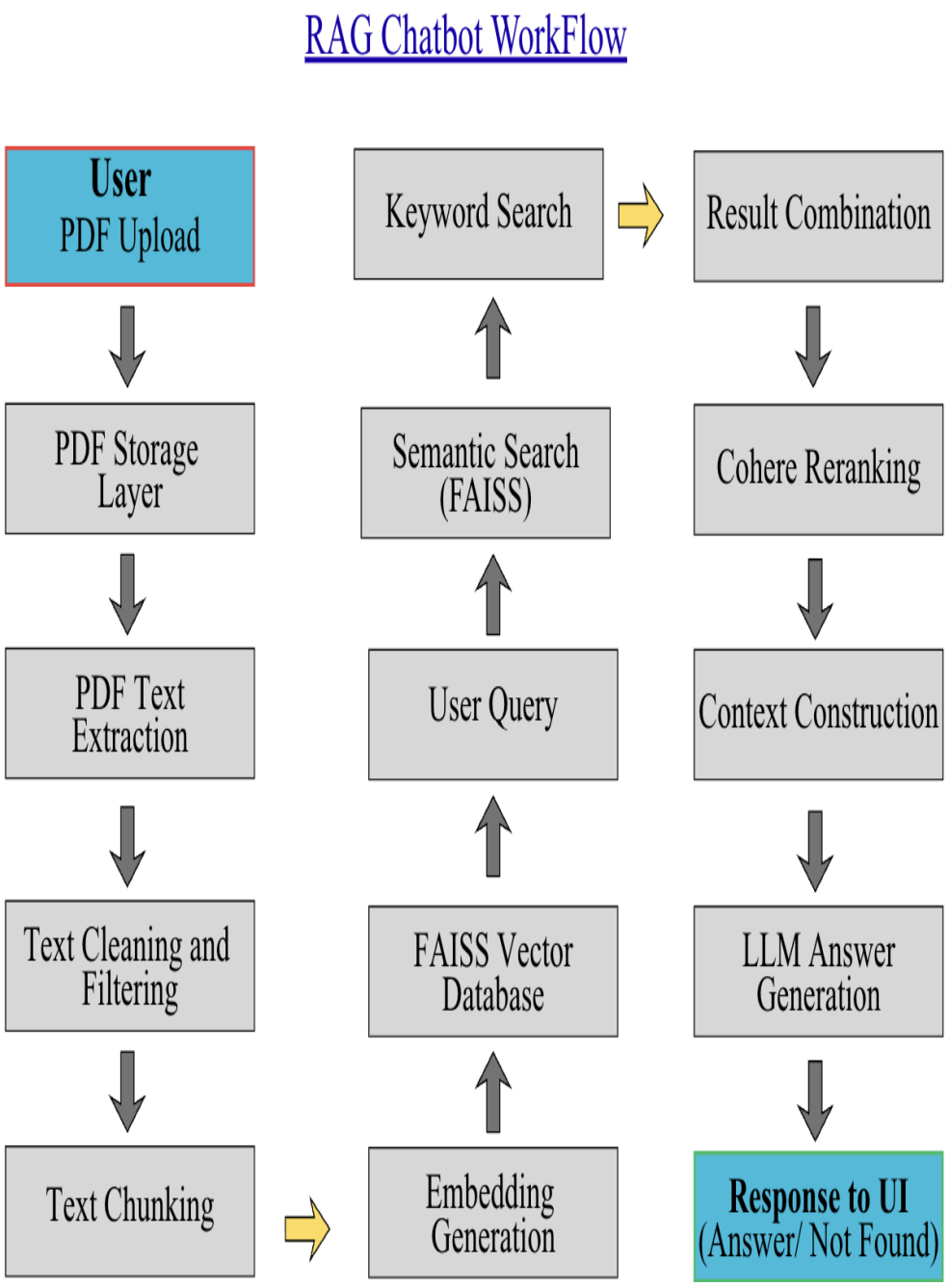
DOCUMIND RAG_DOCUMENTATION
A RAG BASED CHATBOT

The System follows a Retrieval Augmented Generation (RAG) architecture to answer the user questions/ prompts strictly based on the uploaded files.



1. EXPLANATION OF WORKFLOW

PDF Upload

- Users upload multiple PDF files (up to 10 files, including large documents).
- Files are saved safely to disk to avoid memory overload.

PDF Text Extraction

- Each PDF is read page by page using a PDF parsing library.
- Text is extracted from every page.
- Pages without readable text are skipped.
- Each page is treated as an individual document unit.

Text Chunking

- Extracted page text is split into overlapping chunks.
- Chunking ensures:
    - Large pages are broken into manageable sizes.
    - Context is preserved across chunks.

Embedding Generation

- A multilingual sentence transformer converts each chunk into a vector embedding.
- This allows semantic similarity search across multiple languages.

Vector Storage (FAISS)

- Embeddings are stored in a FAISS (Facebook AI Similarity Search) vector database.
- Indexing is done incrementally in batches to support very large datasets.
- Metadata such as file name and page number is stored with each chunk.

User Question Input

- The user enters a question through the designed chatbot UI.

Document Retrieval

- Semantic search retrieves the most relevant chunks from FAISS.
- Keyword-based search is also performed to capture exact facts (emails, Mobileno.)

Reranking

- At present, the system relies on high-quality semantic retrieval using FAISS embeddings.
- The architecture is designed to support reranking as a future enhancement if required.

Answer Generation

- The top reranked chunks are passed as context to the language model.
- The model generates a response strictly based on the retrieved document content.

- If no relevant information exists, the system responds with **“Answer not found in documents.”**

Response Display

- The final answer is generated using the **Cohere Large Language Model API** and displayed in the chatbot UI.

2. TOOLS AND TECHNOLOGIES

Programming Language **:** Python

Reason : Rich in ecosystem and highly can be integrated with Machine learning, Deep Learning, NLP and AI.

