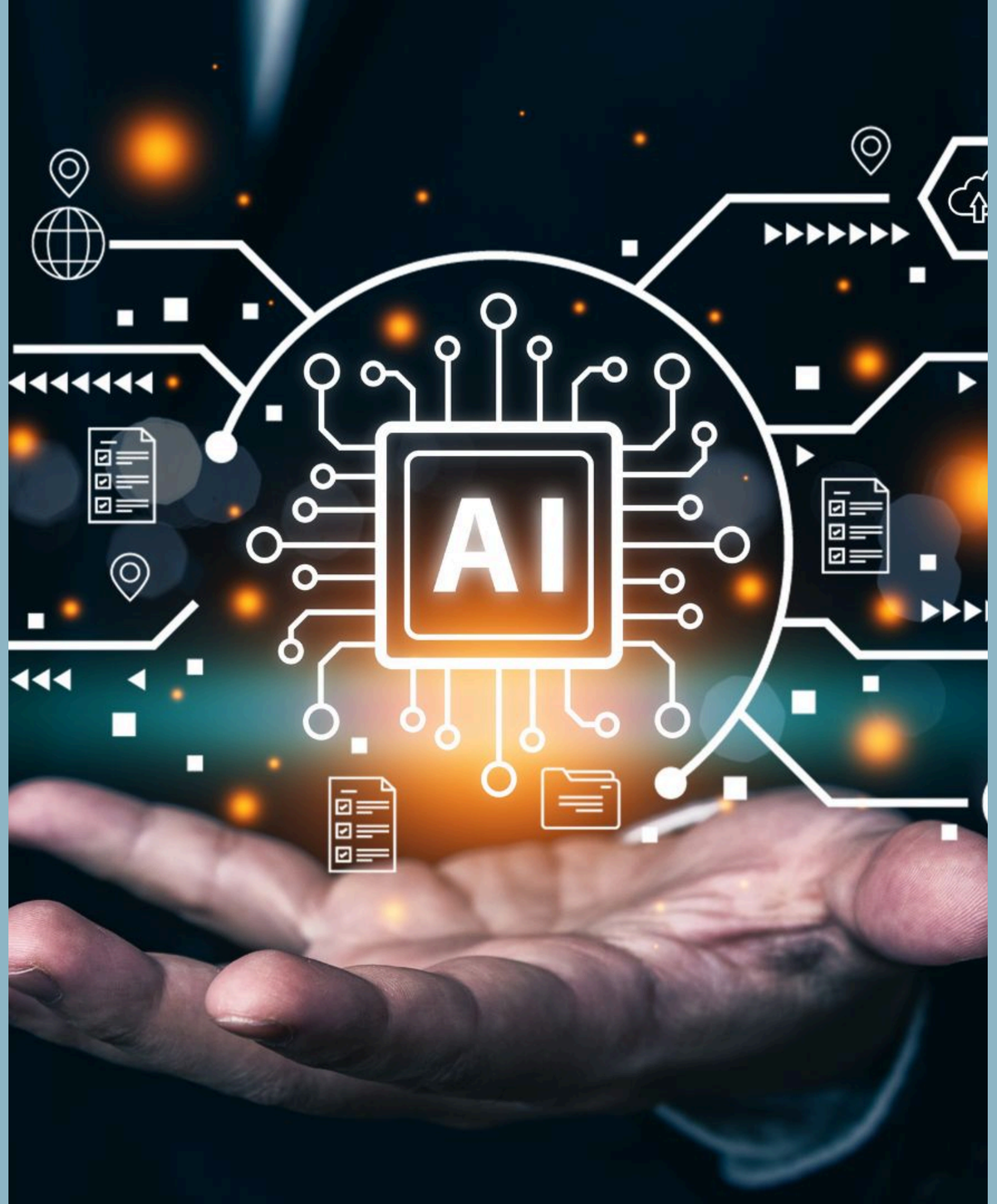
2025

# Project

## CHAT WITH MULTIPLE PDFS USING LLM



# TABLE OF CONTENTS

# Team

CARRIED OUT BY
HIBA EZZAHI

CARRIED OUT BY
NIAMA FTASSI

CARRIED OUT BY
ABDELHAKIM HANI

CARRIED OUT BY
HAMZA EZZAHAOUI

# Team

SUPERVISED BY
HAKIM HAFIDI

SUPERVISED BY
YASSER ADERGHAL

01

# Introduction

## Overview

The PDF Chat Application allows users to upload PDF documents and interact with the content using natural language. The application processes PDF files, converts them to text, divides the text into chunks, and creates embeddings for each chunk. Using a vector store for efficient retrieval, the system leverages Google's Gemini AI model to generate answers to user queries based on the content of the uploaded document. This report outlines the technical details, components, and functionality of the application.


