BUCKMAN LABORATORIES

DATA SCIENCE INTERN ROLE

TASK-BASED SELECTION ROUND

Retrieval-Augmented Generation (RAG) Chatbot Documentation

-HARINNE R KUMAR

Harinnerkumar2006@gmail.com

Introduction

In an era where information is scattered across countless PDFs, Word documents, spreadsheets, images, and videos, the ability to quickly extract and query relevant data is a game-changer. The Retrieval-Augmented Generation (RAG) Chatbot is a web-based tool designed to empower users to converse with their documents effortlessly. Built with Streamlit, it processes diverse file types—think legal contracts (PDFs), meeting notes (DOCX), sales data (CSV), scanned receipts (images), or training videos—and answers questions using a blend of retrieval and generative AI. This project turns static files into a dynamic, interactive resource, addressing real-world needs across industries.

Consider a small business owner needing to analyze monthly sales trends from CSV reports, a healthcare worker retrieving patient care instructions from video tutorials, or a student pulling key concepts from a stack of lecture PDFs. The RAG Chatbot eliminates the tedious manual search, delivering precise answers in seconds. By leveraging cutting-edge AI, it bridges the gap between document overload and actionable insights, making it an indispensable tool for professionals, educators, and everyday users alike.

Problem Statement

In real-world settings, accessing information locked within diverse document formats is a persistent struggle:

- **Small Businesses**: Entrepreneurs often juggle CSV sales records, PDF invoices, and DOCX supplier agreements. Finding answers like “Which product sold best in Q3?” or “What’s the payment term with Vendor X?” requires opening multiple files and sifting through rows or pages—a slow, error-prone process.
- **Healthcare**: Nurses and doctors need quick access to protocols buried in training videos or scanned handwritten notes (images). Without a way to query “What’s the dosage for Drug Y?” directly, critical time is lost converting multimedia into usable text.
- **Education**: Students and professors face stacks of PDFs from research papers or lecture slides, plus hours of recorded video classes. Extracting answers like “What did the professor say about photosynthesis?” demands hours of review, disrupting study efficiency.
- **Personal Use**: Individuals archiving receipts (images), user manuals (PDFs), or DIY video guides struggle to recall specifics like “What’s covered under my appliance warranty?” without revisiting each source manually.

These scenarios reveal core challenges: **fragmented file formats**, **unstructured multimedia data**, and **lack of conversational access**. Traditional tools—PDF readers, video players, or spreadsheets—operate in silos, forcing users to switch contexts or rely on memory. The RAG

Chatbot solves this by unifying document processing, text extraction, and natural language querying into one platform, streamlining workflows and unlocking real-time insights from real-world data.

Objectives

1. **Broad File Compatibility**: Support the processing and extraction of content from a wide range of file formats, including documents (e.g., PDFs, DOCX), tabular data (e.g., CSV), and multimedia (e.g., images, videos).
2. **Effective Content Extraction**: Accurately retrieve text from both structured sources (e.g., spreadsheets) and unstructured sources (e.g., images or video frames) using appropriate techniques.
3. **Efficient Data Storage**: Organize extracted content in a searchable format, such as a vector-based system, to enable quick and reliable retrieval.
4. **Natural Language Interaction**: Provide a conversational system that answers user queries by combining retrieved content with generative capabilities, ensuring relevant and coherent responses.
5. **User-Friendly Experience**: Offer an accessible and intuitive interface for uploading files and interacting with the system, suitable for diverse users.
6. **Performance and Scalability**: Ensure the system can process multiple files efficiently and deliver responses in real-time, adapting to varying workloads and use cases.

Approach 1: Streamlit with LangChain

Technical Stack

- **Programming Language**: Python
  - Core language for scripting, integrating libraries, and executing logic.
- **Web Framework**: Streamlit (streamlit)
  - A Python library that simplifies creating interactive web apps, rendering a browser-based UI for file uploads and chat with minimal HTML/CSS knowledge required.
- **AI/ML Libraries**:
  - google.generativeai: Provides an interface to Google’s Gemini 1.5 Pro model, a cloud-based LLM for generating query responses.
  - langchain: A framework for building RAG systems, handling text splitting (RecursiveCharacterTextSplitter), embeddings (HuggingFaceEmbeddings),

vector storage (Chroma), and QA chains (RetrievalQA, ChatGoogleGenerativeAI).

    - sentence-transformers: Supplies the all-MiniLM-L6-v2 model, a lightweight transformer for generating dense text embeddings.
    - chroma: An open-source vector database that stores embeddings and retrieves similar chunks using cosine similarity.

- **Document Processing**:
    - PyPDF2: A library for reading PDF files, extracting text from each page.
    - unstructured: Processes DOCX files via UnstructuredWordDocumentLoader, converting structured content to plain text.
    - pandas: Converts CSV data into a string representation using to_string() for text processing.
    - json: Parses JSON files into a string format for further splitting.
- **Image/Video Processing**:
    - Pillow (PIL): Opens images, converts them to RGB format for OCR compatibility.
    - opencv-python (cv2): Extracts frames from videos, enabling text detection from visual content.
    - easyocr: Performs optical character recognition (OCR) on images and video frames, extracting English text with confidence scores.
- **Utilities**:
    - python-dotenv: Loads environment variables (e.g., GOOGLE_API_KEY) from a .env file for secure configuration.
    - python-magic: Identifies file MIME types to determine appropriate processing logic.
    - numpy: Facilitates array operations, crucial for image and video frame handling.
    - tempfile: Creates temporary files to store uploaded content during processing.
- **External Tools**: Poppler (optional)
    - A PDF rendering library, checked via check_poppler_installation for compatibility with some PDF operations.

Workflow :

1. **Initialization**:
    - Load environment variables using load_dotenv() to securely access GOOGLE_API_KEY.
    - Configure Google API with genai.configure to enable Gemini model access.
    - Initialize embeddings with HuggingFaceEmbeddings using the all-MiniLM-L6-v2 model for text vectorization.
    - Set up RecursiveCharacterTextSplitter with 1000-character chunks and 200-character overlap to break text into manageable pieces.
    - Create a Chroma vector store at the chroma_db directory to persist embeddings.
    - Initialize the EasyOCR reader for English (reader = easyocr.Reader(['en'])) to handle image/video text extraction.
2. **File Upload and Processing**:
    - **UI Setup**: Streamlit configures a wide-layout page titled "RAG Chatbot" with a file uploader supporting multiple files (PDF, DOCX, CSV, JSON, JPG, PNG, MP4, AVI).
    - **File Loop**: For each uploaded file:
        - Save the file to a temporary location using tempfile.NamedTemporaryFile.
        - Detect the MIME type with magic.from_file to route to the correct processor.
        - Process based on type:
            - **PDF**: process_pdf_file opens the file with PyPDF2, extracts text from all pages, and splits it with text_splitter.
            - **Image**: process_image_file loads with PIL, converts to RGB, applies EasyOCR (confidence > 0.5), and splits extracted text.
            - **Video**: process_video_file uses OpenCV to sample every 5th frame (up to 100), applies EasyOCR, and splits text.
            - **CSV**: pandas.read_csv converts to a string, split by text_splitter.
            - **DOCX**: UnstructuredWordDocumentLoader loads the file, splits the resulting text.
            - **JSON**: json.load converts to a string, split by text_splitter.
        - Add non-empty chunks to Chroma using vectordb.add_texts.