Backend API : Fast API

Key endpoints : /upload, /ask

Reason : High performance, async support and easy integration.

FrontEnd UI **:** Streamlit

Reason : Rapid development of clean interactive UI's.

Text Extraction **:** PyPdf/ PDF Reader

Reason : Reliable for the page-level PDF parsing.

Text Splitting and Vector Integration **:** Langchain

Reason : Simplifies chunking and vector workflows.

Embeddings **:** Sentence Transformers

Reason : Supports multiple languages and semantic search.

Vector Database **:** FAISS (Facebook AI Similarity Search)

Reason : 1. Fast, scalable, open-source vector search compared to Chroma.
2. FAISS was chosen over Chroma because it provides high-performance, scalable vector similarity search with fine-grained control over indexing, making it more suitable for production-grade RAG systems handling large document collections.

Retrieval Chunks **:** FAISS, Sentence Transformer Embeddings
Reason : Helpful in Improving generating answer accuracy and relevance.

Fact Extraction **:** Hybrid Retrieval (Semantic + Keyword)

Reason : Can be able to Handle both semantic queries and exact matches.

Environment Variable Management **:** .env
Reason : .env is mainly used for secure API key handling.

API : Cohere
Reason: Cohere is API is used because it is limitation free and Free of cost for unlimited trials and perfect for implementation and testing,
Gemini API also can be used but it is free-tier only for some limitation and while using Gemini API we can use the model **gemini-2.5-flash-lite**, which is a cost efficient model

Here im trying to implement the bot and test the responses, so if i have the limitation i cannot implement the bot it takes long time to wait for the limit requests to send to google API, for fast implementation i have used Cohere API

Model : Command–r-08-2024

This model is optimised for complex tasks and offers advanced language understanding, higher capacity and more responses than the model cohere **command-r,** which is a pro model, this **command-r-08-2024** can maintain context from long conversation history around 1.3k tokens.

3. PROJECT FILE STRUCTURE

```
Chatbot_Datai2i/
│
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── config.py
│   ├── pdf_loader.py
│   ├── vector_store.py
│   ├── rag.py
│   ├── cohere_llm.py
│   └── keyword_index.py
│
├── ui/
│   └── streamlit_app.py
│
├── data/
│   └── pdfs/
│       ├── file1.pdf
│       ├── file2.pdf
│       └── ...
│
├── faiss_db/
│   ├── index.faiss
│   └── index.pkl
│
├── .env (stores the api key)
├── requirements.txt
└── .venv/
```

And Some test files are included, these are the main files of the chatbot for working

4. INTERNAL CODE WORKING

4.1. User Interface Layer

File: `ui/streamlit_app.py`

- **Purpose:** Serves as the primary user interface.
- **Key Functions:** Enables users to upload PDF documents.
    - Provides a chat interface for users to submit questions.
    - Communicates with the backend to display generated answers and retrieved information.

4.2. Backend Controller

File: `app/main.py`

- **Purpose:** Acts as the central orchestrator for the application.
- **Key Functions:** Receives PDF files from the UI and saves them to the `data/pdfs` directory.
    - Triggers the data processing and indexing workflows.
    - Routes user queries to the retrieval system and returns the final response.

4.3. PDF Processing

File: `app/pdf_loader.py`

- **Purpose:** Handles document ingestion and text extraction.
- **Key Functions:** Iterates through all files in `data/pdfs`.
    - Extracts readable text from every page of the documents.
    - Attaches essential metadata to the text, including the **file name** and **page number**.

4.4. Vector Storage & Embeddings

File: `app/vector_store.py`

- **Purpose:** Manages the conversion of text into searchable numerical data.
- **Key Functions:** Splits extracted text into small, overlapping chunks to maintain context.
    - Converts text chunks into numerical embeddings using the **HuggingFace** model: **`sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2`**.
    - Stores embeddings in a **FAISS** vector database.
    - **Output:** Generates `index.faiss` and `index.pkl` within the `faiss_db` folder.

4.5. Retrieval Logic (RAG)

File: `app/rag.py`

- **Purpose:** Executes the core "Retrieval-Augmented Generation" logic.
- **Key Functions:** Performs **Semantic Search** to understand the intent behind a user's question.
    - Combines keyword-based results with vector search to ensure both context and facts are captured.
    - Selects the most relevant content snippets to provide to the LLM.

4.6. Keyword Indexing

File: `app/keyword_index.py`

- **Purpose:** Ensures high accuracy for specific data points.
- **Key Functions:** *Performs exact word matching.
    - Optimized for retrieving specific identifiers such as **emails, phone numbers, IDs,** and **bullet points**.

4.7. Large Language Model (LLM) Integration

File: `app/cohere_llm.py`

- **Purpose:** Generates the final natural language response.
- **Key Models:** `command-r-08-2024`
- **Key Functions:** Receives the user query and the retrieved document snippets.
    - Generates an answer strictly based on the provided text.
    - **Safety Check:** Returns "Answer not found in documents" if the information is missing from the uploaded PDFs.
    - The Model used in this chatbot is "`command-r-08-2024`"
    - `Embeddings: Sentence Transformers`

4.8. Configuration Management

File: `app/config.py`

- **Purpose:** Centralizes all system settings and environment variables.
- **Key Functions:** Defines and manages directory paths such as `DATA_DIR` (for PDFs) and `FAISS_DIR` (for the vector database).
    - Stores API keys and model parameters in a single location to ensure the code remains clean and maintainable.

- Allows for easy adjustments to system-wide settings without modifying core logic files.

Note1: To check the Chatbot is giving the correct answer, design the code files and code to check its output.