01

# *Introduction*

## Project Objectives

- User Interaction: Provide an interface for users to upload PDF documents and ask questions about the document’s content.
- Document Processing: Extract text from the uploaded PDFs and split it into manageable chunks for easier querying.
- Efficient Retrieval: Use embeddings and a vector search algorithm to allow for fast retrieval of relevant document chunks based on user queries.
- AI Integration: Use Google's Gemini AI model to generate relevant responses to user queries by analyzing the retrieved document chunks.


# 02 Key Components

## I. Dependencies

The application utilizes several key libraries and tools:

- Streamlit: A web framework used to build the interactive user interface.
- PyPDF2: A library for reading and extracting text from PDF files.
- Langchain: A framework for interacting with large language models, which helps in processing text and creating embeddings.
- FAISS: A library used to perform efficient similarity searches for document chunk retrieval.
- google.generativeai: The API used for generating AI-based responses to user queries using Google's Gemini AI model.

02

# *Key Components*

*2. Technologies Used*

- Streamlit for UI creation.
- PyPDF2 for PDF processing and text extraction.
- Langchain for working with LLMs and generating embeddings.
- FAISS for vector similarity search and storing embeddings.
- Google Gemini AI for generating relevant responses to queries.

# 03 System Workflow

## User Uploads PDF Documents

- The user uploads a PDF file via the Streamlit web interface.
- The file is processed by PyPDF2, which extracts the text from the document

## Text Processing and Chunking

- After the text is extracted from the PDF, it is split into smaller chunks. This ensures that the text is manageable and that the language model can better interpret it.
- This chunking process is performed using Langchain or custom splitting functions, optimizing the text for AI processing.

# 03 System Workflow

## Embedding Creation

- Each text chunk is converted into an embedding using an embedding model (through Langchain).
- These embeddings are stored in a FAISS vector database, which allows for fast similarity searches.

## User Queries the Document

- Users input their queries in natural language via the web interface.
- The query is converted into an embedding and compared against the document embeddings stored in FAISS to retrieve the most relevant text chunks.