        - Display feedback: st.success for success, st.warning for no text, st.error for failures.
    - Clean up by deleting temporary files with os.unlink.

3. **Chat Interaction**:
    - **Chat Setup**: Initialize st.session_state.messages if absent to store chat history, display prior messages with st.chat_message.
    - **User Input**: Capture the query via st.chat_input (e.g., “What’s in this document?”).
    - **Processing**:
        - Append query to history and display with st.markdown.
        - Create a QA chain with get_qa_chain(): combines ChatGoogleGenerativeAI (Gemini) with a Chroma retriever (top-3 chunks).
        - Run the query through qa_chain.run to retrieve and generate a response.
    - **Output**: Display the response with st.markdown, append to history.
4. **Cleanup**: Temporary files are removed post-processing; Chroma persists embeddings for future sessions.

Approach 2: Command-Line with FAISS and Ollama

Technical Stack

- **Programming Language**: Python
    - Foundation for all logic, file handling, and library integration.
- **AI/ML Libraries**:
    - sentence-transformers: Provides all-MiniLM-L6-v2, a transformer model for embedding text chunks into dense vectors.
    - faiss: A lightweight library for efficient similarity search, using a FlatL2 index to store and retrieve embeddings.
    - ollama: Interfaces with the llava:7b model (assumed local) for image description and query answering, supporting text generation.
- **Document Processing**:
    - PyPDF2: Extracts text from PDFs by concatenating page content.

- docx (python-docx): Parses DOCX files, joining paragraphs into a single text string.
- pandas: Converts CSV data to a text string using to_string().

- **Image/Video Processing**:
  - moviepy: Extracts audio from videos (saved as WAV) and saves frames for analysis.
  - speech_recognition (sr): Transcribes video audio using Google Speech Recognition API.
  - ollama (llava:7b): Describes image content and video frame visuals based on binary input.
- **Utilities**:
  - os: Manages file paths, existence checks, and temporary file deletions.
  - numpy: Handles array operations, essential for embedding calculations.

Workflow :

1. **Initialization**:
   - No explicit setup phase; libraries are imported and used as needed in main().
   - Suppress warnings with warnings.filterwarnings("ignore") to streamline terminal output.
2. **File Upload and Processing**:
   - **User Prompt**: Terminal prompts users to enter full file paths (e.g., D:\file.pdf) or type done to finish.
   - **File Loop**: For each input path:
     - Verify file existence with os.path.exists; skip if invalid.
     - Extract text based on extension:
       - **PDF**: extract_text_from_pdf uses PyPDF2 to concatenate page text.
       - **DOCX**: extract_text_from_docx joins paragraphs with python-docx.
       - **CSV**: extract_text_from_csv converts pandas DataFrame to string.
       - **Image (JPG/PNG)**: extract_text_from_image reads binary data, sends to Ollama’s llava:7b for description.

- **Video (MP4/AVI)**: extract_text_from_video:
    - Extracts audio with MoviePy, saves as temp_audio.wav.
    - Transcribes audio with SpeechRecognition (Google API).
    - Saves a frame at 1.0s as temp_frame.jpg, describes with Ollama.
    - Combines audio and visual text.
    - Print extracted text to terminal and save to processed_data.txt with metadata (filename).
    - Store in documents list with text and source.
- **Chunking**: Split each document's text into 500-word chunks, store with metadata (source filename).

3. **Retrieval System Setup**:
    - Embed chunks with SentenceTransformer (all-MiniLM-L6-v2), generating dense vectors.
    - Build a FAISS FlatL2 index with embedding dimensions, add embeddings for similarity search.
4. **Chat Interaction**:
    - **Prompt**: Display "Chatbot is ready!" and accept queries until exit.
    - **Processing**:
        - Embed query with embedder.encode.
        - Search FAISS index for top-3 similar chunks using index.search.
        - Retrieve chunks and metadata with retrieve_chunks.
        - Combine chunks into context, prompt llava:7b with generate_answer.
    - **Output**: Print the answer and source filenames to the terminal.
5. **Cleanup**: Remove temporary files (temp_audio.wav, temp_frame.jpg) after video processing.

Approach 3: Flask with Ollama

Technical Stack

- **Programming Language**: Python
    - Core language for scripting, file operations, and library integration.

- **Web Framework**: Flask (flask)
  - A lightweight web framework that serves an HTML frontend (index.html) and handles API requests (/api/chat) for file uploads and queries, enabling a browser-based interface.
- **Networking**: Requests (requests)
  - Facilitates HTTP POST requests to Ollama's API (http://localhost:11434/api/generate) for image/video processing and text generation.
- **AI/ML Libraries**:
  - ollama (via API): Uses the llava:7b model (assumed running locally at port 11434) for image description, video frame analysis, and query answering, supporting text generation and base64-encoded image input.
- **Document Processing**:
  - PyPDF2: Extracts text from PDFs by reading page content.
  - docx (python-docx): Parses DOCX files, concatenating paragraphs into text.
  - pandas: Converts CSV and XLSX files to text strings using to_string(), with openpyxl as the XLSX engine.
- **Image/Video Processing**:
  - moviepy: Extracts video audio (saved as WAV) and captures frames at 1-second intervals (up to 20 frames).
  - speech_recognition (sr): Transcribes video audio using Google Speech Recognition API.
  - base64: Encodes image and video frame data into base64 strings for Ollama API calls.
- **Utilities**:
  - os: Manages file operations (e.g., saving/removing temporary files like temp_video.mp4).
  - io: Handles in-memory file streams (e.g., resetting pointers with file.seek(0)).
  - collections.deque: Maintains a conversation history with a maximum of 3 entries for context.
  - json: Parses Ollama API responses into usable data.