The code files are

1. **test_faiss_build.py**
2. **test_faiss_search.py**
3. **test_pdf_load.py**
4. **cohere_test.py**
5. **test_rag.py**

test_faiss_build.py

To check how the FAISS is done indexing of the uploaded documents, it's a small sample test working on small pdf's and 1 or 2 documents. To check its compatibility.

test_faiss_search.py

To check the extraction of text from the documents, simple data using the hugging model embeddings. It confirms the FAISS retrieval works by running this sample test.

test_pdf_load.py

It is used to check how the text operations are performed on the uploaded sample document like cleaning and removing spaces and shows us the first 500 characters and last 500 characters according to our compatibility.

test_rag.py

In this we are installing modules and inheriting the main rag code which is in the app and giving a *single question/ prompt* to test from the uploaded document, it shows how the context and answer will be given from the doc. We get a clear idea whether it's working or not.

Note2

In Some cases we do not get answer, it can be unclear whether issues originate from the API or from data ingestion, API problems like *Exhausted Quota* and *reaching the limits*, and confused about the models to use, because some models are introduced and some models are removed, we cannot know which model is supported to our system, to rectify this designed two code files

1. `cohere_test.py`
2. `Test_models.py`

`cohere_test.py`

In this code file, we will initialise our API key and will give a context which we know, we will give a prompt, which knows the answer from the context, if it responds correctly then the API is not the problem, the problem with the data ingestion, we can change our code there.

`test_models.py`

We use one model in LLM code, we does not know whether it is supported or not, in the time of execution is shows an error, likely we try with several models and do not know which is supported,

By running this code file by initialising the api in the code file, it delivered the list of models which our systems supports and which are valid, from the list we can select the suitable model.

EXPLANATION ON EMBEDDINGS

- **Embedding type:** Sentence Transformer
- **Provider:** Hugging Face
- **Model name:** `paraphrase-multilingual-MiniLM-L12-v2`
- **Language support:** Multilingual (English + many other languages)
- **Use case:** Semantic search for RAG
- **Vector DB:** FAISS

This model converts each text chunk from the PDFs into numerical vectors that capture meaning, not just keywords.

5. SYSTEM INTERFACES AND DEMONSTRATION

5.1 USER INTERFACE

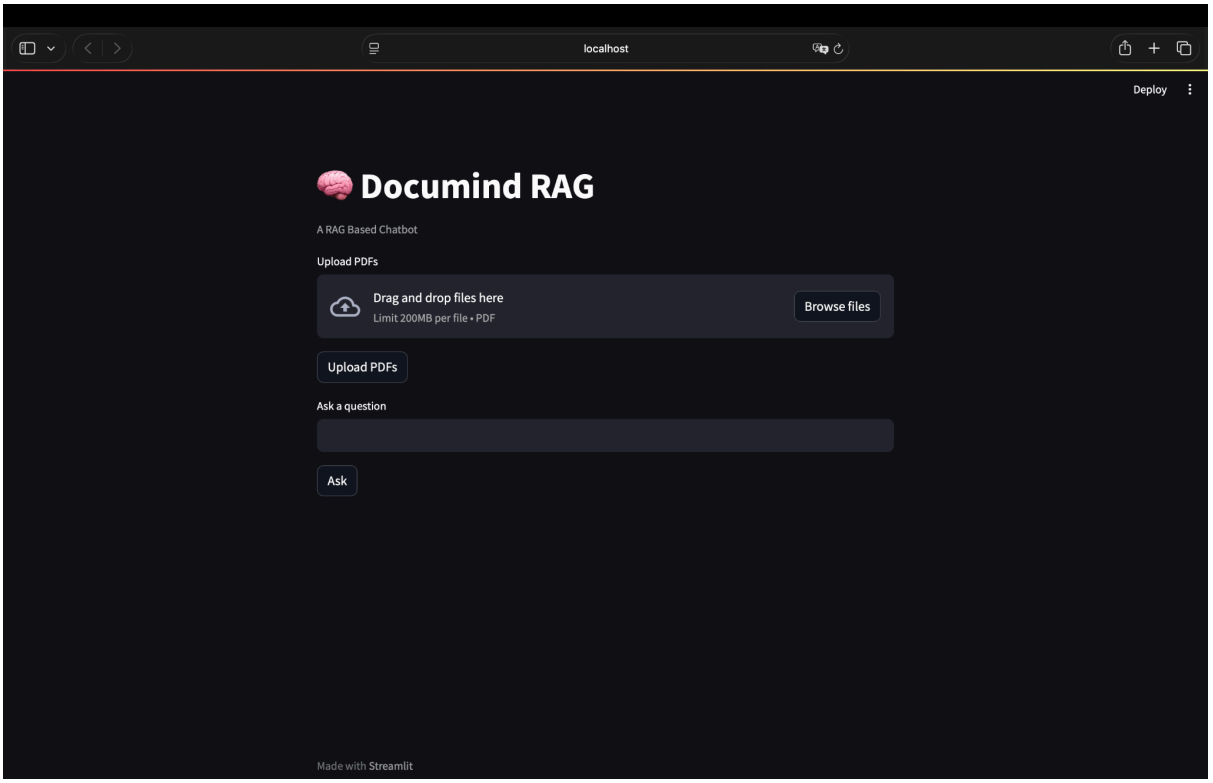


Fig.1: Streamlit interface for Files Uploading by User and To know status of Indexing

