PDFs and ask questions in natural language
- Auto-summarizes, explains, and cites answers
- Best for: Research papers, reports, legal docs

2. **ChatPDF**

- Simple UI: Drop in a PDF and chat with it
- Free tier available, supports multiple languages
- Best for: Quick Q&A over casual documents

3. **AskYourPDF**

- PDF chatbot powered by OpenAI or custom LLMs
- Browser extension and file history support
- Best for: Repeated queries over large files

2.2.1 Drawbacks

- Requires internet to function
- No data privacy – files uploaded to cloud
- Free versions have limited features
- Not accurate with complex PDFs (tables, code, etc.)
- No offline/local deployment support
- Cannot customize for specific domains or needs

2.2.3 Limitation of existing system

- **Dependence on Cloud APIs**: Most RAG systems rely on OpenAI or other hosted LLM APIs, unsuitable for secure/offline environments.
- **Limited PDF Parsing**: Many systems extract plain text without preserving tables, code snippets, or citation context.

2.3 PROPOSED SYSTEM

The proposed system is a fully offline, AI-powered PDF analysis tool that operates using a Retrieval-Augmented Generation (RAG) framework. It eliminates internet dependency by using FAISS for vector storage and SentenceTransformers for embeddings. For PDF parsing, the system leverages powerful libraries like PyMuPDF and pdfplumber to accurately extract structured data from complex layouts, including tables, diagrams, and embedded code.

To maintain semantic context, documents are divided into overlapping chunks with metadata tags that indicate content type (e.g., citation, code, table). This improves the relevance of responses during user queries. The architecture is modular—each component such as the loader, chunker, embedder, retriever, and UI can be individually customized or extended. A simple Streamlit-based interface allows users to upload documents and interact with them through conversational input.

- **Offline RAG Framework**: Uses FAISS and SentenceTransformers to eliminate internet dependency.
- **Enhanced PDF Parsing**: Utilizes PyMuPDF and pdfplumber to accurately extract complex structures like tables, diagrams, and code.
- **Semantic Chunking**: Splits documents into overlapping chunks to preserve context, with tagging for content type (citation, table, etc.).
- **Modular Architecture**: Each component (PDF loader, chunker, embedder, retriever, UI) is independently extensible.
- **User Interface**: Simple Streamlit front-end allowing document upload and conversational interaction.
- **Error Handling & Validation**: Captures edge cases like invalid PDFs, empty documents, or irrelevant queries.

2.3.1 ADVANTAGES OF PROPOSED SYSTEM

- **Smart Retrieval** – Uses semantic search for accurate, context-aware results.
- **Handles Complex PDFs** – Supports academic, legal, and technical documents.
- **Contextual Answers** – Generates answers grounded in the document content.
- **Offline Functionality** – Works without internet, ensuring data privacy.
- **Modular Design** – Easy to maintain, upgrade, and debug.
- **Scalable** – Easily extendable to other models, formats, or databases.
- **User-Friendly** – Simplifies access to complex information via natural language.
- **Secure** – No external API or cloud use—ideal for sensitive content.

CHAPTER 3

SYSTEM SPECIFICATION

3.1 HARDWARE CONFIGURATION

- **PROCESSOR** : Intel Core i5 or higher / AMD Ryzen 5 or higher
- **RAM** :8GB(16 GB recommended for large document processing)
- **STORAGE** : At least 5 GB of free disk space
- **GPU (optional)** :NVIDIA GPU with CUDA support (for faster embedding and model inference)

3.2 SOFTWARE CONFIGURATION

- **OPERATING SYSTEM** :Windows 10/11, Ubuntu 20.04+, or macOS 11+
- **PYTHON VERSION** : Python 3.9 or later
- **LIBRARIES&DEPENDENCIES** : PyMuPDF or pdfminer.six (for PDF parsing)

sentence-transformers (for embeddings)

faiss-cpu or faiss-gpu (for vector indexing)

langchain, transformers, torch (for retrieval and answer generation)

CHAPTER 4

SOFTWAR DESCRIPTION

4.1 FRONT END

Streamlit

The frontend of the PDF GENIE system is developed using Python along with the Streamlit framework. Streamlit provides a fast and simple way to build web applications with minimal code. It enables users to interact with the system directly through their web browser without requiring any frontend development experience. Since it runs locally, it ensures user privacy and smooth performance even without internet access.

4.1.1 Features

- PDF Upload Interface
- The frontend includes a user-friendly upload interface where users can select and upload PDF files from their local system for analysis.
- Chatbox for User Queries

  A simple text input field is provided where users can enter their questions related to the uploaded document in natural language.
- Answer Display Area

  The system displays AI-generated answers along with context or citations from the document, making it easier for users to understand the source of the response.
- Real-Time Feedback

  Users receive instant feedback on their queries, and the system updates responses dynamically, enhancing user engagement.
- Error Notification System

  If the user uploads an invalid file or asks an unsupported question, the frontend handles it gracefully by showing clear and informative messages.

4.2 BACK END

Python

The backend of PDF GENIE is built using **Python** and follows a modular, offline-capable architecture. It integrates advanced natural language processing, semantic search, and document parsing components. The backend handles the entire logic of reading the PDF, converting it into searchable chunks, retrieving relevant information, and generating meaningful answers. It works completely offline, ensuring data security and high performance in local environments.

4.2.1 Features

- **PDF Parsing Module**

  The system uses libraries like **PyMuPDF** or **pdfplumber** to extract structured content such as text, tables, and code blocks from complex PDF documents.

- **Semantic Chunking**
  Extracted content is broken into meaningful, overlapping chunks to retain context and improve answer relevance. Each chunk is tagged based on content type like text, table, or citation.

- **Embedding Generation**
  The backend employs **sentence-transformers** to convert document chunks into dense vector representations for semantic similarity comparison.

- **Vector Search using FAISS**
  FAISS is used to index and retrieve the most relevant chunks based on user queries, allowing quick and accurate information retrieval.

- **Answer Generation (RAG Framework)**

  By using **Langchain** with **Transformers** and **Torch**, the backend formulates human-like answers from retrieved content, forming an offline RAG (Retrieval-Augmented Generation) pipeline.

- **Error Handling & Validation**
  The system includes checks for invalid PDFs, empty documents, or unrelated queries, and provides informative error messages without crashing.

CHAPTER 5

PROJECT DESCRIPTION

5.1 OVERVIEW OF THE PROJECT

PDF GENIE is an AI-powered offline tool designed to help users understand complex PDF documents such as research papers, contracts, manuals, and technical reports. The system allows users to upload PDF files and interact with them using natural language queries. It extracts and semantically processes content—including text, tables, and code—from the document and provides accurate, context-aware answers along with references. Built using Python, the project integrates a local RAG (Retrieval-Augmented Generation) framework with tools like PyMuPDF, FAISS, and Langchain. Its simple Streamlit-based frontend offers a user-friendly interface for document upload and conversation, while the backend handles intelligent parsing, retrieval, and response generation—all running locally without internet dependency. This makes PDF GENIE ideal for use in educational institutions, law firms, and organizations that require secure, private document handling.

5.2 MODULES DESCRIPTION

- pdf_loader.py
- chunker.py
- embedder.py
- vector_store.py
- rag_pipleine.py
- interface.py
- helpers.py
- app.py
- requirements.txt
- README.md

5.2.1 pdf_loader.py

The **pdf_loader.py** module is designed to handle the loading and extraction of textual content from PDF documents. It reads the PDF files page by page, processes the embedded text, and converts it into a raw text format that can be easily manipulated in the later stages of the project. This module ensures that the complex structure of PDFs is simplified into clean, accessible text, which forms the foundation for subsequent analysis and processing.