Workflow

1. **Initialization**:
    - Create a Flask app (app = Flask(__name__)) with routes for the homepage (/) and chat API (/api/chat).
    - Define global variables:

- OLLAMA_API_URL: Set to http://localhost:11434/api/generate for Ollama's API.
- conversation_history: A deque with maxlen=3 to store recent user queries and bot responses.
- last_image_context: Tracks the latest image-based response for follow-up query context.

2. **File Upload and Processing**:
    - **Homepage**: The / route renders index.html, a frontend with a form for file uploads and chat input, sending POST requests to /api/chat.
    - **API Request**: The /api/chat route (POST) receives:
        - query: User's text question from request.form.get("query").
        - file: Uploaded file from request.files.get("file").
    - **File Handling**: If a file is uploaded:
        - Reset file pointer with file.seek(0) to read from the beginning.
        - Detect file type by extension:
            - **PDF**: extract_text_from_pdf uses PyPDF2 to concatenate page text.
            - **DOCX**: extract_text_from_docx joins paragraphs with python-docx.
            - **CSV**: extract_text_from_csv converts pandas DataFrame to text.
            - **XLSX**: extract_text_from_xlsx uses pandas with openpyxl to convert to text.
            - **Image (JPG/JPEG/PNG)**: Read as binary, encode to base64 (image_base64), set file_content to "Image uploaded."
            - **Video (MP4)**: extract_text_from_video:
                - Save to temp_video.mp4.

- Extract audio with MoviePy, save as temp_audio.wav, transcribe with SpeechRecognition (Google API), handling errors (e.g., network issues).
- Capture frames at 1-second intervals (max 20), adjust timestamps to fit duration, prioritize the final frame as most relevant.

- Encode each frame to base64, send to Ollama for detailed description (e.g., settings, characters, actions).
- Combine audio and visual text (e.g., "Audio Content: …\nVisual Content: …").
- **Unsupported**: Returns an error message (e.g., "Unsupported file format").
- Log extracted content to processed_data.txt with log_processed_data.
- Append file content to conversation_history (e.g., "Bot: Extracted text from PDF: …").
- **No File**: Proceeds with query using existing context.

3. **Chat Interaction**:
    - **Context Building**: Combine conversation_history into a single string for context.
    - **Prompt Construction**:
        - If an image is uploaded (image_base64):
        - Include image context in the prompt and send base64 data to Ollama.
        - If no image but last_image_context exists:
        - Use the prior image response as context.
        - Otherwise:
            - Use only text context and query.
            - Prompt includes instructions for video summaries (scene progression) or ad identification (main product, emphasizing final frame).
    - **Ollama Request**: POST to OLLAMA_API_URL with stream=True:
        - Payload: Model (llava:7b), prompt, and optional images list.
        - Stream response chunks with response.iter_lines(), decode JSON to extract response field.
    - **Response Generation**: Yield chunks as they arrive, concatenate into full_response, update last_image_context if image-based, append to conversation_history.

4. **Cleanup**:
    - Remove temporary files (temp_video.mp4, temp_audio.wav, temp_frame_X.jpg) after processing.
    - Persist logged data in processed_data.txt for reference.

5. **Output**:
    - Flask returns a Response with streaming text (mimetype="text/plain"), displayed in the frontend as the bot's answer.

Workflow Summary :

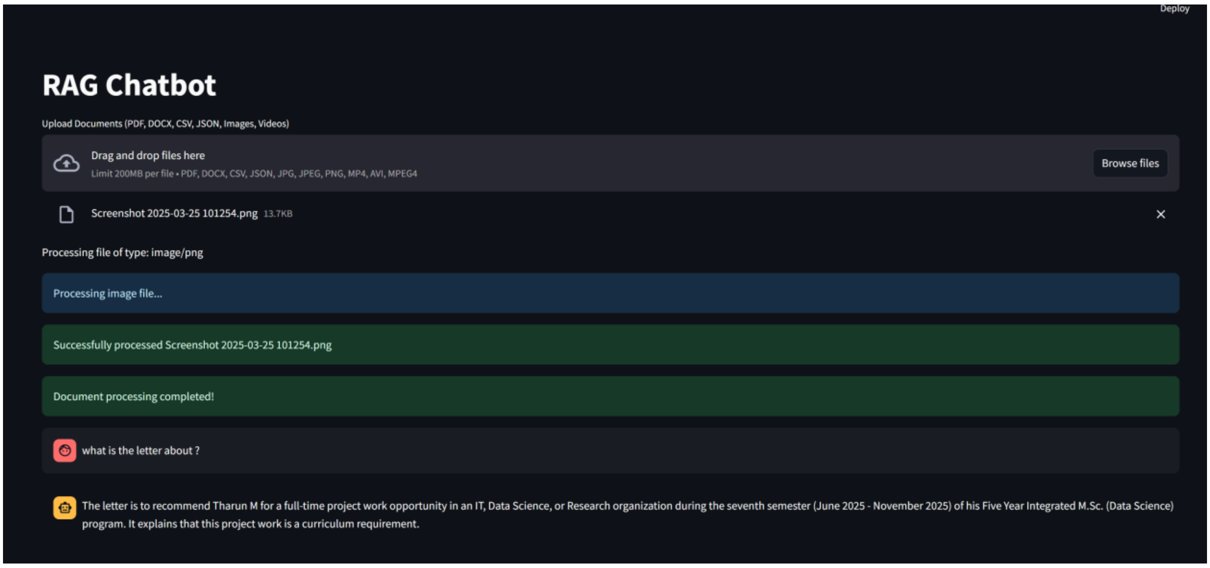
- **Start**: User accesses localhost:5000, views index.html.
- **Action**: Uploads a file (e.g., video) and asks, "What's this about?"
- **Process**: File is saved, audio transcribed, frames described, content logged, and history updated.
- **Response**: Ollama streams a summary (e.g., "This video shows a CPR demo…"), displayed live in the browser.