# 03 System Workflow

## AI Model Generates Responses

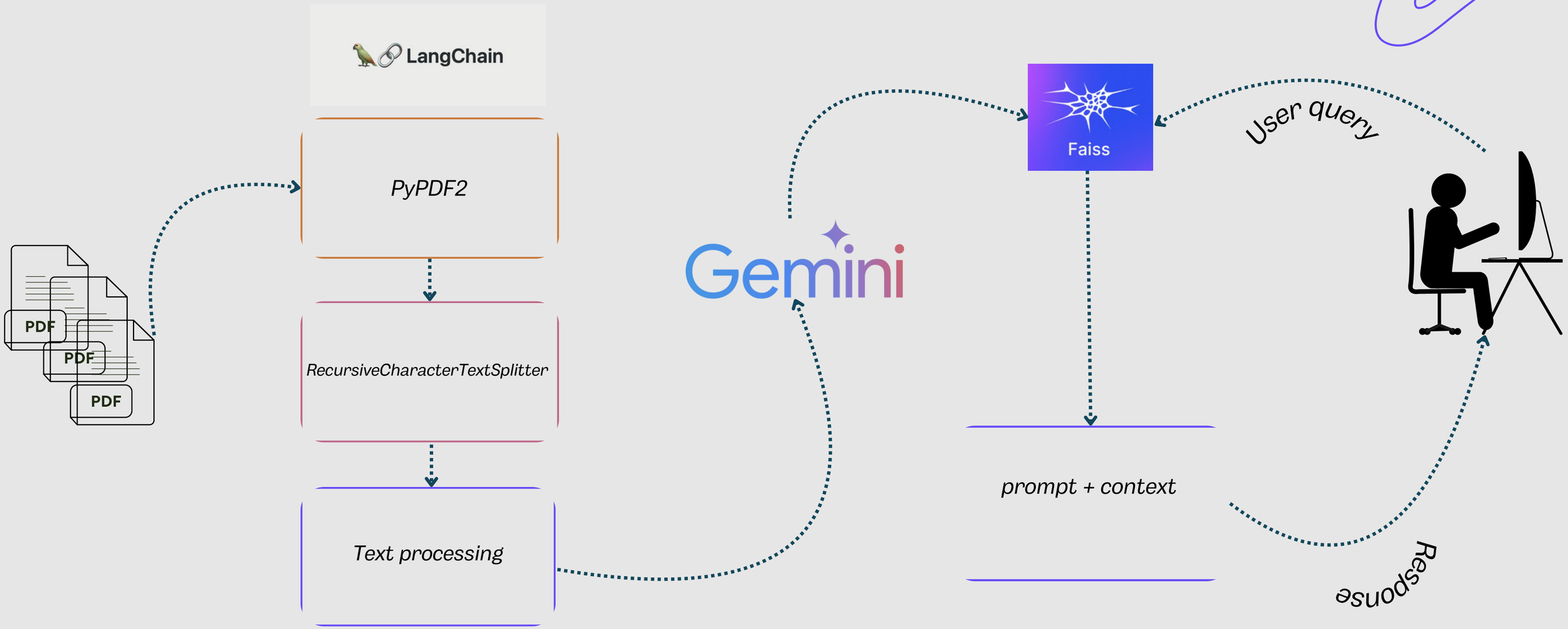
- The relevant chunks retrieved from FAISS are passed to Google Gemini AI to generate responses.
- The AI model processes the text, interprets the context, and produces an answer that is relevant to the user's query.

## Displaying Responses

- The generated response is displayed in the main interface area of the Streamlit app, providing the user with an answer based on the uploaded PDF document's content

04

# PROJECT ARCHITECTURE



# 05 Performance and Optimization

## Text Chunking

The chunking process ensures that the document’s content is divided into manageable segments, improving the AI’s ability to process and answer queries effectively.

## Vector Store (FAISS)

FAISS allows for efficient retrieval of document chunks using vector similarity search, ensuring that the system can scale with large documents and multiple queries.

## Embedding Storage

The chunking process ensures that the document’s content is divided into manageable segments, improving the AI’s ability to process and answer queries effectively.

06

# PDF Chat Application - Step by Step Implementation Guide

## Step I: Initial Setup and Imports

Environment Setup :

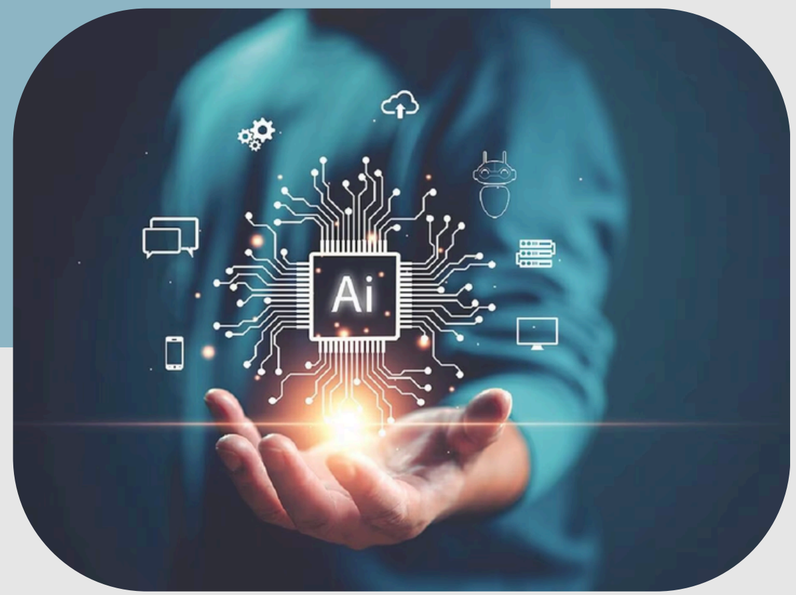
1.Create a .env file in your project directory