5.2.2 chunker.py

The **chunker.py** module takes the raw text extracted from PDFs and breaks it down into smaller, manageable chunks. This segmentation is crucial because processing an entire document as one long string can be inefficient and less effective for tasks like embedding or search. The module employs strategies such as splitting by sentences, paragraphs, or fixed token lengths, and may use overlapping chunks to maintain context between segments, thereby improving downstream processing quality.

5.2.3 embedder.py

In the **embedder.py** module, each text chunk is transformed into a numerical vector representation known as an embedding. This is typically achieved using state-of-the-art pre-trained language models that capture semantic meaning beyond simple keywords. These embeddings enable the system to compare and search text based on meaning rather than exact word matches, making them essential for intelligent retrieval and understanding within the project.

5.2.4 vector_store.py

The **vector_store.py** module is responsible for managing the storage and retrieval of text embeddings. It integrates with vector databases or indexing libraries (like FAISS or Pinecone) to efficiently index and query large sets of embeddings. This allows the system to quickly find the most semantically similar text chunks in response to user queries, which is a critical component for the retrieval-based parts of the application.

5.2.5 rag_pipeline.py

The **rag_pipeline.py** module implements the Retrieval-Augmented Generation workflow by combining the strengths of retrieval and generative models. When a user submits a query, this module retrieves relevant text chunks from the vector store and feeds them as context into a generative language model (such as GPT). This results in more accurate and context-aware responses, as the model uses both the retrieved knowledge and its language generation capabilities.

5.2.6 interface.py

The **interface.py** module manages how users interact with the system, whether through a graphical user interface (GUI), command-line interface (CLI), or API endpoints. It handles input validation, manages user requests, and formats output responses in a user-friendly way. This module acts as the bridge between the backend processing and the end-user experience.

5.2.7 helpers.py

The **helpers.py** module contains a collection of utility functions that support various parts of the project. These might include text cleaning functions, error handling, logging utilities, file input/output helpers, and other reusable code snippets that keep the main modules clean and maintainable by avoiding redundancy.

5.2.8 app.py

The **app.py** module is the main entry point for the application. It orchestrates the integration of all other modules, sets up the user interface, and manages the overall workflow. Whether launching a Streamlit app or a Flask API, this module ensures that user requests flow correctly through loading, chunking, embedding, retrieval, and response generation.

5.2.9 requirements.txt

The **requirements.txt** file lists all Python dependencies necessary for the project to run. It specifies exact versions of libraries to ensure that the environment setup is reproducible and consistent, preventing compatibility issues across different machines or deployments.

5.2.10 README.md

The **README.md** file serves as the main documentation for the project. It provides an overview of what the project does, how to install and set it up, how to use it, and explains the role of the different modules. It helps users and developers understand the purpose and workflow of the project at a glance.

5.3 ALGORITHMS USED

5.3.1 RETRIEVAL-AUGMENTED GENERATION (RAG)

The Retrieval-Augmented Generation (RAG) algorithm starts by receiving a user query, which it first converts into a vector embedding using a pre-trained language model. Simultaneously, the system has a collection of documents or text chunks that are also converted into embeddings and stored in a vector database. The algorithm then searches this database to retrieve the most relevant chunks whose embeddings are closest to the query embedding, based on similarity measures like cosine similarity. These retrieved chunks are combined with the original query and fed into a generative language model, which uses this enriched context to produce a precise and context-aware response. Finally, the generated answer is returned to the user. This stepwise process allows the model to ground its output in actual document content, improving accuracy and relevance over purely generative approaches.

5.3.2 ALGORITHM STEPS FOR PDF GENIE

Step-1: Input

- User uploads a PDF document or inputs a query.

Step-2: PDF Text Extraction

- Extract raw text from the PDF file (pdf_loader.py).

Step-3: Text Chunking

- Split the extracted text into smaller chunks (chunker.py).

Step-4: Embedding Creation

- Convert text chunks into vector embeddings (embedder.py).

Step-5: Vector Storage and Indexing

- Store embeddings in a vector database for similarity search (vector_store.py).

Step-6: Query Embedding

- Convert the user query into an embedding vector.

Step-7: Similarity Search & Retrieval

- Retrieve most relevant chunks from vector store based on similarity to query.

Step-8: Answer Generation

- Feed retrieved chunks + query into generative model to produce answer (rag_pipeline.py).

Step-9: Output

- Display the generated answer to the user (interface.py).

5.4 DATA FLOW DIAGRAM (DFD)

The Data Flow Diagram (DFD) for the PDF Genie project illustrates how data moves through the system, starting from user input to the final generated output. Initially, the user uploads a PDF file or submits a query through the interface. The PDF loader module extracts raw text from the uploaded document, which is then passed to the chunker module to break the text into smaller, manageable pieces. These chunks are converted into vector embeddings by the embedder module and stored in a vector database for efficient retrieval. When the user submits a query, it is embedded into the same vector space, allowing the vector store to retrieve the most relevant chunks based on similarity.

These retrieved chunks, along with the query, are fed into the generative language model within the RAG pipeline to produce a contextually accurate response. Finally, this response is sent back through the interface to the user. The DFD helps visualize the flow of data between modules, making it easier to understand the system's architecture and interaction points.