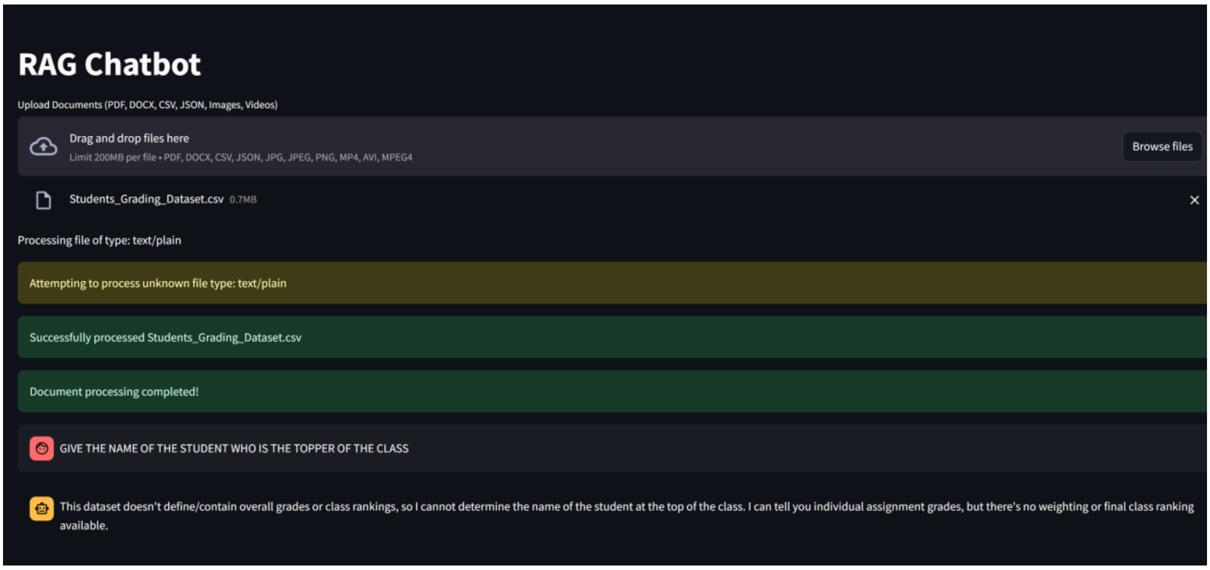
Approach	Tech Stack	Insights	Advantages	Disadvantages	How Better Than Previous
Approach 1: Streamlit with LangChain	Python, Streamlit, google.generativeai, LangChain, sentence-transformers, Chroma, PyPDF2, unstructured, pandas, json, PIL, cv2, easyocr, python-dotenv, python-magic, numpy, tempfile, Poppler (optional)	Web-based UI with robust RAG via LangChain; strong text-based retrieval but limited multimedia depth.	Intuitive UI, persistent vector storage (Chroma), cloud-based LLM (Gemini), multi-file support.	Heavy LangChain dependency, resource-intensive, limited video processing (OCR only), API costs.	N/A (Baseline approach).
Approach 2: Command-Line with	Python, sentence-transformers, FAISS, ollama, PyPDF2, python-docx, pandas,	Lightweight, LangChain-free; excels in video audio +	No LangChain overhead, local LLM (Ollama),	Terminal-only, no persistent storage, single-frame	Reduces dependencies (no LangChain), adds audio

FAISS and Ollama	moviepy, speech_recognition, os, numpy	visual processing but lacks UI and scalability	dual audio-visual video processing, efficient FAISS retrieval.	video analysis, manual file input.	extraction, lighter resource use than Approach 1
Approach 3: Flask with Ollama	Python, Flask, requests, ollama (API), PyPDF2, python-docx, pandas (openpyxl), moviepy, speech_recognition, base64, os, io, collections.deque, json	Web-based with advanced multimedia (multi-frame video analysis); balances usability and functionality.	Web UI, rich video processing (audio + multi-frame), context retention (deque), streaming responses.	No vector storage, local Ollama dependency, frame limit (20), lacks retrieval optimization.	Enhances UI over Approach 2 (Flask vs. terminal), improves video analysis (multi-frame vs. single), retains web accessibility from Approach 1 without LangChain overhead.

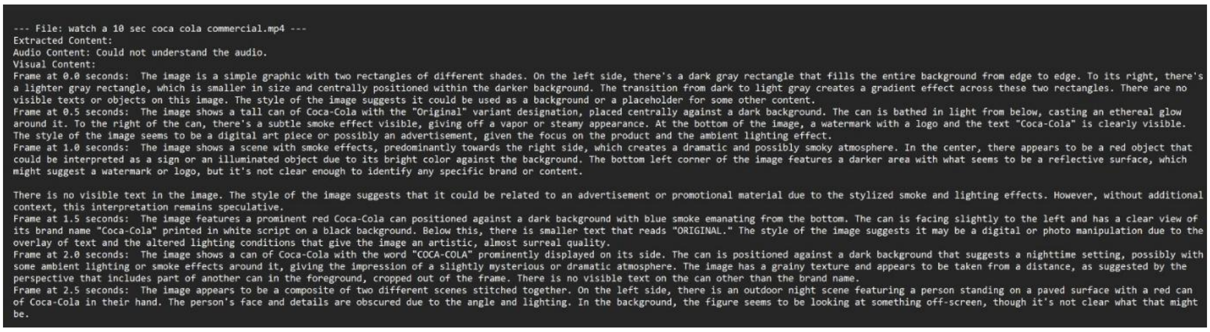
Findings/Output:

1.Approach 1:



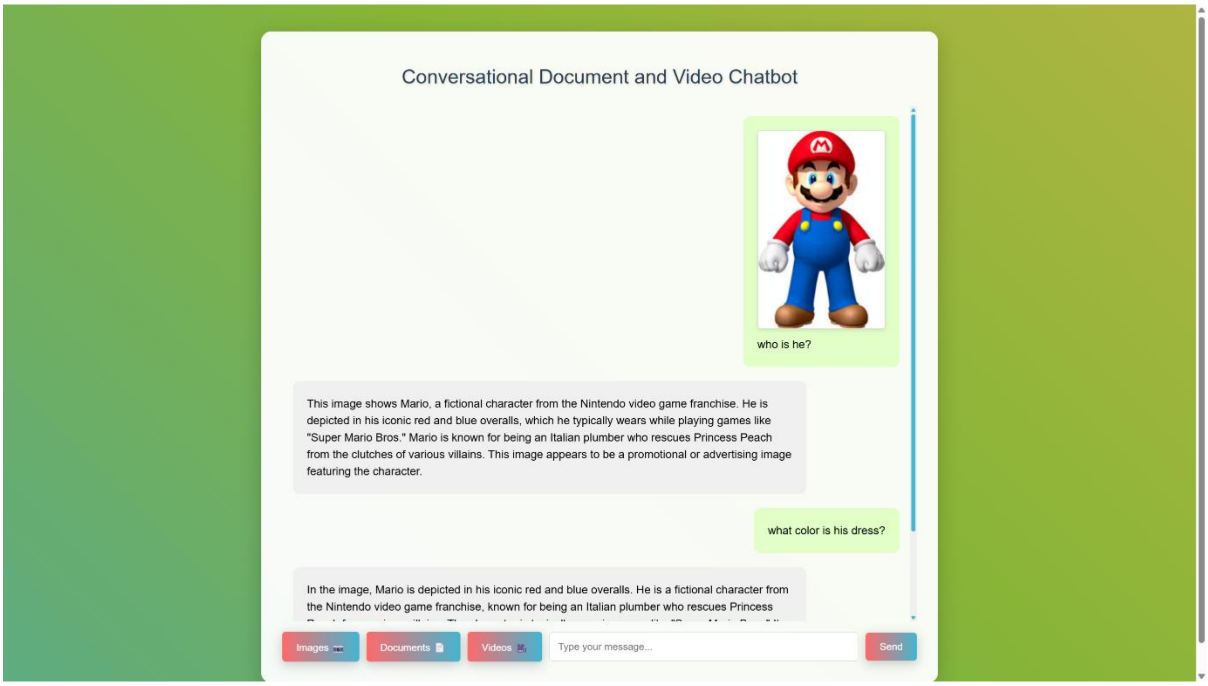


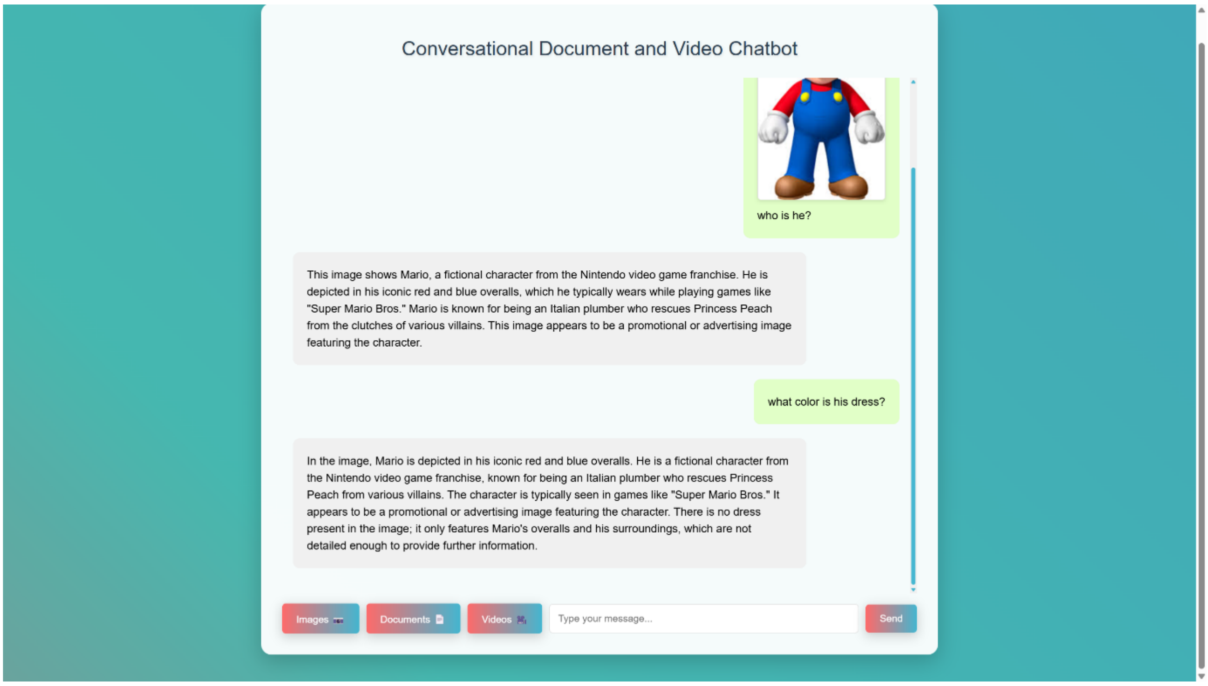
2. Approach 2:

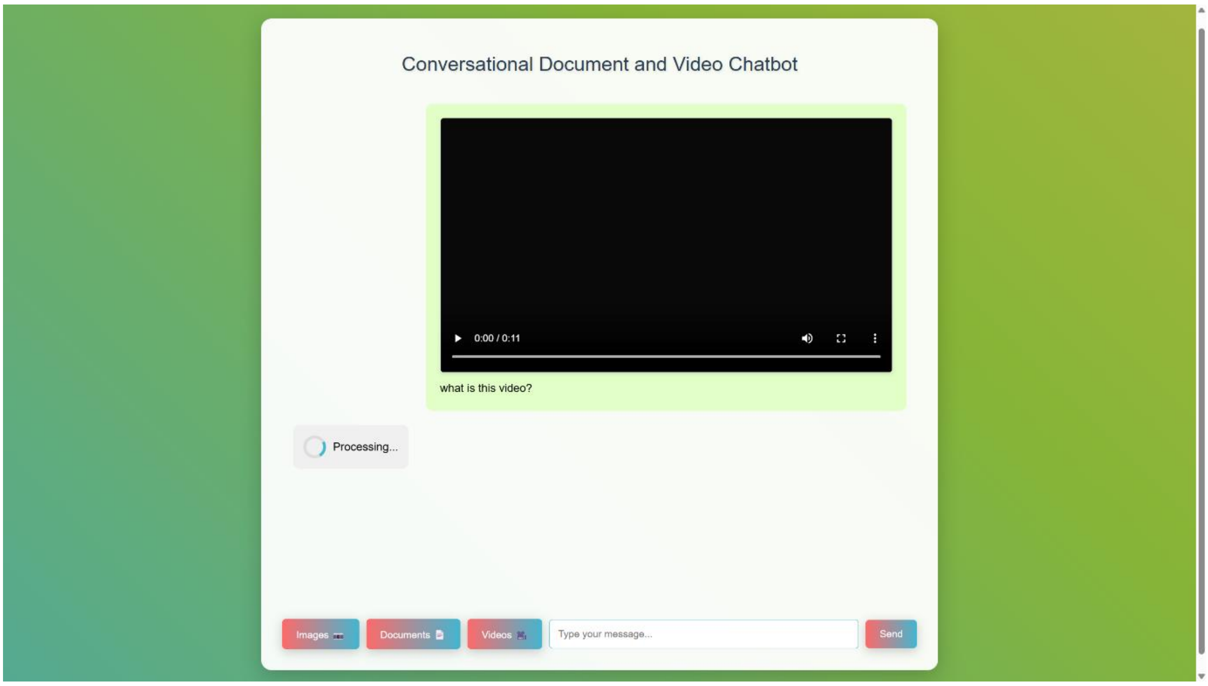


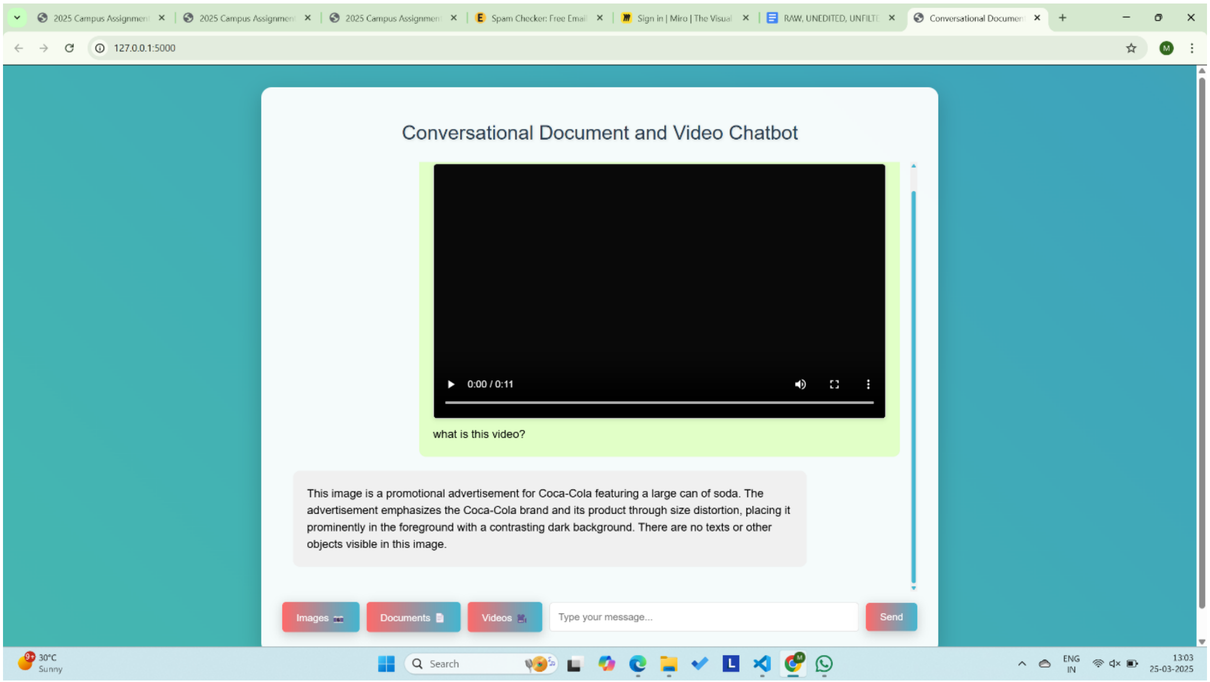
```
--- File: watch a 10 sec coca cola commercial.mp4 ---
Extracted Content:
Audio Content: Could not understand the audio.
Visual Content:
Frame at 0.0 seconds:  The image is a simple graphic with two rectangles of different shades. On the left side, there's a dark gray rectangle that fills the entire background from edge to edge. To its right, there's a lighter gray rectangle, which is smaller in size and centrally positioned within the darker background. The transition from dark to light gray creates a gradient effect across these two rectangles. There are no visible texts or objects on this image. The style of the image suggests it could be used as a background or a placeholder for some other content.
Frame at 0.5 seconds:  The image shows a tall can of Coca-Cola with the "Original" variant designation, placed centrally against a dark background. The can is bathed in light from below, casting an ethereal glow around it. To the right of the can, there's a subtle smoke effect visible, giving off a vapor or steamy appearance. At the bottom of the image, a watermark with a logo and the text "Coca-Cola" is clearly visible. The style of the image seems to be a digital art piece or possibly an advertisement, given the focus on the product and the ambient lighting effect.
Frame at 1.0 seconds:  The image shows a scene with smoke effects, predominantly towards the right side, which creates a dramatic and possibly smoky atmosphere. In the center, there appears to be a red object that could be interpreted as a sign or an illuminated object due to its bright color against the background. The bottom left corner of the image features a darker area with what seems to be a reflective surface, which might suggest a watermark or logo, but it's not clear enough to identify any specific brand or content.

There is no visible text in the image. The style of the image suggests that it could be related to an advertisement or promotional material due to the stylized smoke and lighting effects. However, without additional context, this interpretation remains speculative.
Frame at 1.5 seconds:  The image features a prominent red Coca-Cola can positioned against a dark background with blue smoke emanating from the bottom. The can is facing slightly to the left and has a clear view of its brand name "Coca-Cola" printed in white script on a black background. Below this, there is smaller text that reads "ORIGINAL." The style of the image suggests it may be a digital or photo manipulation due to the overlay of text and the altered lighting conditions that give the image an artistic, almost surreal quality.
Frame at 2.0 seconds:  The image shows a can of Coca-Cola with the word "COCA-COLA" prominently displayed on its side. The can is positioned against a dark background that suggests a nighttime setting, possibly with some ambient lighting or smoke effects around it, giving the impression of a slightly mysterious or dramatic atmosphere. The image has a grainy texture and appears to be taken from a distance, as suggested by the perspective that includes part of another can in the foreground, cropped out of the frame. There is no visible text on the can other than the brand name.
Frame at 2.5 seconds:  The image appears to be a composite of two different scenes stitched together. On the left side, there is an outdoor night scene featuring a person standing on a paved surface with a red can of Coca-Cola in their hand. The person's face and details are obscured due to the angle and lighting. In the background, the figure seems to be looking at something off-screen, though it's not clear what that might be.
```