2. Add your Google API key:

```
GOOGLE_API_KEY=your_api_key_here
```

3. Install required packages :

```
install streamlit PyPDF2 langchain langchain-google-genai faiss-cpu python-dotenv google-generativeai
```

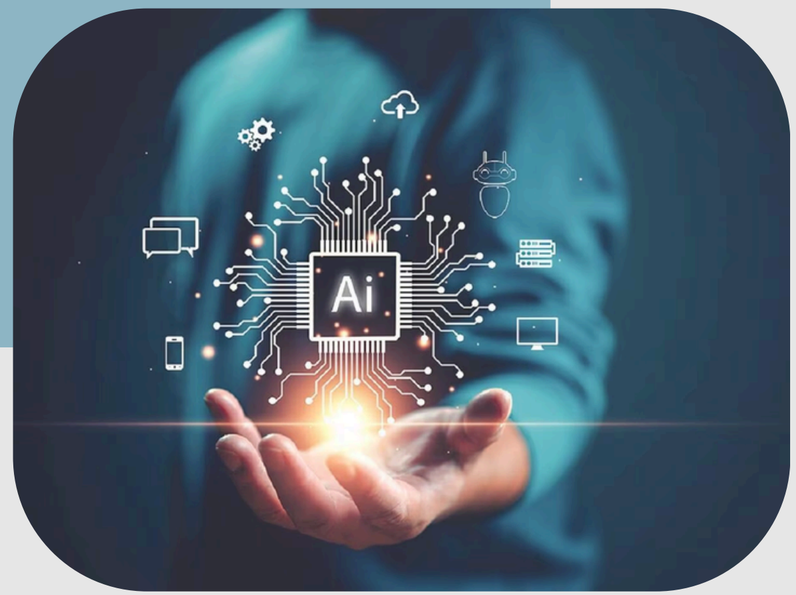


06

# PDF Chat Application - Step by Step Implementation Guide

*Import Configuration :*

```
# Load environment variables
load_dotenv()
os.getenv("GOOGLE_API_KEY")
genai.configure(api_key=os.getenv("GOOGLE_API_KEY"))
```




## Step 2: PDF Text Extraction

Function : `get_pdf_text(pdf_docs)`

Process:

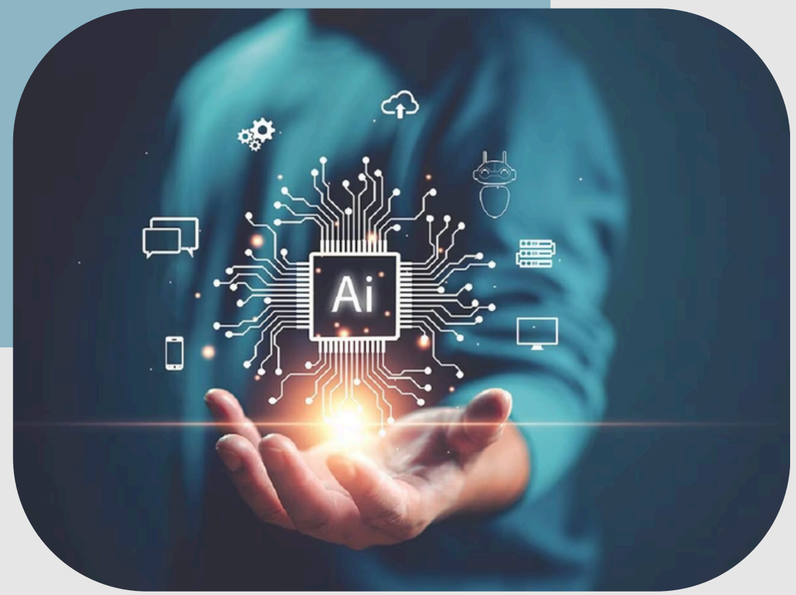
1. Initialize empty string for text storage 2. For each uploaded PDF:

- Create PdfReader object
- Iterate through pages
- Extract text from each page
- Concatenate to main text string