5.4.1 DATAFLOW DIAGRAM

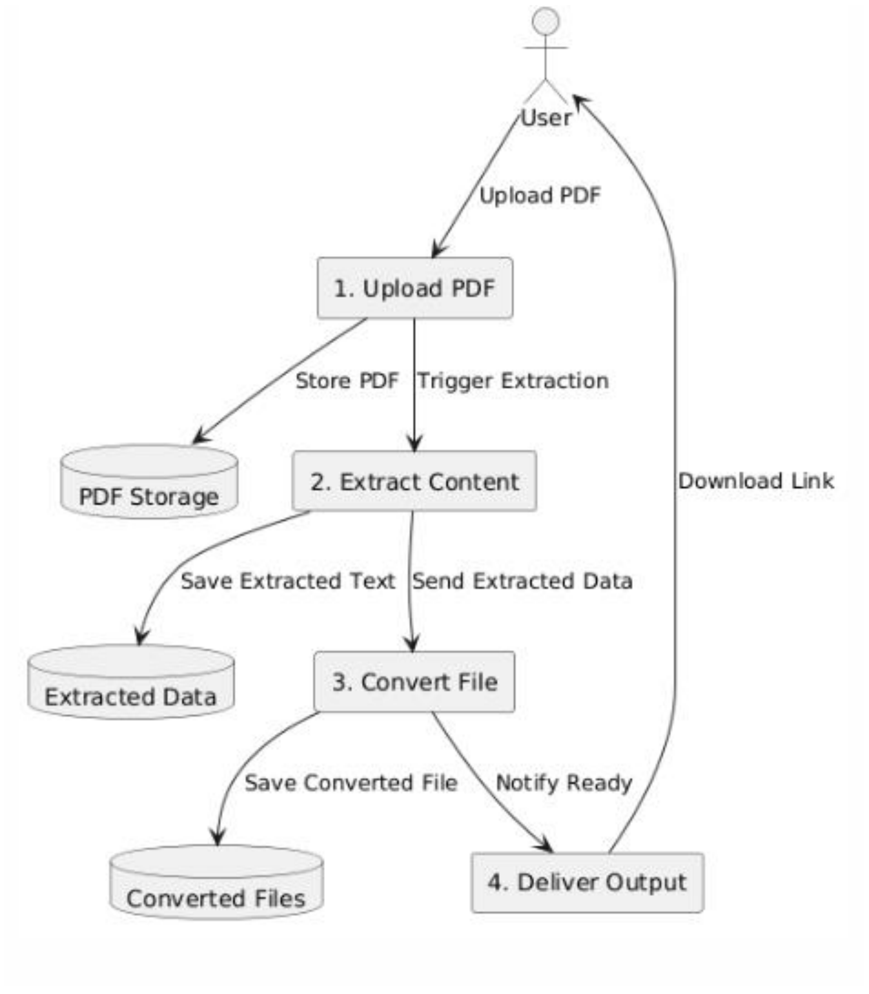


Fig.No.1.1

LEVEL 0

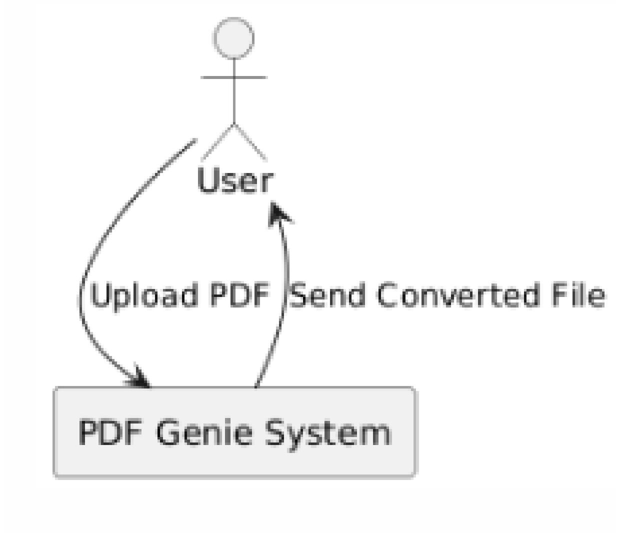


Fig.No.1.2

LEVEL 1

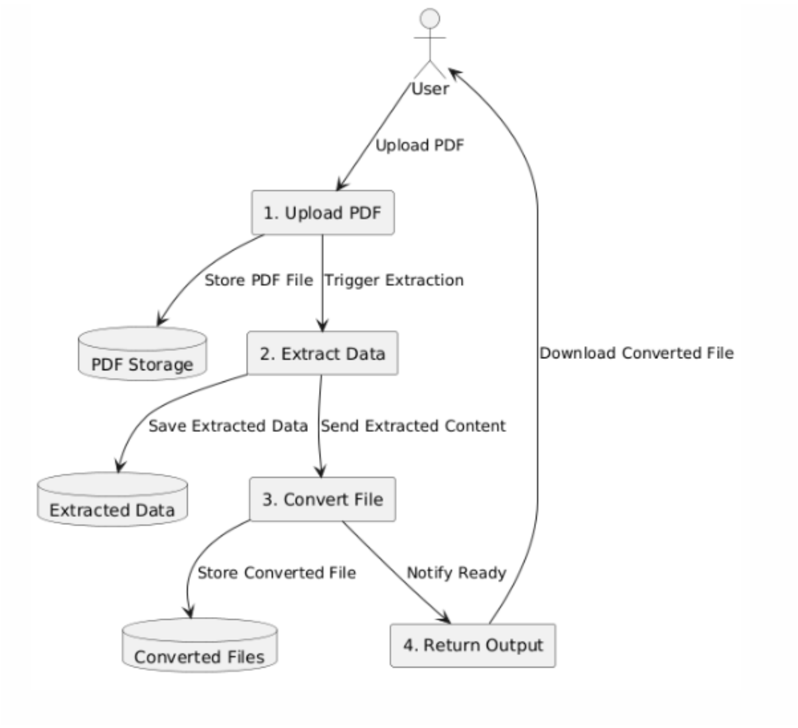


Fig.No.1.3

LEVEL 2

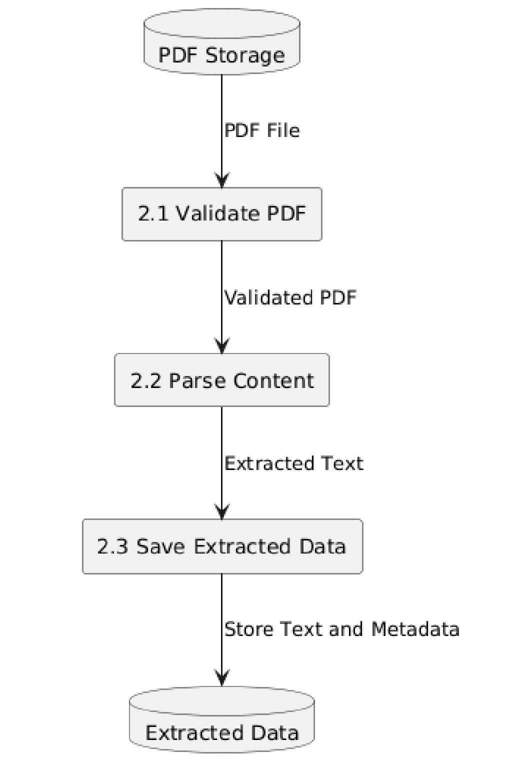
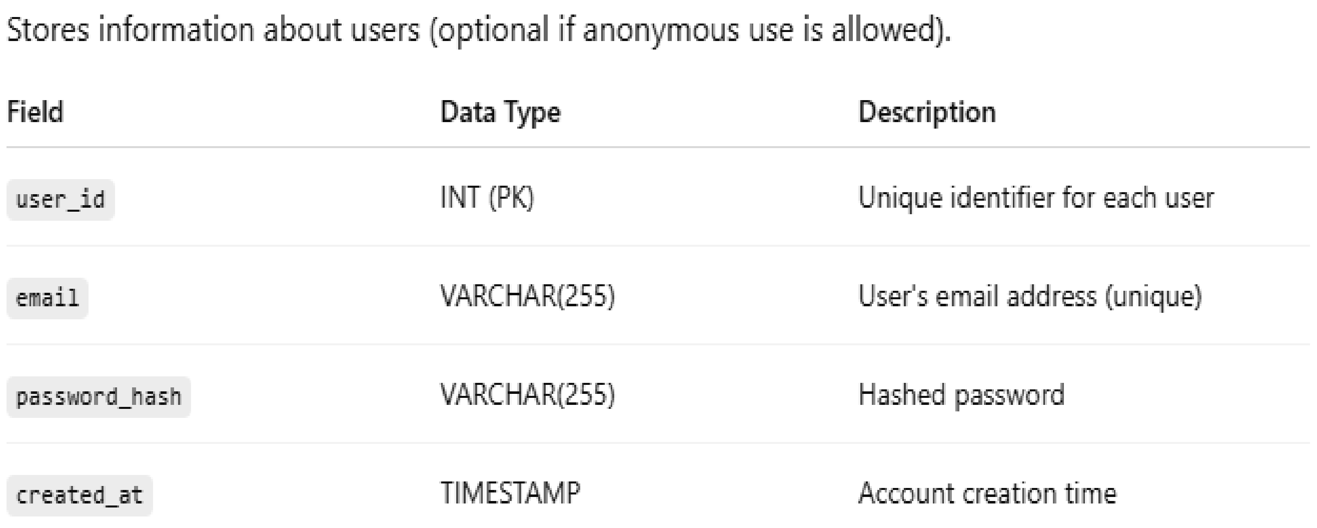


Fig.No.2.1

5.5 DATABASE DESIGN

The PDF Genie database stores vector embeddings of PDF text chunks along with metadata like document ID, chunk ID, and text position. This setup enables fast similarity searches to retrieve relevant chunks for user queries. Embeddings and metadata are organized to support scalable, efficient retrieval using vector databases or indexing tools like FAISS. The database may also track user queries and logs to improve performance. Overall, this design ensures quick access to context-rich information needed for generating accurate answers.