5.2 BACKEND API DOCUMENTATION (SWAGGER UI)

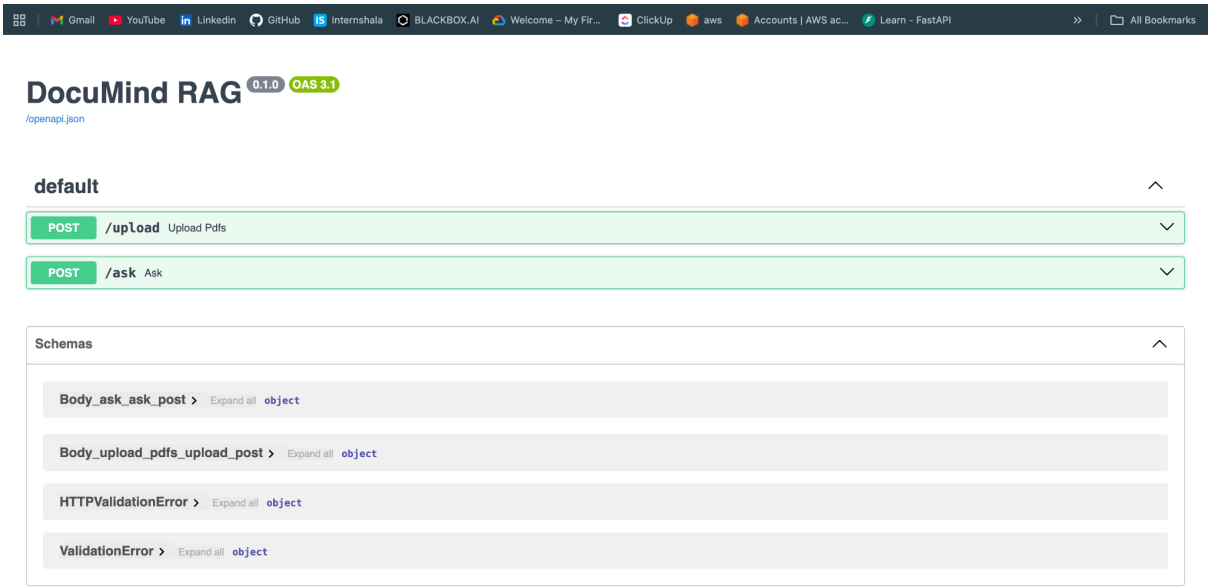


Fig.2: *This interface provides a real-time overview of the backend RESTful endpoints. It allows developers to test the* `/upload_pdf` *and* `/ask_question` *routes independently of the Streamlit frontend, ensuring the API logic and data validation are functioning correctly.*