06

# PDF Chat Application - Step by Step Implementation Guide

```
def get_pdf_text(pdf_docs):
    text=""
    for pdf in pdf_docs:
        pdf_reader= PdfReader(pdf)
        for page in pdf_reader.pages:
            text+= page.extract_text()
    return text
```




## Step 3: Text Chunking

Function : `get_text_chunks(text)`

1. Create RecursiveCharacterTextSplitter with:
   - chunk_size = 10000 (characters per chunk)
   - chunk_overlap = 1000 (overlap between chunks)
2. Split text into chunks
3. Return chunk list

Purpose: Makes large texts manageable and maintains context between chunks

06

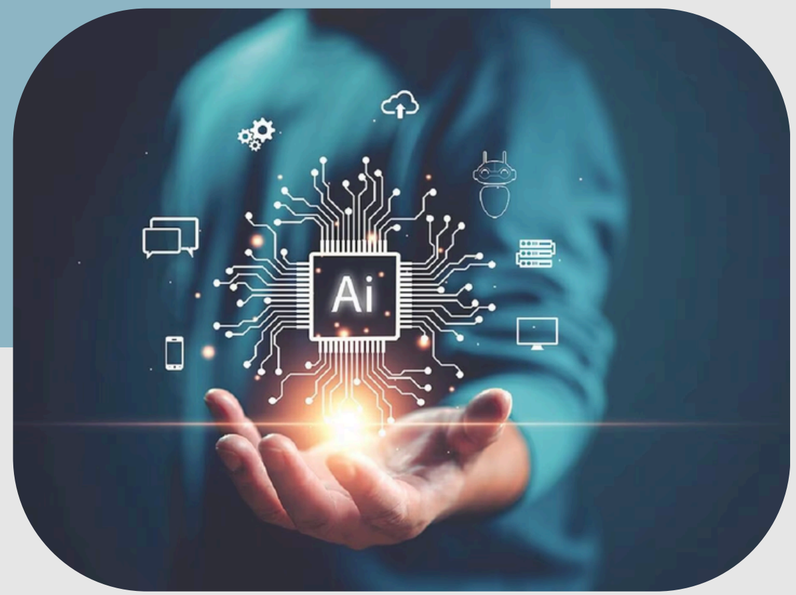
# PDF Chat Application - Step by Step Implementation Guide

## Step 4: Vector Store Creation

Function : `get_vector_store(text_chunks)`

Process :

1. Initialize Google's embedding model
2. Create FAISS vector store:
   - Convert text chunks to embeddings
   - Store embeddings in FAISS index
3. Save index locally for future use Purpose:
   - Creates searchable database of text embeddings



06

# PDF Chat Application - Step by Step Implementation Guide

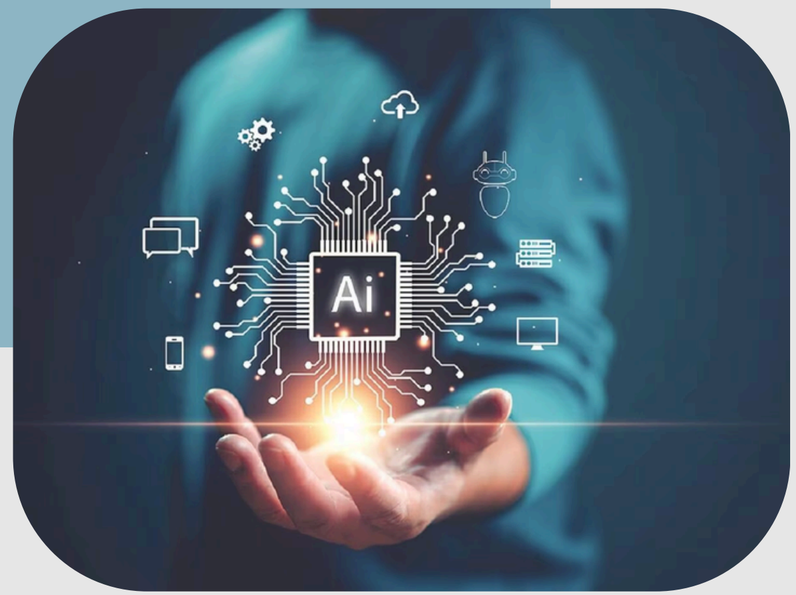
## Step 5: Conversation Chain Setup

Function : `get_conversational_chain()`

Process :