5.5.1 DATABASE TABLE



Stores information about users (optional if anonymous use is allowed).

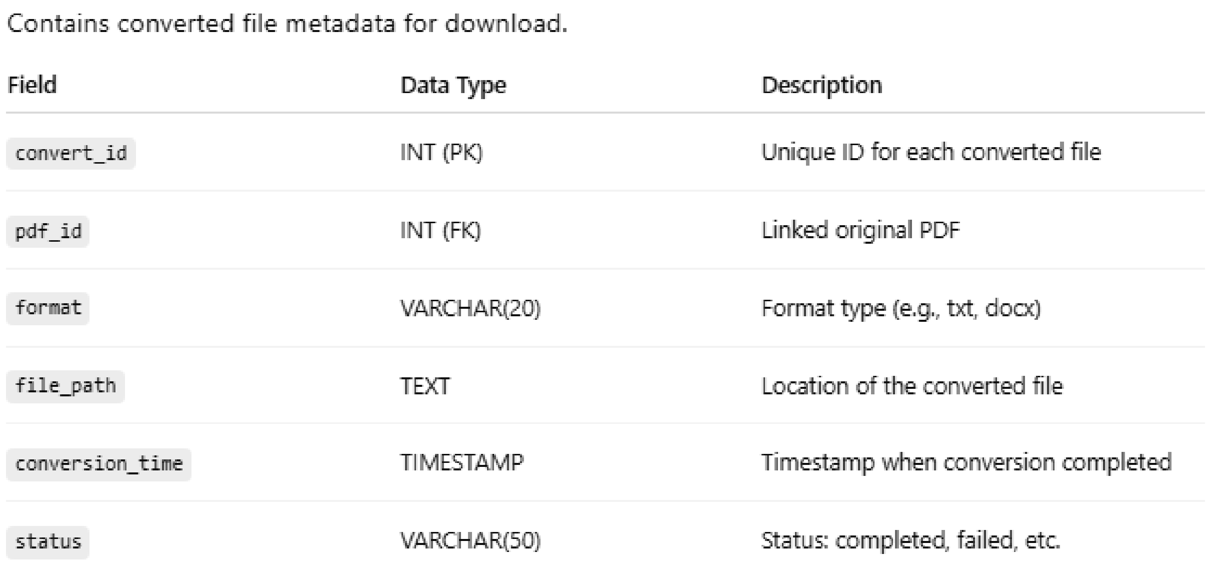
Field	Data Type	Description
`user_id`	INT (PK)	Unique identifier for each user
`email`	VARCHAR(255)	User's email address (unique)
`password_hash`	VARCHAR(255)	Hashed password
`created_at`	TIMESTAMP	Account creation time

Stores uploaded PDF file metadata and status.

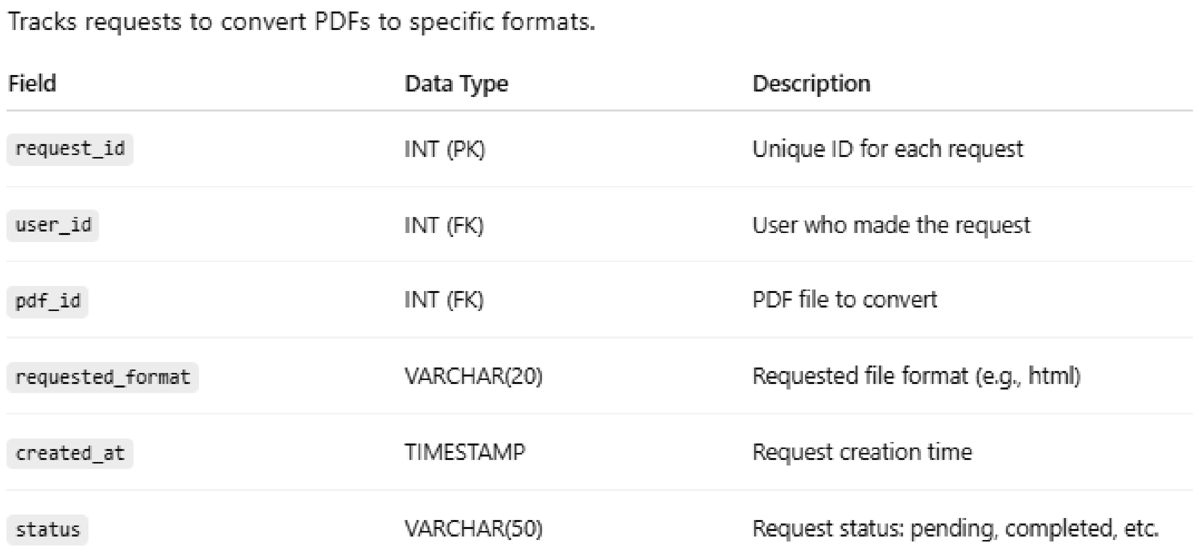
Field	Data Type	Description
`pdf_id`	INT (PK)	Unique ID for each uploaded PDF
`user_id`	INT (FK)	Linked user who uploaded the file
`file_name`	VARCHAR(255)	Original name of the uploaded file
`upload_path`	TEXT	Path where the file is stored
`upload_time`	TIMESTAMP	Upload timestamp
`status`	VARCHAR(50)	File status: uploaded, processed, etc.

Stores text and metadata extracted from the PDFs.

Field	Data Type	Description
`extract_id`	INT (PK)	Unique ID for each extract record
`pdf_id`	INT (FK)	Reference to uploaded PDF
`extracted_text`	TEXT	Full extracted text from PDF
`page_count`	INT	Number of pages in the PDF
`language`	VARCHAR(10)	Detected language of the content
`extract_time`	TIMESTAMP	Time of extraction


Contains converted file metadata for download.

Field	Data Type	Description
`convert_id`	INT (PK)	Unique ID for each converted file
`pdf_id`	INT (FK)	Linked original PDF
`format`	VARCHAR(20)	Format type (e.g., txt, docx)
`file_path`	TEXT	Location of the converted file
`conversion_time`	TIMESTAMP	Timestamp when conversion completed
`status`	VARCHAR(50)	Status: completed, failed, etc.


Tracks requests to convert PDFs to specific formats.