5.3 EXECUTION EXAMPLES

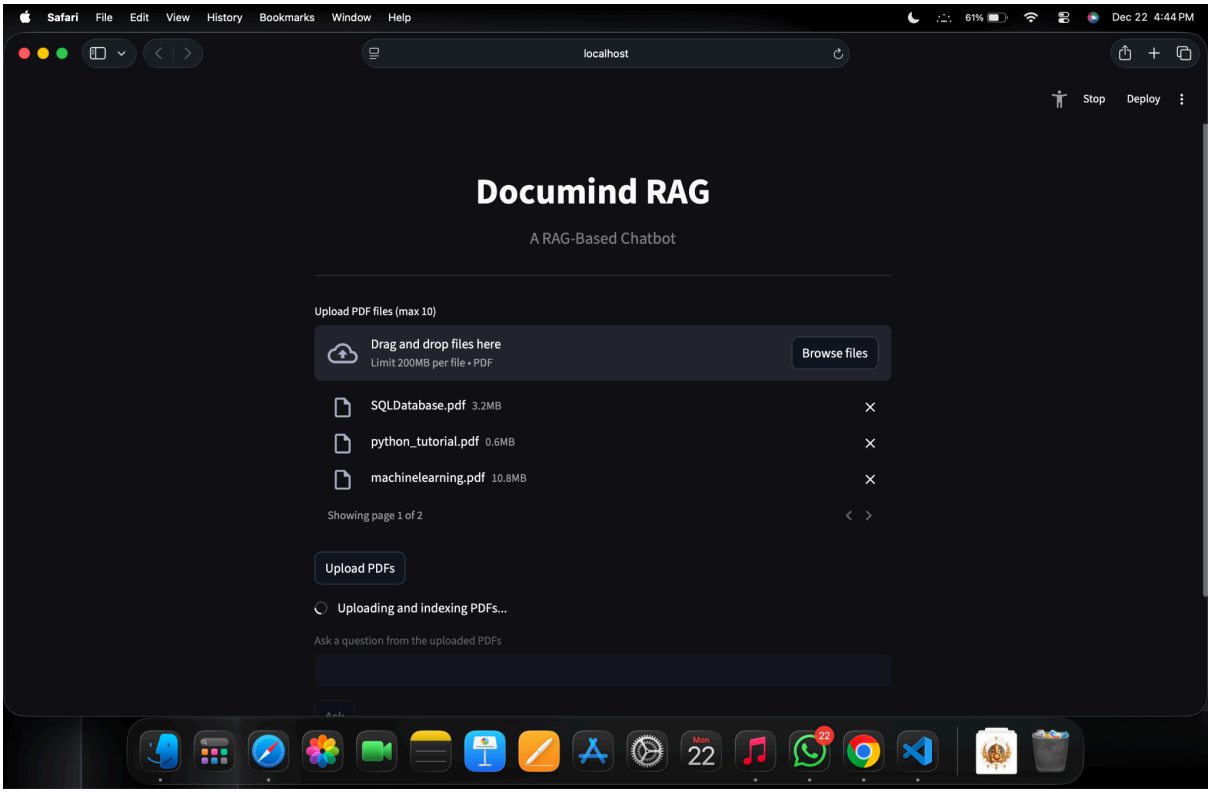


Fig.3: Processing Uploading & Indexing of files after Uploading

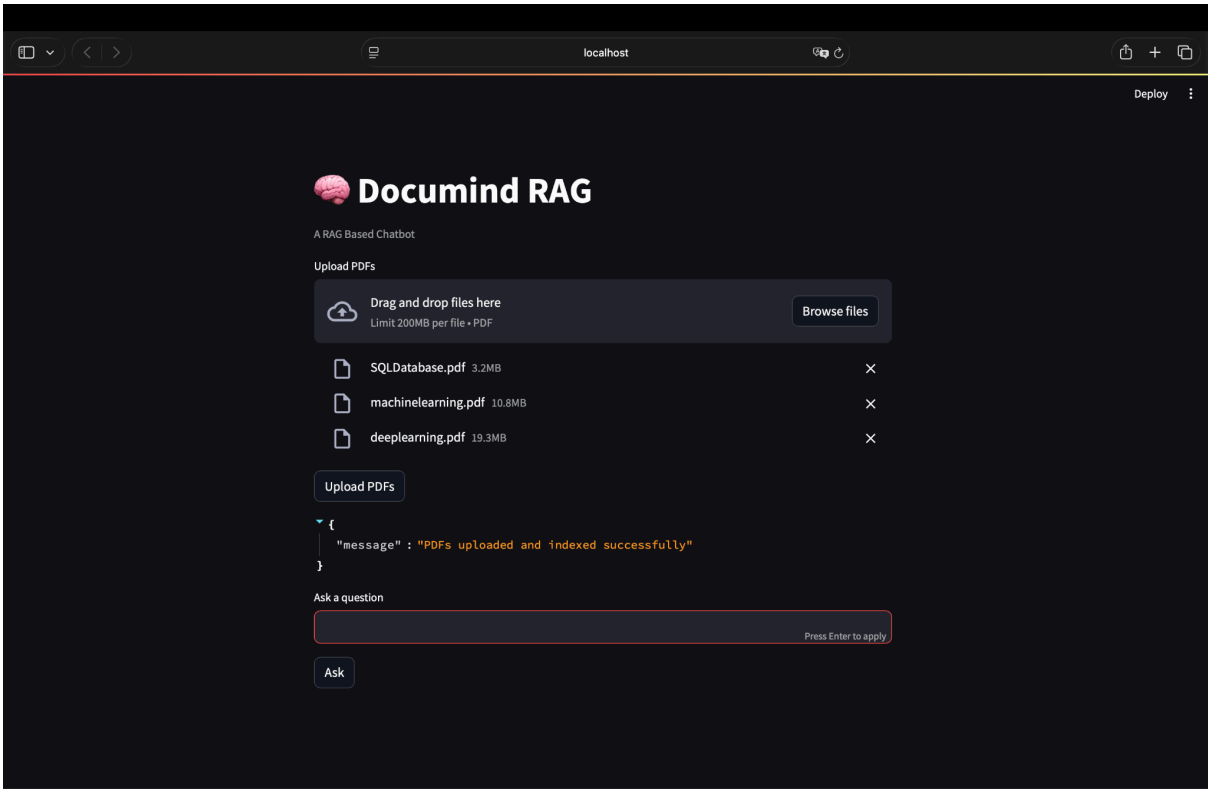


Fig.4: Successful Uploading of PDFs and Indexed

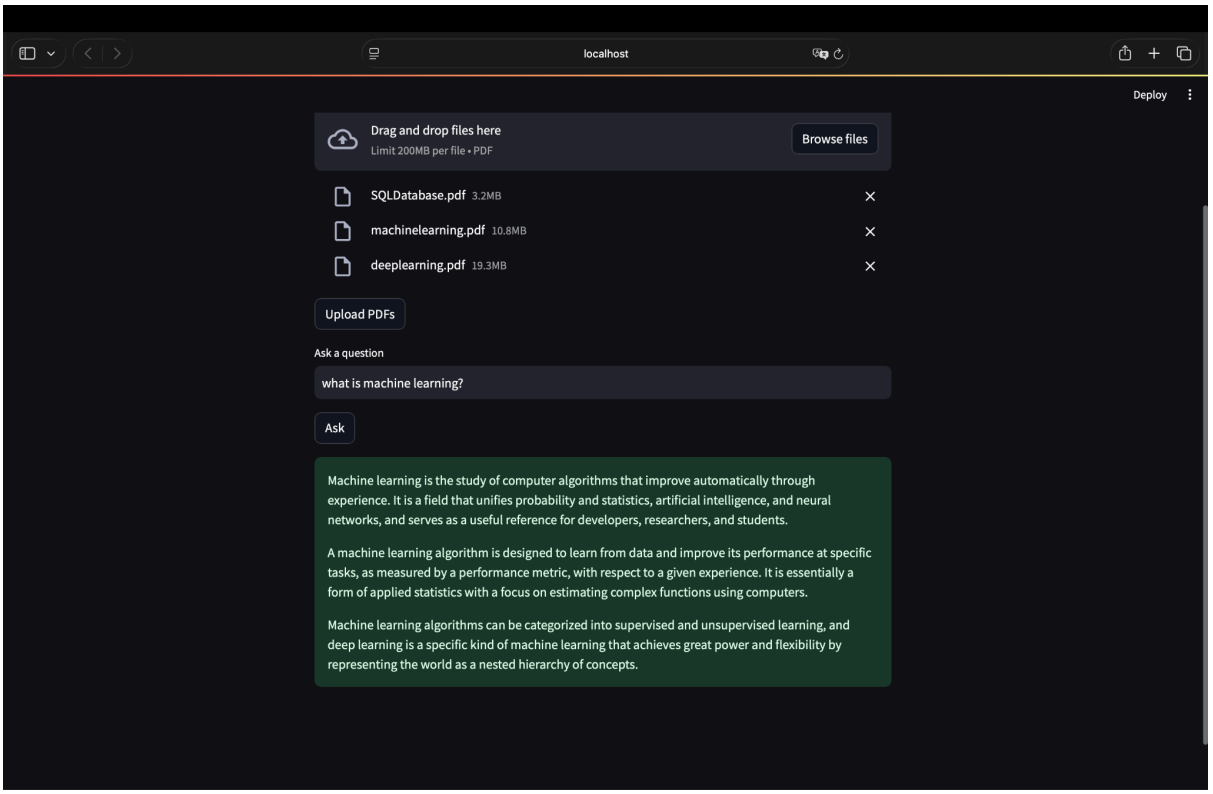