1. Define prompt template with:
   - Context placeholder
   - Question placeholder
   - Instructions for answering
2. Initialize Gemini Pro model
3. Create QA chain with:
   - Model configuration
   - Chain type: "stuff"
   - Custom prompt

Purpose: Sets up the question-answering pipeline



06

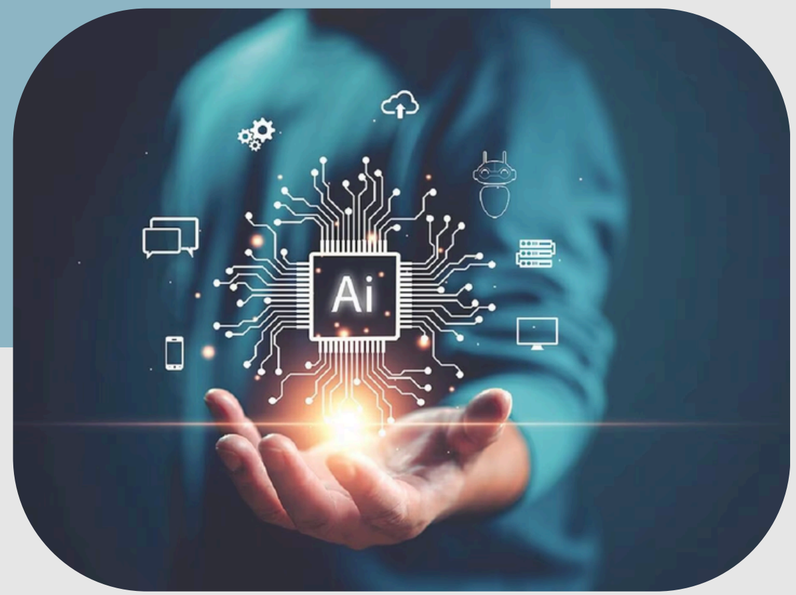
# PDF Chat Application - Step by Step Implementation Guide

## Step 6: User Input Processing

Function : `user_input(user_question)`

Process :

1. Load saved FAISS index
2. Perform similarity search for question
3. Get relevant document chunks
4. Run question through conversation chain
5. Display response in Streamlit interface



06

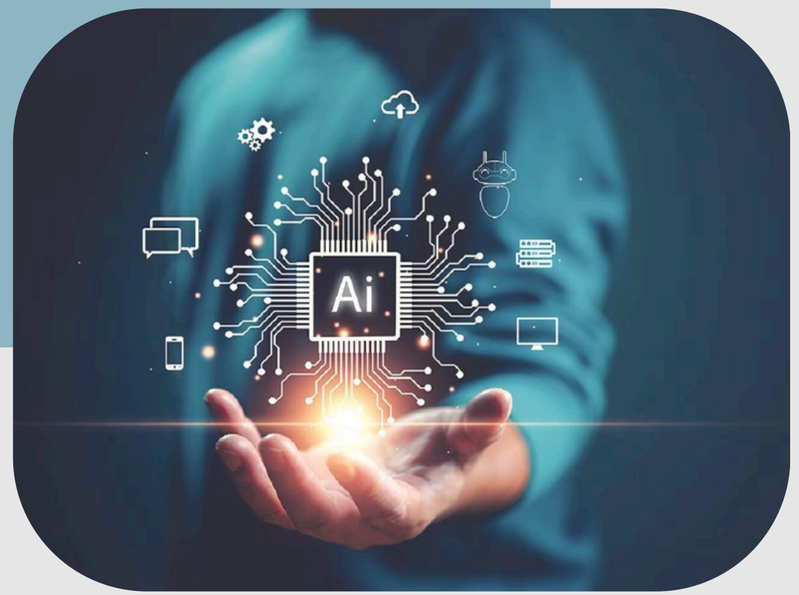
# PDF Chat Application - Step by Step Implementation Guide

## Step 7: Main Application Interface

*Function :* `main()`

*Process :*

1. Configure Streamlit page:
   - Set title
   - Create header
2. Create main interface:
   - Question input field
   - Response display area
3. Create sidebar:
   - PDF upload interface
   - Processing button
4. Handle document processing:
   - Extract text
   - Create chunks
   - Build vector store