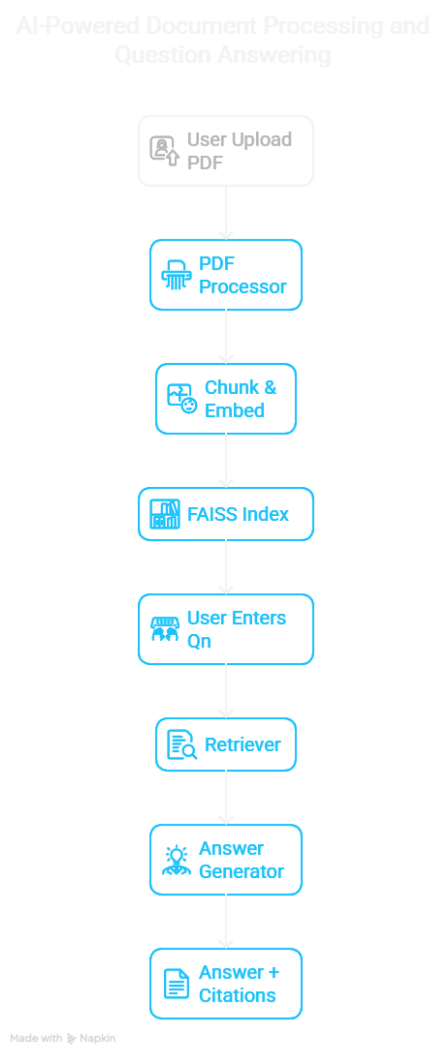
Field	Data Type	Description
`request_id`	INT (PK)	Unique ID for each request
`user_id`	INT (FK)	User who made the request
`pdf_id`	INT (FK)	PDF file to convert
`requested_format`	VARCHAR(20)	Requested file format (e.g., html)
`created_at`	TIMESTAMP	Request creation time
`status`	VARCHAR(50)	Request status: pending, completed, etc.

5.6 INPUT DESIGN

The input design of PDF Genie focuses on providing a simple and intuitive way for users to interact with the system. Users can upload PDF documents through a user-friendly interface, which accepts various PDF formats and validates file integrity. Additionally, users can enter natural language queries to search and extract information from the uploaded PDFs. The system processes these inputs by extracting text, chunking it, and converting both document chunks and queries into embeddings for efficient retrieval and response generation. Clear input validation and feedback mechanisms ensure smooth user experience and error handling.

- ➢ User uploads PDF files via an easy-to-use interface.
- ➢ Supports multiple PDF formats with file validation.
- ➢ Users enter natural language queries related to PDF content.
- ➢ System validates inputs to prevent errors and ensures proper formatting.
- ➢ Inputs (PDF and queries) are processed to extract and chunk text for embedding.
- ➢ Clear feedback and error messages improve user experience.
- ➢ Designed to be simple and responsive for smooth interaction.

5.7 SYSTEM ARCHITECTURE



System Architecture

5.7.1 TEXT PREPROCESSING

In the PDF Genie project, text preprocessing is a crucial step that prepares raw text extracted from PDF documents for further processing. After the PDF is uploaded, the text is cleaned by removing unwanted characters, line breaks, extra spaces, and non-informative content. The text is then normalized—such as converting to lowercase, fixing encodings, or handling special characters—to ensure consistency. This cleaned and standardized text is then passed to the chunking module, which splits the text into smaller, manageable units for embedding. Proper text preprocessing ensures that irrelevant noise is removed, improving the quality of embeddings and the accuracy of query responses.

Text Cleaning

Once the PDF is uploaded and the raw text is extracted, the first step involves cleaning the text. This includes:

- Removing unwanted characters (e.g., \n, \t, non-printable symbols)
- Eliminating extra spaces and line breaks
- Stripping out non-informative elements such as headers, footers, and page numbers

Text Normalization

After cleaning, the text undergoes normalization to ensure consistency in format. Key normalization steps include:

- Converting all text to **lowercase**
- Fixing **character encoding issues**
- Handling or removing **special characters** (e.g., emojis, symbols)
- Standardizing punctuation

Chunking Preparation

The cleaned and normalized text is then sent to the **chunking module**, where it is split into smaller, manageable units. This is essential for:

- Improving the performance of embedding models
- Enabling more accurate and relevant query responses
- Reducing processing complexity in downstream modules

Importance of Preprocessing

Proper text preprocessing ensures that irrelevant or noisy information does not affect the performance of embedding models or the accuracy of the final results. It plays a foundational role in ensuring the reliability and efficiency of the PDF Genie system.

CHAPTER 6

SYSTEM TESTING

TESTING DESGRIPTION

System testing in the PDF Genie project ensures that all integrated components such as PDF loading, text processing, embedding, retrieval, and response generation work together as expected. It involves testing the complete system as a whole to verify functional and non-functional requirements. The goal is to identify any defects that arise from module interactions and to ensure that the system behaves correctly under various conditions. Test cases cover scenarios like valid and invalid PDF uploads, query accuracy, retrieval quality, and interface responsiveness. This phase confirms that the system delivers accurate results, handles errors gracefully, and provides a smooth user experience before deployment.

6.1 UNIT TESTING

Unit testing in the PDF Genie project focuses on verifying the functionality of individual modules in isolation. Each module—such as pdf_loader.py, chunker.py, embedder.py, and vector_store.py—is tested separately to ensure it performs its intended function correctly. For example, the PDF loader is tested to confirm accurate text extraction, while the chunker is tested for proper splitting logic. These tests are typically automated and include both valid and edge case inputs. Unit testing helps identify bugs early in development, ensures code reliability, and simplifies debugging by narrowing issues down to specific components before integration.

6.2 INTEGRATION TESTING

Integration testing in the PDF Genie project ensures that all individual modules work together seamlessly when combined. After unit testing each component—such as text extraction, chunking, embedding, vector storage, and query handling these modules are integrated and tested as a group. The focus is on data flow between modules, correct input-output mapping, and smooth communication across components. For example, the output of the pdf_loader.py must be properly chunked by chunker.py, and those chunks must be correctly embedded and stored. Integration testing helps detect interface mismatches, data formatting issues, and functional gaps that may not appear during unit testing, ensuring the overall system functions smoothly as one cohesive pipeline.

6.3 FUNCTIONAL TESTING

Functional testing is a critical process in the PDF Genie project that verifies whether each individual feature operates according to the specified requirements. This involves thoroughly testing core functionalities such as uploading PDF documents, extracting text accurately, breaking down the text into chunks, generating embeddings, and responding correctly to user queries. The goal is to ensure that every component performs its intended task without errors or failures. For example, during functional testing, the system is checked to confirm it correctly extracts text from a variety of PDF formats and returns relevant content when queried. This testing helps catch issues early, ensuring that the fundamental building blocks of the system work seamlessly before integrating them into a complete pipeline.

6.4 PERFORMANCE TESTING

Performance testing evaluates the speed, responsiveness, and stability of the PDF Genie system under different operational conditions. This type of testing is crucial to understand how the system behaves when processing large PDF documents or handling multiple user queries simultaneously. The objective is to identify bottlenecks or delays in the retrieval and answer generation process. For instance, performance testing measures the time taken for the system to extract text, perform similarity searches on vector embeddings, and generate accurate answers. Ensuring quick response times and smooth handling of workloads is essential for providing a positive user experience, especially when dealing with large-scale document repositories.

6.5 REGRESSION TESTING

Regression testing is conducted after any modifications to the codebase, such as bug fixes, feature additions, or updates, to verify that existing functionalities continue to work as expected. This testing prevents previously resolved issues from resurfacing and ensures that new changes do not introduce unintended side effects. In the context of PDF Genie, regression testing might include rerunning tests that verify text extraction, chunking, embedding, and querying to confirm that all modules still integrate correctly and deliver consistent results. This ongoing validation process helps maintain system reliability throughout the development lifecycle.