Fig.5: Successful Retrieval of Answer for the given Query Scenario -1

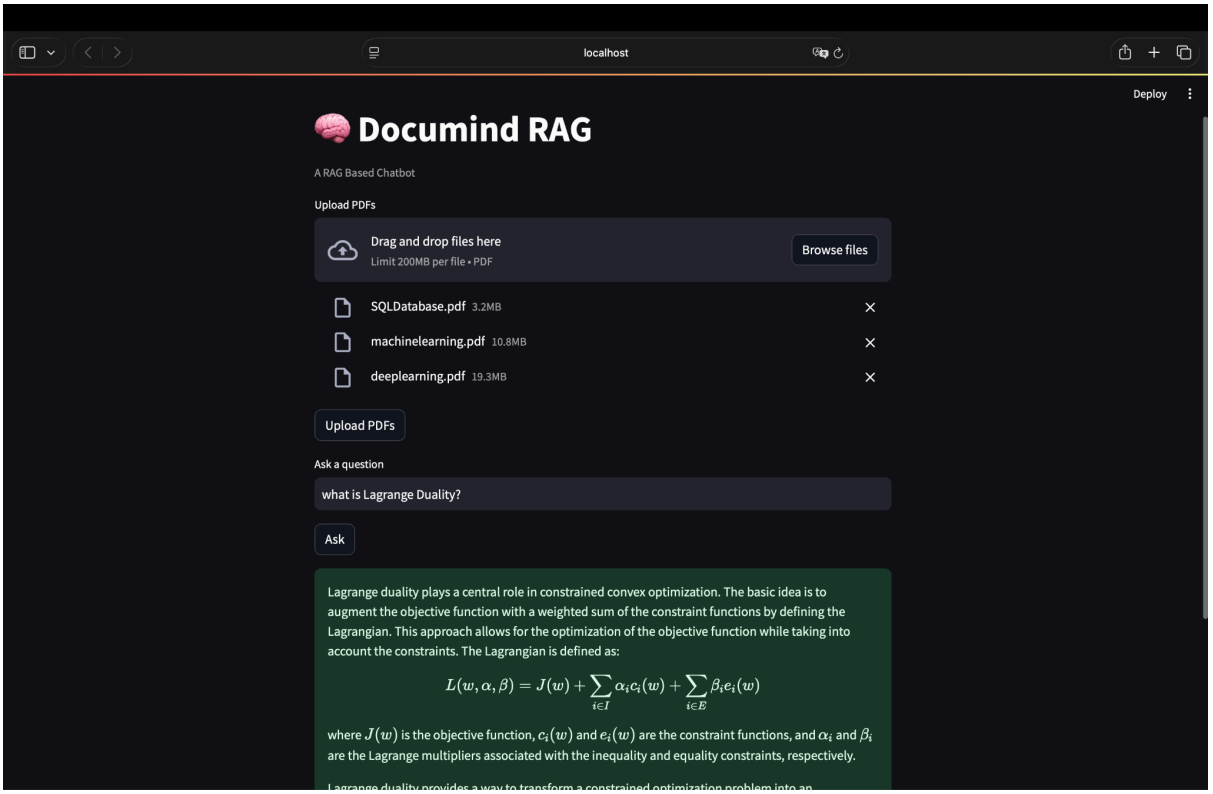


Fig.6: Successful Retrieval of Answer to the Query from the Documentation Scenario-2

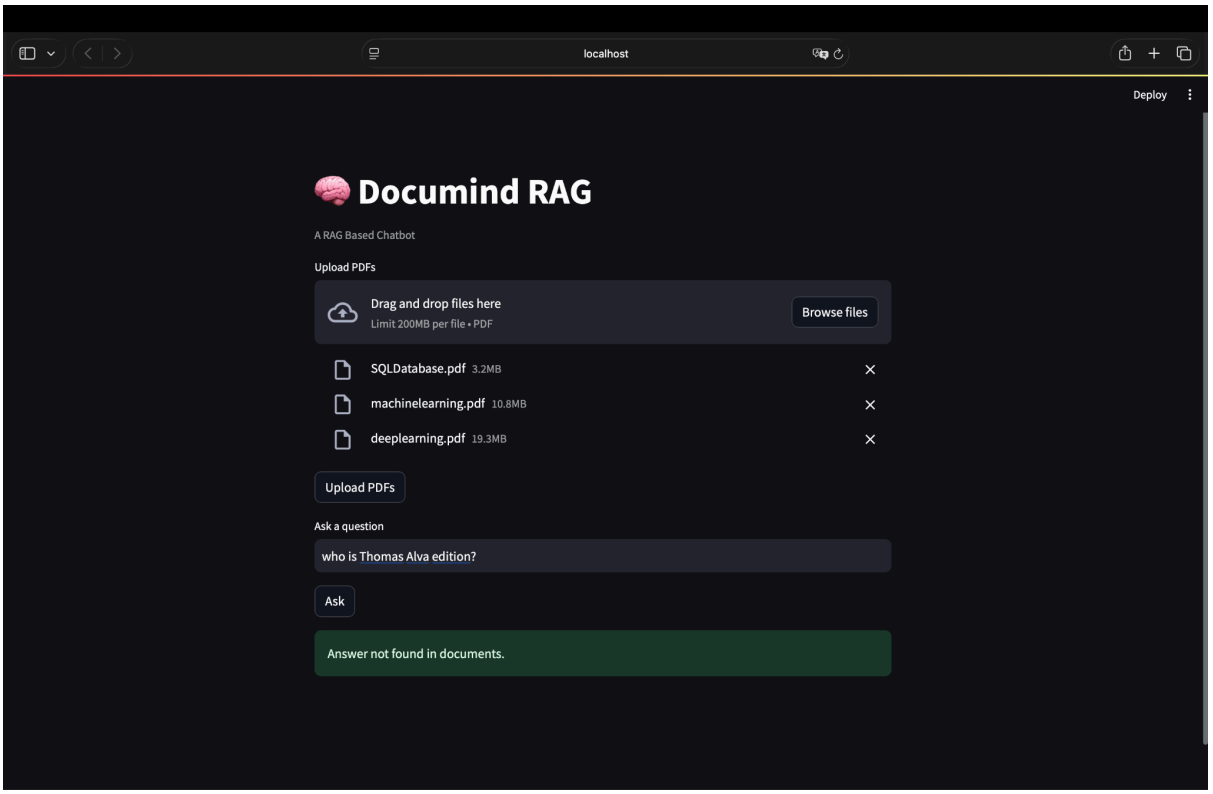


Fig.7. Fallback Response to the Query that is not in the documentation, Answer Not Found.