06

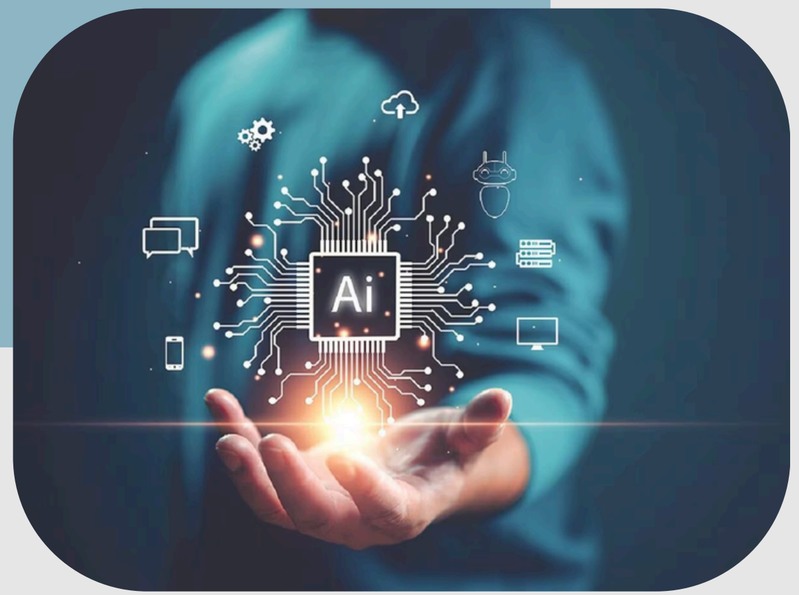
# PDF Chat Application - Step by Step Implementation Guide

## File Structure

```
your_app_directory/
├── .env                    # API key
├── app.py                  # Main application file
├── faiss_index/            # Generated vector store
└── requirements.txt        # Dependencies
```

## Running the Application

1. Save the code as app.py
2. Ensure all dependencies are installed
3. Run with command: `streamlit run app.py`



06

# PDF Chat Application - Step by Step Implementation Guide

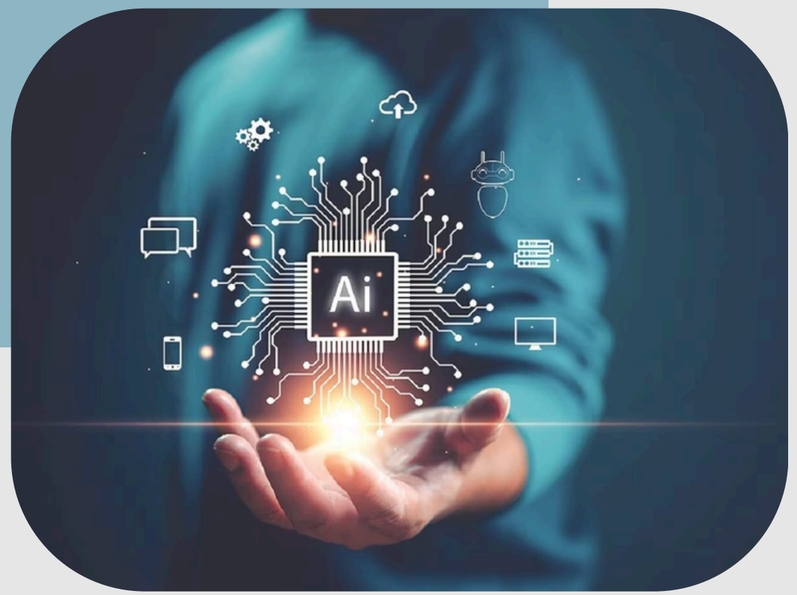
## Usage Flow

*I. User Interface:*

- *Access web interface*
- *Upload PDF(s) via sidebar*
- *Click "Submit & Process"*
- *Wait for processing completion*
- *Ask questions in main interface*

*2. Backend Process:*

- *PDFs processed to text*
- *Text split into chunks*
- *Chunks converted to embeddings*
- *Embeddings stored in FAISS*
- *Questions matched to relevant chunks*
- *AI generates answers from chunk*