6.6 VALIDATION TESTING

Validation testing focuses on assessing whether the overall system meets the defined user requirements and project goals. It evaluates the quality and relevance of the answers generated by the system when users submit queries. This includes verifying that the system not only retrieves correct information from the PDF documents but also presents it in a clear, coherent, and contextually appropriate manner. Validation testing ensures that the system fulfills its intended purpose and delivers value to users by providing accurate and meaningful responses. It often involves feedback from actual users or domain experts to confirm that the application aligns with expectations.

6.7 ERROR HANDLING TESTING

Error handling testing is designed to verify that the PDF Genie system can gracefully manage invalid or unexpected inputs without crashing or producing incorrect results. This includes scenarios such as uploading corrupted or unsupported PDF files, submitting malformed queries, or encountering missing data in documents. The system should provide informative and user-friendly error messages that guide users to correct their inputs or understand issues. Effective error handling contributes to the robustness and usability of the application by preventing abrupt failures and enhancing user trust and satisfaction.

CHAPTER 7

SYSTEM IMPLEMENTATION

The system implementation phase of the PDF Genie project marks the transition from design concepts to a fully functioning application capable of processing PDF documents and responding to user queries effectively. This phase begins with setting up a suitable development environment, including selecting programming languages, frameworks, and tools that align with the project goals such as Python for backend processing, vector databases like FAISS for efficient similarity searches, and Streamlit for building a user-friendly interface. Each module, including PDF loading, text extraction, chunking, embedding generation, vector storage, and the retrieval-augmented generation (RAG) pipeline, is developed individually following the design specifications. The PDF loader module handles robust text extraction from various PDF formats, ensuring accuracy and completeness. The chunker module divides the extracted text into manageable segments to facilitate embedding and retrieval processes. Embedding generation converts text chunks into high-dimensional vectors that capture semantic meaning, enabling efficient similarity search within the vector store. The vector storage system is implemented to index and store these embeddings with associated metadata, allowing quick retrieval of relevant information in response to user queries. Integration of these modules is carefully managed to ensure seamless data flow, error handling, and performance optimization.

7.1 CODING

Coding in the PDF Genie project involves converting the detailed design specifications into functional code. The development team writes scripts and programs for each module, such as PDF loading, text chunking, embedding generation, vector storage, and the retrieval pipeline. Python is primarily used due to its powerful libraries for text processing and machine learning. The coding phase ensures that the planned system logic, data flow, and algorithms are accurately implemented and ready for testing.

7.2 INSTALLATION

Installation refers to deploying the PDF Genie system in the intended environment, replacing any previous tools or manual processes used for PDF information retrieval. This includes setting up the application on servers or user machines, configuring dependencies like vector databases and ML models, and migrating any existing data if applicable. Proper installation ensures that the system is fully operational and accessible for end users.

7.3 DOCUMENTATION

Documentation is created alongside installation to provide users and developers with clear instructions on how to use and maintain PDF Genie. This includes user manuals explaining how to upload PDFs, enter queries, and interpret answers, as well as technical documentation covering system architecture, module functions, and troubleshooting guides. Good documentation supports smooth user onboarding and future system updates.

7.4 TRAINING AND SUPPORT

Training in the PDF Genie project equips users with the knowledge and skills to effectively operate the system. Training sessions cover uploading PDFs, formulating queries, and understanding the results. Personalized, hands-on training may be provided to administrators or frequent users to address specific questions and workflows. Continuous support is offered through help desks, FAQs, and regular updates to ensure users can maximize the system’s benefits and address any issues promptly.

CHAPTER 8

CONCLUSION & FUTURE ENHANCEMENT

8.1 CONCLUSION

The proposed Retrieval-Augmented Generation (RAG) system offers a robust, offline solution for extracting meaningful insights from complex PDF documents, including academic papers, legal contracts, and technical manuals. By combining advanced text processing, semantic embeddings, and natural language generation, the system enables users to interact with documents intelligently and efficiently. Its modular architecture, offline capability, and emphasis on accuracy, usability, and security make it a practical and scalable tool for real-world deployment. This solution significantly enhances information accessibility while preserving document context and reliability.

8.2 FUTURE ENHANCEMENTS

The PDF Genie project offers a solid foundation for intelligent PDF document querying, but there are several opportunities for future enhancements to improve its functionality, scalability, and user experience. One key area is expanding support for a wider variety of document formats beyond PDFs, such as Word documents, PowerPoint presentations, and scanned images using OCR (Optical Character Recognition). Additionally, integrating more advanced natural language processing techniques, like transformer-based models or domain-specific fine-tuning, can enhance the accuracy and contextual understanding of user queries. Improving the user interface by adding features such as interactive visualizations, query suggestions, and personalized user profiles could greatly enhance usability. Scalability can be addressed by optimizing the vector database for larger document repositories and enabling distributed processing for faster response times. Security enhancements, including user authentication and data encryption, would ensure safe handling of sensitive documents.

APPENDICES

SOURCE CODE

```
# FAISS vector store
import faiss
import numpy as np


class VectorStore:
    def __init__(self):
        self.index = None
        self.texts = []

    def add(self, embeddings, texts):
        if self.index is None:
            self.index = faiss.IndexFlatL2(embeddings.shape[1])
        self.index.add(embeddings)
        self.texts.extend(texts)

    def query(self, embedding, top_k=5):
        distances, indices = self.index.search(embedding, top_k)
        return [self.texts[i] for i in indices[0]]
```