6. CURL DEMONSTRATION AND SDK

Documind RAG exposes REST APIs using **FastAPI**, making it easy to integrate with any frontend, backend, or automation tool.
The system can be accessed using **standard HTTP requests (cURL)** or **via a Python SDK-style integration**.

Steps to be followed : **Using Documind RAG via cURL**

1. Open the Terminal in local/ VS Code terminal and start the backend server with the command line

```
uvicorn app.main:app
```

2. After the starting of server, check the browser for confirmation using the address and it displays in the swagger,

```
http://127.0.0.1:8000/docs
```

3. In the terminal, Go to the Root Project Directory and upload the required documents using the cURL.

```
curl -X POST http://127.0.0.1:8000/upload \
  -F "files=@data/pdfs/deeplearning.pdf" \
  -F "files=@data/pdfs/machinelearning.pdf" \
  -F "files=@data/pdfs/statistics.pdf"
```

4. After the uploading, a message will be delivered.

```
{
"message": "PDFs uploaded and indexed successfully",
}
```

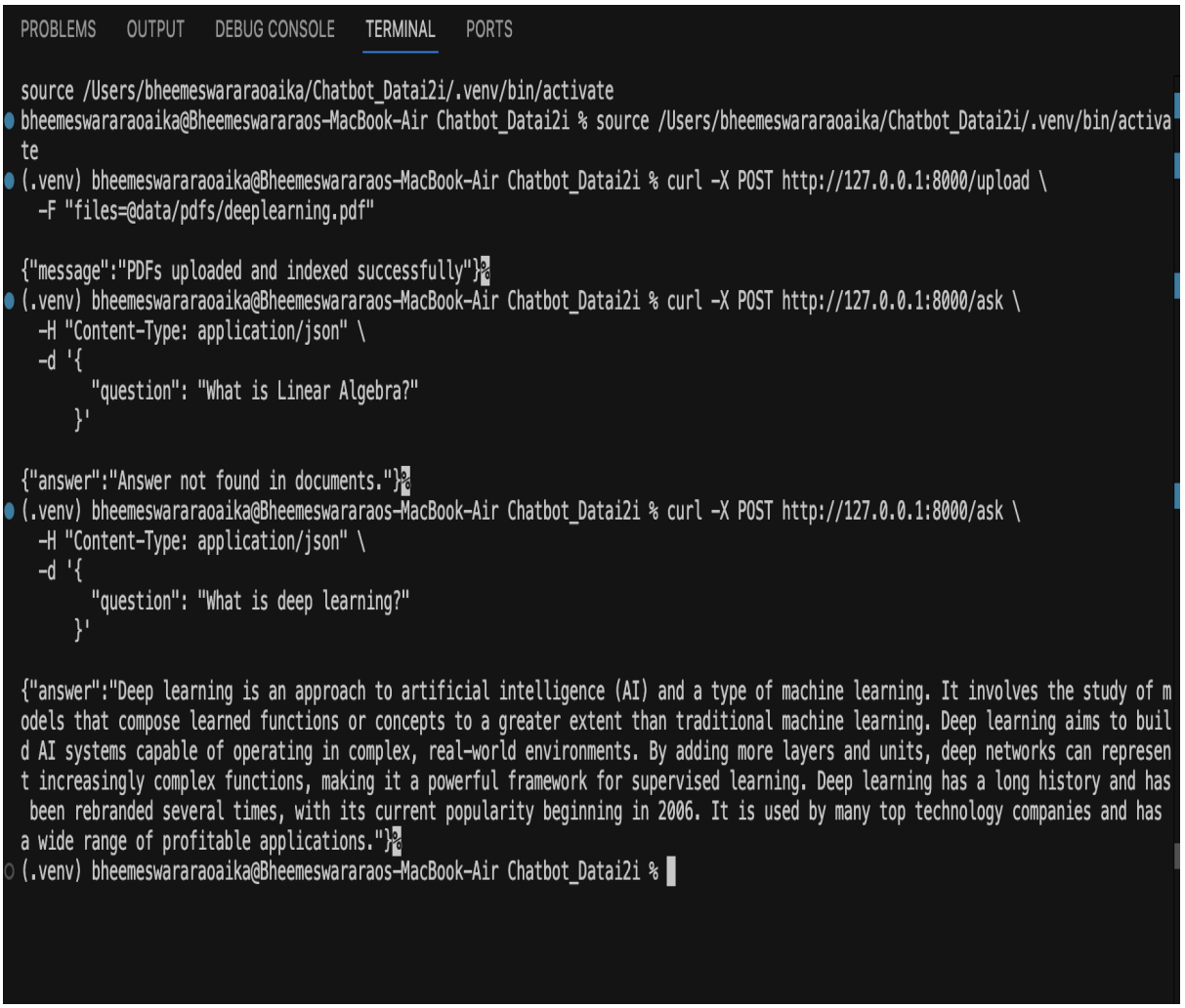
5. Ask a Question using the cURL

```
curl -X POST http://127.0.0.1:8000/ask \
 -H "Content-Type: application/json" \
-d '{
        "question": "What is deep learning?"
 }'
```

If the answer is found in the documents, it returns *the answer*, else it returns the answer *not found.*