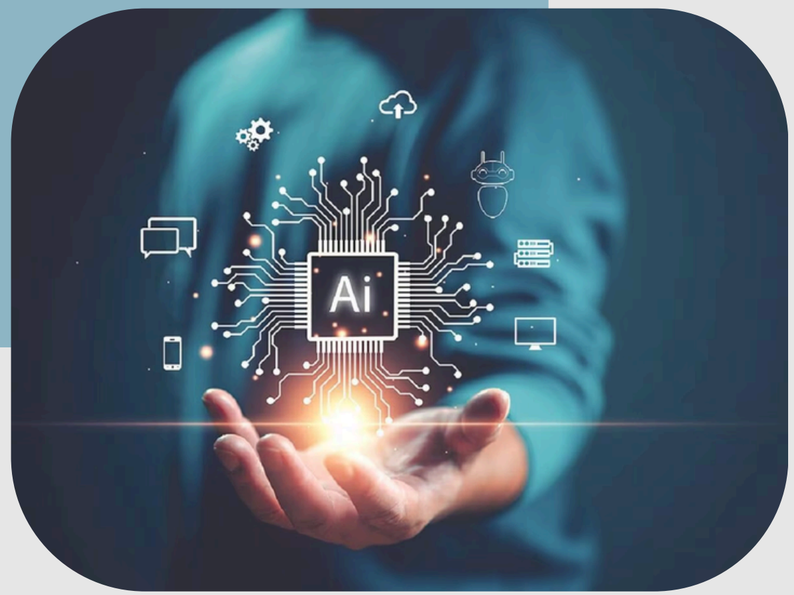
06

# PDF Chat Application - Step by Step Implementation Guide

## Error Handling

The application includes several safety measures:

- Checks for API key presence
- Safe deserialization of FAISS index
- Processing status indicators
- Clear success/failure messages



06

# PDF Chat Application - Step by Step Implementation Guide

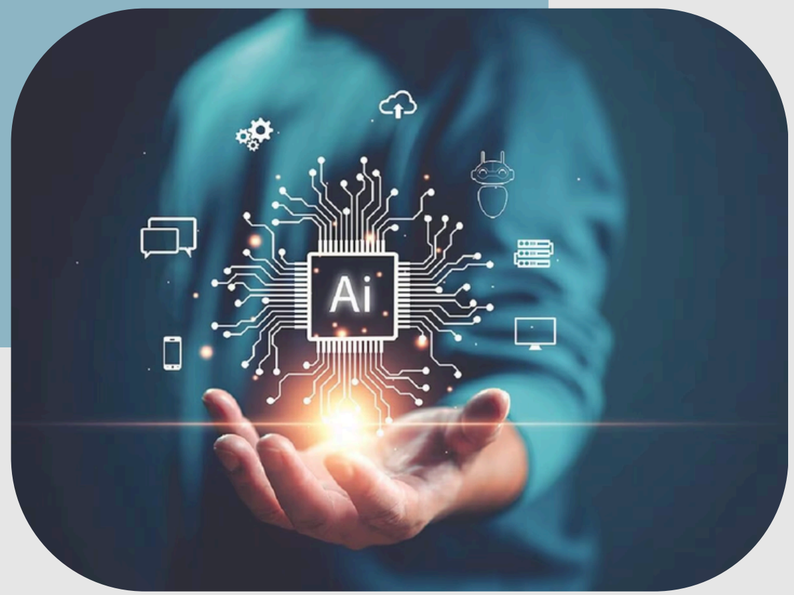
## Performance Considerations

*I. Text Processing:*

- *Efficient chunk size for balance*
- *Overlap for context preservation*
- *Local storage for speed*

*2. Model Configuration:*

- *Temperature: 0.3 for focused responses*
- *Embedding model optimized for performanc*



06

# *Conclusion*

The PDF Chat Application provides an innovative and user-friendly interface for querying and interacting with PDF documents using natural language. By combining efficient document processing, embedding creation, and leveraging Google’s Gemini AI, the system ensures that users can obtain relevant and accurate information from PDFs. With ongoing optimizations, the application can be further enhanced to handle larger documents and more complex queries, making it a valuable tool for document analysis and knowledge retrieval.

# Thank you for your attention

*any questions ?*