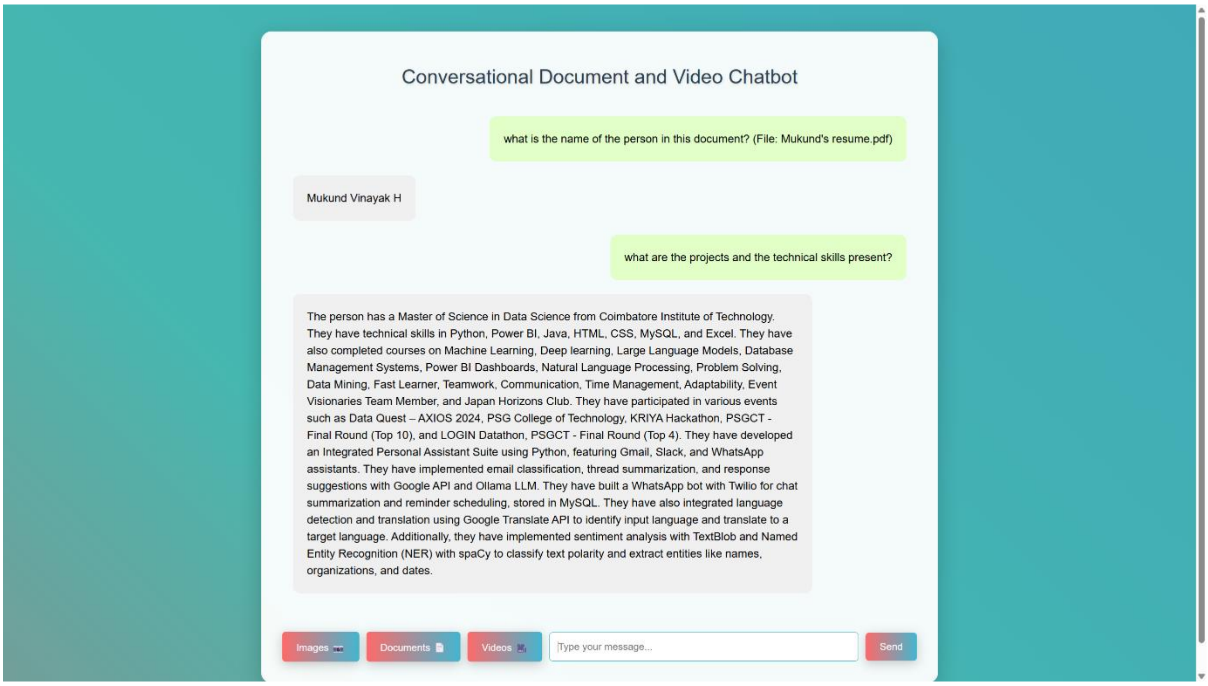
3. Approach 3:

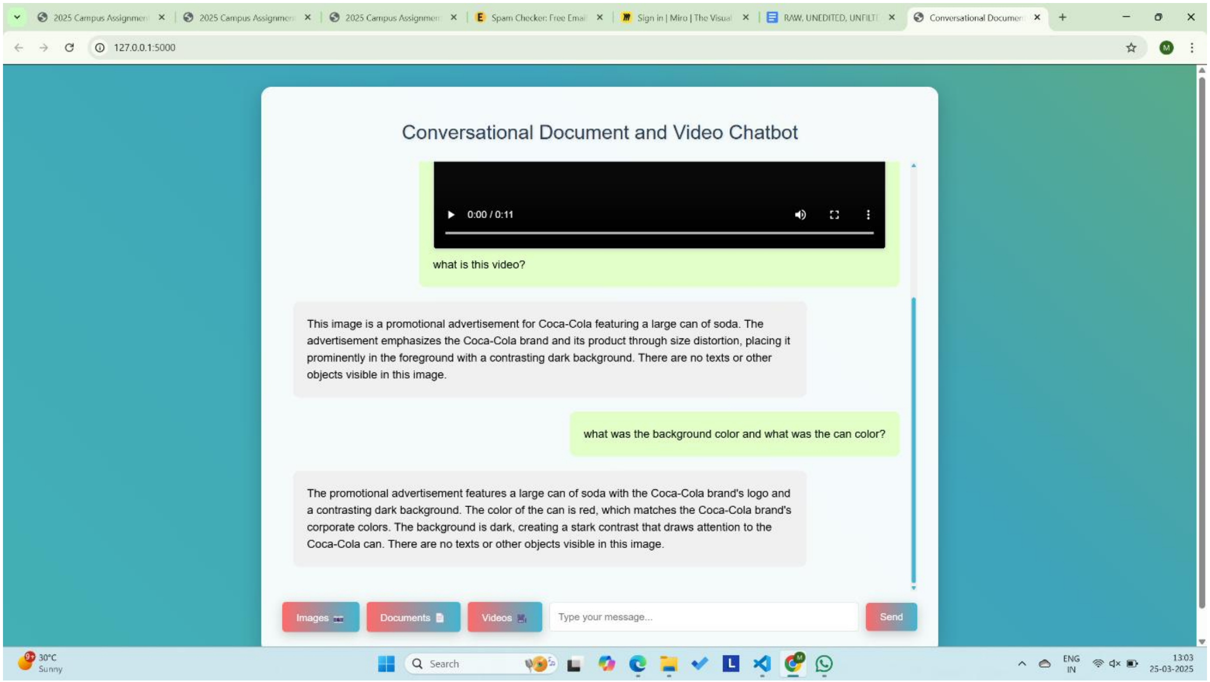

Conversational Document and Video Chatbot
who is he?
This image shows Mario, a fictional character from the Nintendo video game franchise. He is depicted in his iconic red and blue overalls, which he typically wears while playing games like "Super Mario Bros." Mario is known for being an Italian plumber who rescues Princess Peach from the clutches of various villains. This image appears to be a promotional or advertising image featuring the character.
what color is his dress?
In the image, Mario is depicted in his iconic red and blue overalls. He is a fictional character from the Nintendo video game franchise, known for being an Italian plumber who rescues Princess
Images
Documents
Videos
Type your message...
Send


Conversational Document and Video Chatbot
who is he?
This image shows Mario, a fictional character from the Nintendo video game franchise. He is depicted in his iconic red and blue overalls, which he typically wears while playing games like "Super Mario Bros." Mario is known for being an Italian plumber who rescues Princess Peach from the clutches of various villains. This image appears to be a promotional or advertising image featuring the character.
what color is his dress?
In the image, Mario is depicted in his iconic red and blue overalls. He is a fictional character from the Nintendo video game franchise, known for being an Italian plumber who rescues Princess Peach from various villains. The character is typically seen in games like "Super Mario Bros." It appears to be a promotional or advertising image featuring the character. There is no dress present in the image; it only features Mario's overalls and his surroundings, which are not detailed enough to provide further information.
Images
Documents
Videos
Type your message...
Send


Conversational Document and Video Chatbot
0:00 / 0:11
what is this video?
Processing...
Images
Documents
Videos
Type your message...
Send


127.0.0.1:5000
Conversational Document and Video Chatbot
0:00 / 0:11
what is this video?
This image is a promotional advertisement for Coca-Cola featuring a large can of soda. The advertisement emphasizes the Coca-Cola brand and its product through size distortion, placing it prominently in the foreground with a contrasting dark background. There are no texts or other objects visible in this image.
Images
Documents
Videos
Type your message...
Send
Search