EXECUTING DOCUMING RAG USING CURL IN VSCODE TERMINAL

```
source /Users/bheemeswararaoaika/Chatbot_Datai2i/.venv/bin/activate
bheemeswararaoaika@Bheemeswararaos-MacBook-Air   Chatbot_Datai2i   %
source /Users/bheemeswararaoaika/Chatbot_Datai2i/.venv/bin/activa
te

(.venv)bheemeswararaoaika@Bheemeswararaos-MacBook-Air
Chatbot_Datai2i % curl -X POST http://127.0.0.1:8000/upload \
  -F "files=@data/pdfs/deeplearning.pdf"

{"message":"PDFs uploaded and indexed successfully"}%

(.venv)bheemeswararaoaika@Bheemeswararaos-MacBook-Air
Chatbot_Datai2i % curl -X POST http://127.0.0.1:8000/ask \
  -H "Content-Type: application/json" \
  -d '{
        "question": "What is Linear Algebra?"
      }'

{"answer":"Answer not found in documents."}%

(.venv)bheemeswararaoaika@Bheemeswararaos-MacBook-Air
Chatbot_Datai2i % curl -X POST http://127.0.0.1:8000/ask \
  -H "Content-Type: application/json" \
  -d '{
        "question": "What is deep learning?"
      }'

{"answer":"Deep learning is an approach to artificial intelligence
(AI) and a type of machine learning. It involves the study of models
that compose learned functions or concepts to a greater extent than
traditional machine learning. Deep learning aims to build AI systems
capable of operating in complex, real-world environments. By adding
more layers and units, deep networks can represent increasingly
complex functions, making it a powerful framework for supervised
learning. Deep learning has a long history and has been rebranded
several times, with its current popularity beginning in 2006. It is
used by many top technology companies and has a wide range of
profitable applications."}%

(.venv)bheemeswararaoaika@Bheemeswararaos-MacBook-Air
Chatbot_Datai2i %
```



Visual Representation of DocuMind RAG using the cURL

IMPLEMENTATION USING SDK

SDK Represents the Software Development Kit, An SDK is an official library provided by a company that lets developers use their service easily inside code.

Instead of writing raw API calls (HTTP requests), the SDK:

- Handles authentication
- Sends requests correctly
- Parses responses
- Manages errors and retries

In this ChatBOT SDK is implemented by the following Files

1. Cohere SDK (**`cohere_llm.py`**)
   Here in this file we will use co.chat(), which is provided by the cohere SDK.
2. Embeddings SDK (**`vectorstore.py`**)
3. FAISS SDK
   Useful in the Vector search

SDK is mainly used in the chatbot for the following reasons likely to reduce complexity, improve reliability, and are best suited for large-scale document processing systems.

CHALLANGES FACED DURING THE PROJECT

1. Handling Large PDF without Memory issues
2. Missing Answers even the answer present in the PDF
3. Ensuring the Model does not Hallucinate Answers
4. Choosing the right database for the large data
5. Handling Exact facts like Emails, Phone numbers
6. Model compatibility and API Failures
7. UI and Backend Sync Issues
8. Ensuring Multi-language PDF Support
9. Version Support issues
10. Improper Storing and Chunking Issues

Handling Large PDF Without Memory Issues

Users can upload **very large PDFs**, Loading the entire file into memory can lead to Slow Processing, High Memory Usage and the system can be Crashed.

Solution: The small and large PDF's are processed page to page and Each page is treated as a separate Document Unit and the files are saved to disk first to avoid memory overloading. By this process the chunking will be normal and the data will be accurate while generating the answer.

Missing Answers even the answer present in the PDF

Initially, chatbot responded with "Answer not found in documents" even for a straight forward question and information is clearly present mainly because large pages were not split properly and context was lost and semantic search missed exact values like email and phone numbers.

Solution: to overcome this implemented **Overlapping text chunking, used smaller chunks with overlap** and added Hybrid Retrieval like Semantic search for meaning and **keyword search** for exact matches.

Now it can retrieve bullet points, IDS and phone numbers and facts easily and accurately.

Ensuring the Model does not Hallucinate Answers

The models can generate answers not present in documents, which is unacceptable for document-based systems.

Solution: Enforced the Strict prompt rules like **Answer only from the provided context and Return fall back message if data is missing.** No external knowledge is allowed.

Choosing the Right Vector Database for Large Data

Initially with the chroma DB struggled with performance on large datasets and Re-Indexing flexibility and Scalability.

Solution: Switched to FAISS, used the local FAISS Indexing and enabled the batch-based indexing. Now it can give the **Fast retrieval, scales to thousands of pages and Production grade performance.**

Handling Exact Facts Like Emails, Phone Numbers, IDs

Semantic search alone failed to retrieve the Emailid, Mobile numbers and short factual values.

Solution: Added a **Keyword-based indexing and Used Regex and exact matching for factual queries,** combined the results with the semantic retrieval.

Model Compatibility and API Failures

Some models are deprecated and API Quota exhausted. Moreover, Unsupported models caused runtime failures.

Solution: Designed the coding for model testing test, and wrote a code to know the supported models based on the API key and verified models with test code before using in the main code. Added isolated API tests to separate data issues from the API issues.

Now it is stable model selection and easier debugging.

UI and Backend Synchronization Issues

The UI which is built by using streamlit shows success even when backend operation fails, likely uploading and indexing of pdf's and mainly JSON Parsing errors in streamlit. Improper response formats.

Solution: to Overcome this i have standardized API response structure and added Error Handling in the UI, also displayed the indexing status and pagecount which is connected with the backend for a proper accurate information.

Ensuring Multi Language PDF Support

A User may upload a different language pdf and ask questions on it and embedding needs to work universally.

Solution: To make easier i have used Multilingual sentence transformer embeddings which avoid language specific tokenizers and allows the chatbot to support multi-language PDF's

Version Support issues

The langchain framework does not support the latest python versions.

Solution: to make use of them reinstalling the python supported versions like python3.11.9.

Improper Storing and Chunking Issues

The text from the pdf is not stored properly in the database and the answer was not proper and inaccurate.

Solution: To overcome the issue I have imported libraries like pypdf and enabled the two level chunking for accurate information. To make sure it's working I have created the files and tested with sample pdf’s using inheritance importing the main code to the test file before finalising .

Now it's working properly with the accurate retrievals.