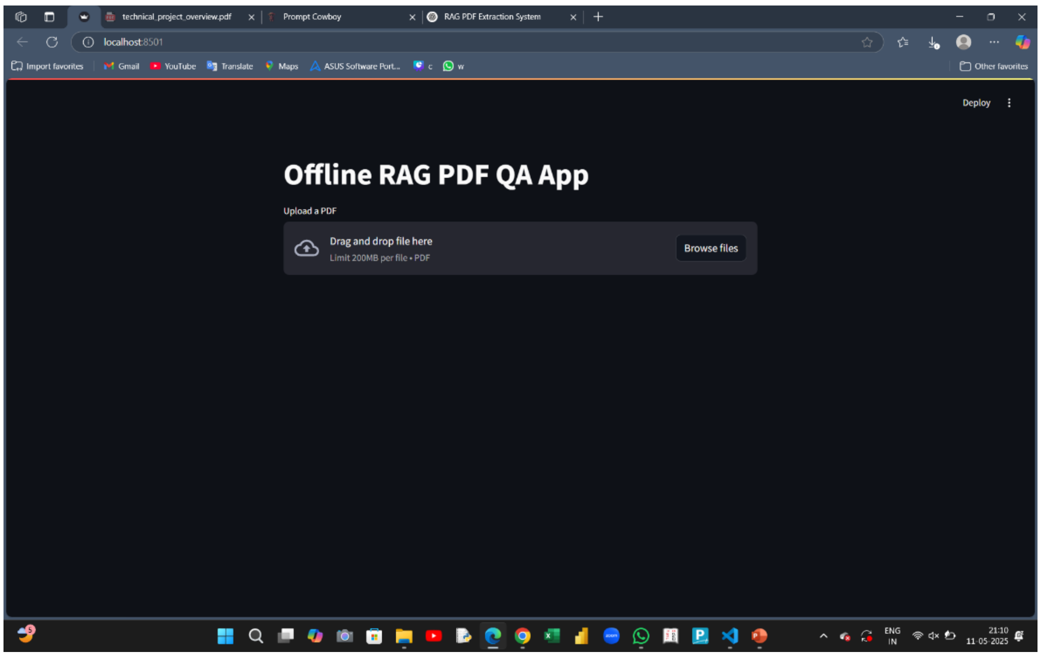
```
import re

from typing import List


def chunk_text(text: str, chunk_size: int = 500, overlap: int = 100) -> List[str]:

    """

    Chunk text into semantically meaningful pieces while preserving context.


    Args:

        text: Input text to chunk

        chunk_size: Target size for each chunk

        overlap: Number of characters to overlap between chunks


    Returns:

        List of text chunks

    """

    # Clean and normalize text

    text = re.sub(r'\s+', ' ', text).strip()


    # Split into sentences (basic implementation)

    sentences = re.split(r'(?<=[.!?])\s+', text)


    chunks = []
    current_chunk = []
    current_length = 0

    for sentence in sentences:
        sentence_length = len(sentence)

        # If adding this sentence would exceed chunk size
        if current_length + sentence_length > chunk_size and current_chunk:
            # Store current chunk
            chunks.append(' '.join(current_chunk))
            # Start new chunk with overlap
            overlap_point = max(0, len(current_chunk) - 2)  # Keep last 2 sentences
for context
            current_chunk = current_chunk[overlap_point:]
            current_length = sum(len(s) for s in current_chunk)

        current_chunk.append(sentence)
        current_length += sentence_length

    # Add the last chunk if it exists
    if current_chunk:
        chunks.append(' '.join(current_chunk))
    return chunks


# RAG pipeline logic
from modules.embedder import embed_chunks
from modules.vector_store import VectorStore
from modules.chunker import chunk_text
from transformers import pipeline, AutoTokenizer,
AutoModelForQuestionAnswering
import torch
import re


# Initialize a more precise QA model
MODEL_NAME = "deepset/deberta-v3-large-squad2"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForQuestionAnswering.from_pretrained(MODEL_NAME)
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)


store = VectorStore()


def clean_context(text):
    """Clean and format context for better QA."""
    # Remove multiple spaces and newlines
    text = re.sub(r'\s+', ' ', text)
    # Remove special characters but keep basic punctuation
    text = re.sub(r'[^a-zA-Z0-9\s.,!?;:()\-\'"]', ' ', text)
    return text.strip()

def get_answer_from_context(question, context):
    """Get precise answer using token-level analysis."""
    # Tokenize input
    inputs = tokenizer(
        question,
        context,
        max_length=512,
        truncation=True,
        stride=128,
        return_overflowing_tokens=True,
        return_offsets_mapping=True,
        padding=True,
        return_tensors="pt"
    ).to(device)
    # Get model predictions
    with torch.no_grad():
        outputs = model(**{k: v for k, v in inputs.items() if k not in
['offset_mapping', 'overflow_to_sample_mapping']})

    # Get answer span
    start_logits = outputs.start_logits
    end_logits = outputs.end_logits

    # Get the most likely answer span
    start_idx = torch.argmax(start_logits)
    end_idx = torch.argmax(end_logits)

    # Convert to list for processing
    input_ids = inputs["input_ids"][0].tolist()
    tokens = tokenizer.convert_ids_to_tokens(input_ids)
    answer = tokenizer.convert_tokens_to_string(tokens[start_idx:end_idx + 1])

    # Calculate confidence score
    start_score = torch.max(torch.softmax(start_logits, dim=1)).item()
    end_score = torch.max(torch.softmax(end_logits, dim=1)).item()
    confidence = (start_score + end_score) / 2
    return answer, confidence


def build_knowledge_base(text):
    chunks = chunk_text(text)
    embeddings = embed_chunks(chunks)
    store.add(embeddings, chunks)


def answer_question(question):
    try:
        # Get relevant chunks with overlap
        q_embedding = embed_chunks([question])
        relevant_chunks = store.query(q_embedding, top_k=5)

        # Process each chunk separately and get the best answer
        best_answer = None
        best_confidence = 0
        debug_info = []

        # Try different context windows
        for i, chunk in enumerate(relevant_chunks):
            # Clean the context
            clean_chunk = clean_context(chunk)

            # Get answer from this chunk
            answer, confidence = get_answer_from_context(question, clean_chunk)
            debug_info.append(f"Chunk {i+1} confidence: {confidence:.3f}")

            # Update best answer if confidence is higher
            if confidence > best_confidence:
                best_answer = answer
                best_confidence = confidence

        # Validate and format the answer
        if best_confidence < 0.2 or not best_answer or len(best_answer.strip()) < 2:
            debug_str = "\n".join(debug_info)
            return f"I'm not confident about finding a precise answer to this
question. Please try rephrasing it or check if the information is in the
document.\n\nDebug info:\n{debug_str}"

        # Clean up the answer
        best_answer = best_answer.strip()
        best_answer = re.sub(r'\s+', ' ', best_answer)
```

```
        return f"Answer: {best_answer}\n(Confidence: {best_confidence:.2f})"

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



# Streamlit interface
import streamlit as st
from modules.pdf_loader import extract_text
from modules.rag_pipeline import build_knowledge_base, answer_question



def run_app():
    st.title("Offline RAG PDF QA App")

    uploaded_file = st.file_uploader("Upload a PDF", type="pdf")
    if uploaded_file:
        pdf_path = f"data/{uploaded_file.name}"
        with open(pdf_path, "wb") as f:
            f.write(uploaded_file.read())
        st.success("PDF uploaded successfully")
```

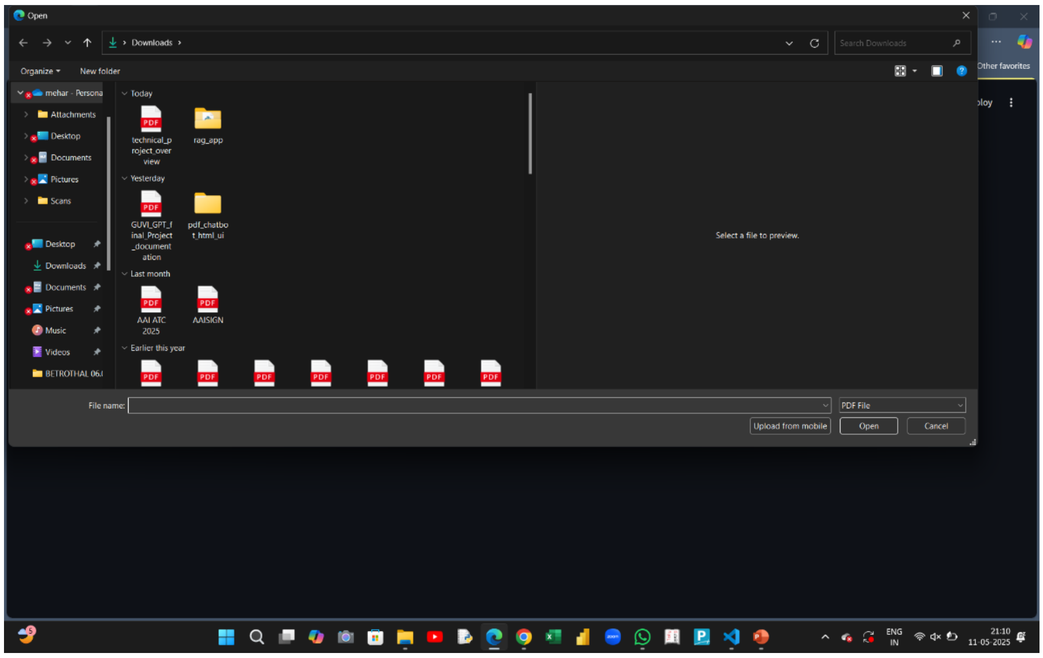
```
text = extract_text(pdf_path)
if text.strip():
    st.info("Building knowledge base...")
    build_knowledge_base(text)
    st.success("Ready to answer questions!")

    query = st.text_input("Ask a question about the PDF:")
    if query:
        answer = answer_question(query)
        st.text_area("Answer:", value=answer, height=300)
else:
    st.error("Could not extract any text from this PDF.")
```