Conversational Document and Video Chatbot
what is the name of the person in this document? (File: Mukund's resume.pdf)
Mukund Vinayak H
what are the projects and the technical skills present?
The person has a Master of Science in Data Science from Coimbatore Institute of Technology. They have technical skills in Python, Power BI, Java, HTML, CSS, MySQL, and Excel. They have also completed courses on Machine Learning, Deep learning, Large Language Models, Database Management Systems, Power BI Dashboards, Natural Language Processing, Problem Solving, Data Mining, Fast Learner, Teamwork, Communication, Time Management, Adaptability, Event Visionaries Team Member, and Japan Horizons Club. They have participated in various events such as Data Quest – AXIOS 2024, PSG College of Technology, KRIYA Hackathon, PSGCT - Final Round (Top 10), and LOGIN Datathon, PSGCT - Final Round (Top 4). They have developed an Integrated Personal Assistant Suite using Python, featuring Gmail, Slack, and WhatsApp assistants. They have implemented email classification, thread summarization, and response suggestions with Google API and Ollama LLM. They have built a WhatsApp bot with Twilio for chat summarization and reminder scheduling, stored in MySQL. They have also integrated language detection and translation using Google Translate API to identify input language and translate to a target language. Additionally, they have implemented sentiment analysis with TextBlob and Named Entity Recognition (NER) with spaCy to classify text polarity and extract entities like names, organizations, and dates.
Images
Documents
Videos
Type your message...
Send


127.0.0.1:5000
Conversational Document and Video Chatbot
0:00 / 0:11
what is this video?
This image is a promotional advertisement for Coca-Cola featuring a large can of soda. The advertisement emphasizes the Coca-Cola brand and its product through size distortion, placing it prominently in the foreground with a contrasting dark background. There are no texts or other objects visible in this image.
what was the background color and what was the can color?
The promotional advertisement features a large can of soda with the Coca-Cola brand's logo and a contrasting dark background. The color of the can is red, which matches the Coca-Cola brand's corporate colors. The background is dark, creating a stark contrast that draws attention to the Coca-Cola can. There are no texts or other objects visible in this image.
Images
Documents
Videos
Type your message...
Send
30°C
Sunny
Search
ENG
IN
13:03
25-03-2025

Conclusion :

The Retrieval-Augmented Generation (RAG) Chatbot redefines how users interact with diverse file formats—PDFs, DOCX, CSVs, images, and videos—by transforming static documents into a dynamic, conversational knowledge base, effectively addressing real-world challenges like analyzing sales data, retrieving multimedia protocols, and summarizing educational content. Through three distinct approaches—Streamlit with LangChain for a user-friendly web interface with robust text retrieval, a Command-Line solution with FAISS and Ollama for lightweight efficiency and dual audio-visual processing, and Flask with Ollama for a balanced web-based system with advanced multi-frame video analysis—the project achieves broad file compatibility, precise content extraction, efficient storage, natural language responses, accessibility, and scalability. Each approach caters to unique needs: the first prioritizes usability, the second minimizes overhead, and the third blends functionality with rich multimedia insights, though it trades off vector-based retrieval precision. Together, they showcase a versatile, innovative solution that streamlines information access, proving the power of AI to unlock insights from fragmented data sources.

Future Enhancements

The Retrieval-Augmented Generation (RAG) Chatbot, as implemented across its three approaches, provides a solid foundation for conversational document querying, but several enhancements could elevate its capabilities to meet evolving user demands and technical challenges. Below are key areas for future development:

1. **Speech Recognition for Query Input**
    - **Description**: Integrate speech recognition to allow users to input queries verbally, complementing the existing text-based chat interfaces. This could leverage libraries like speech_recognition (already used in Approaches 2 and 3 for video audio) or advanced models like Google's Speech-to-Text API or Whisper by OpenAI.
    - **Implementation**: Add a microphone button to the Streamlit (Approach 1) or Flask (Approach 3) UI, capturing audio input and converting it to text for processing. For Approach 2, extend the command-line interface to accept voice commands via real-time audio capture.
    - **Benefits**: Enhances accessibility for users with mobility challenges, speeds up interaction in hands-free scenarios (e.g., clinicians in sterile environments), and aligns with multimedia processing strengths in Approaches 2 and 3.

- **Use Case**: A healthcare worker could ask, "What's the dosage protocol?" hands-free while reviewing a video, improving workflow efficiency.

2. **Enhanced JSON Acceptance and Structured Data Processing**
   - **Description**: Expand JSON support beyond basic text conversion (as in Approach 1) to enable structured querying and extraction of specific fields or nested data. This could involve parsing JSON schemas and mapping them to a queryable format.
   - **Implementation**: Integrate a JSON parser (e.g., json with custom logic) to identify key-value pairs or nested objects, storing them in a structured format (e.g., a dictionary or database table). For Approaches 1 and 3, add UI options to select JSON fields; for Approach 2, allow command-line flags to specify keys (e.g., --field "product_name").
   - **Benefits**: Improves precision for business users querying structured data (e.g., "What's the price of item X from this JSON invoice?"), reducing reliance on plain-text conversion and enhancing compatibility with API outputs or configuration files.
   - **Use Case**: A small business owner could query sales JSON data directly (e.g., "List all transactions above $100"), leveraging structured insights without manual parsing.

3. **Database Modeling for Persistent and Scalable Storage**
   - **Description**: Replace or augment the current storage mechanisms—Chroma in Approach 1, in-memory FAISS in Approach 2, and file-based logging in Approach 3—with a relational or NoSQL database (e.g., PostgreSQL, MongoDB) for persistent, scalable, and queryable data management.
   - **Implementation**: Design a schema with tables/collections for documents (metadata like filename, type), extracted text chunks, embeddings (if applicable), and conversation history. For Approach 1, migrate Chroma data to the DB; for Approach 2, store FAISS embeddings alongside metadata; for Approach 3, replace processed_data.txt with DB entries and add indexing for faster retrieval.
   - **Benefits**: Enables long-term storage, multi-session history, and efficient querying (e.g., SQL for text, vector extensions like pgvector for embeddings), supporting larger datasets and concurrent users. Improves scalability for enterprise use.
   - **Use Case**: An educational institution could maintain a searchable archive of lecture PDFs and videos, allowing students to query across semesters without reprocessing files.

4. **Additional Enhancements**

- **Multi-Language Support**: Extend text extraction (e.g., EasyOCR in Approach 1, Ollama in Approaches 2 and 3) to support multiple languages, broadening accessibility for global users (e.g., “Extract Spanish text from this PDF”).
- **Real-Time Video Streaming**: Upgrade video processing to handle live streams (e.g., via WebRTC or OpenCV) instead of static files, enabling real-time analysis (e.g., summarizing a live training session).
- **Advanced Retrieval Optimization**: Incorporate hybrid retrieval (e.g., BM25 + embeddings) or fine-tune embeddings for domain-specific content (e.g., medical terms), enhancing answer relevance across all approaches.
- **User Authentication and Profiles**: Add login features to the web interfaces (Approaches 1 and 3) to save user-specific histories and preferences, improving personalization (e.g., “Show my last 5 queries”).

View code at: Github Link