A. SCREENSHOTS

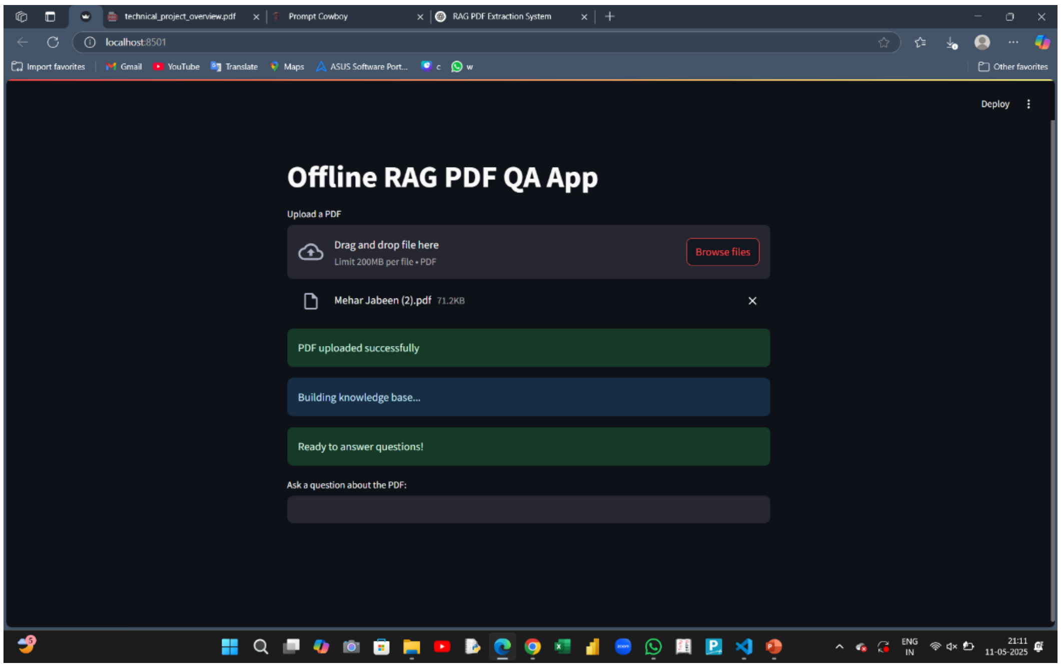
A1. Offline RAG PDF QA App UI Screenshot



A2. PDF File Selection Dialog in RAG App



A3. PDF Upload and Knowledge Base Initialization in RAG App



Offline RAG PDF QA App

Upload a PDF

Drag and drop file here
Limit 200MB per file • PDF

Browse files

Mehar Jabeen (2).pdf 71.2KB

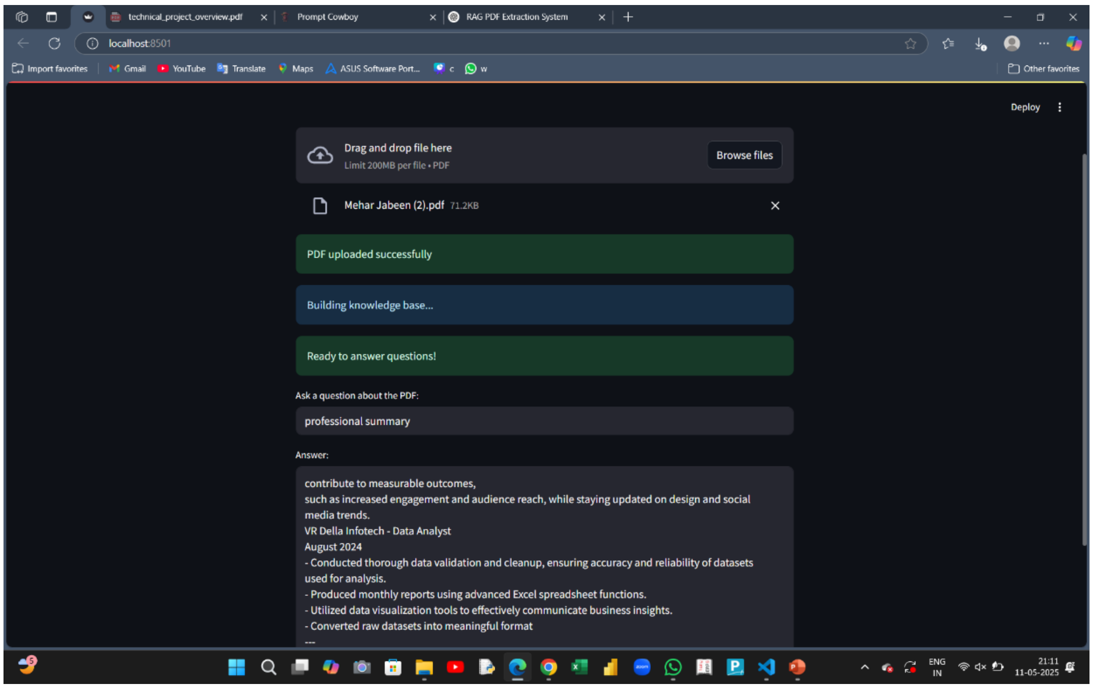
PDF uploaded successfully

Building knowledge base...

Ready to answer questions!

Ask a question about the PDF:

A4. PDF QA Response Display for



Drag and drop file here
Limit 200MB per file • PDF

Browse files

Mehar Jabeen (2).pdf 71.2KB

PDF uploaded successfully

Building knowledge base...

Ready to answer questions!

Ask a question about the PDF:

professional summary

Answer:

contribute to measurable outcomes,
such as increased engagement and audience reach, while staying updated on design and social media trends.
VR Della Infotech - Data Analyst
August 2024
- Conducted thorough data validation and cleanup, ensuring accuracy and reliability of datasets used for analysis.
- Produced monthly reports using advanced Excel spreadsheet functions.
- Utilized data visualization tools to effectively communicate business insights.
- Converted raw datasets into meaningful format

REFERENCES

1. Chollet, F. (2018). *Deep Learning with Python*. Manning Publications.
2. Jurafsky, D., & Martin, J. H. (2020). *Speech and Language Processing* (3rd ed.). Pearson Education.
3. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
4. Tan, C. M., & Wee, L. H. (2021). *Natural Language Processing for Information Retrieval*. Springer.
5. Vaswani, A. et al. (2017). *Attention Is All You Need*. Advances in Neural Information Processing Systems (NeurIPS).

WEB REFERENCES

- https://python.langchain.com/ — Langchain official documentation
- https://www.pypdf.org/ — PDF handling in Python
- https://platform.openai.com/docs — OpenAI API documentation
- https://streamlit.io/ — Streamlit documentation for frontend development
- https://github.com/facebookresearch/faiss — FAISS vector database
- https://scikit-learn.org/ — Machine learning utilities and models