Retrieval-Augmented Generation (RAG)-based chatbot

Contents:

Statement Objective:

About RAGs:

Retrieval-augmented generation (RAG) is a technique that merges information retrieval with generative AI to produce systems capable of generating precise and contextually relevant responses by tapping into large data repositories. In other words, RAGs assists chatbots, AIs etc to respond to the user based on the context provided. Here the context refers to the files that we upload including but not limited to videos, images, pdfs, docs and csv. Rag works in 2 stages:

1. Retrieval: Uses semantic search to find the documents related to the user input/query. Based on the context given (pdfs, csv, images etc) it find the meaning and the importance of the features more that the mere keywords.

2. Generation: It integrates the information obtained in the retrieval phase to produce desired responses, allowing AI to learn from a large domain of content dynamically. Here I have made use of the assistance of gemini api.

Statement Motive:

“Create a RAG based chatbot that analyses and answers the queries asked by the user based on the specific context given by them. This is much essential to save time and energy”.

Files Present and their use

csv_analysis_chatbot.py

docx_analysis_chatbot.py

image_analysis_chatbot.py

main.py

pdf_analysis_chatbot.py

video_analysis_chatbot.py

chat_history.db

csv_analysis_chat_history.db

docx_analysis_chat_history.db

image_analysis_chat_history.db

image_chat_history.db

pdf_analysis_chat_history.db

video_analysis_chat_history.db

video_chat_history.db

To have a modular approach, I have created .py file for each chatbot based on pdf, docx, image, video, csv analysis etc and the .db file records and store the history of the chat which will be retrieved when the app runs.

Tools and Framework

PROGRAMMING LANGUAGE

- PYTHON
- SQL

TOOLS

- GOOGLE COLAB
- VSCODE

FRAMEWORK

- STREAMLIT
- WEBBROWER

LIBRARIES

streamlit

pandas

numpy

faiss-cpu

python-dotenv

pytesseract

pillow

python-docx

PyPDF2

google-generativeai

1. **Streamlit (st)**

All the chatbot in the above .py file uses streamlit for creating a pleasing visual appearance. Its used to create the interface, create textboxes, getting inputs from the user and displaying UI elements.

2. **Google Generative AI (genai)**

Its used for AI model integration. An API key is generated from the google cloud console which is then used inside the chatbot. This assists in text generation, embedding and content analysis.

3. **FAISS**

It's stands for Facebook AI Similarity search and predominantly used for word embeddings. It helps us to convert text to vectors that is it performs vector embeddings. The similarity search is performed from this library between the user vector of the user query and the context vector.

4. **SQLite3**

All the chatbots are integrated with database management, where the user queries are stored in a separate .db file for each chatbot. The creation of table, storing and the retrieval operations o the chatbot are performed by leveraging the functionalities provided in the SQLite 3 libaray. The library manages persistent storage of conversations, allowing users to return to their previous chats and maintaining conversation context across sessions.

5. **Python-dotenv**

The GOOGLE_API_KEY is given as an environment variable and is stored in .env file. We use this library for environment variable management.

6. **PyPDF2**

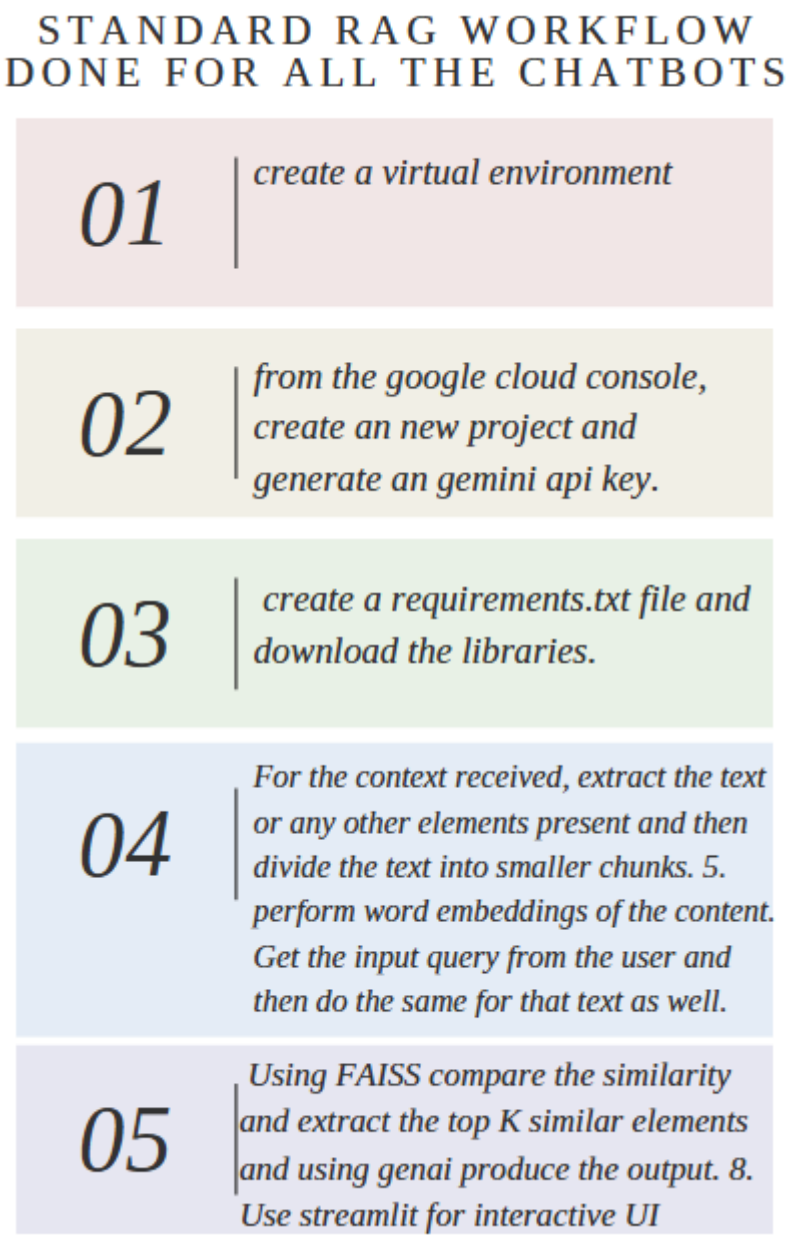
This library is used for the reading of the pdf file which is then converted to text for word embeddings.]

7. **PIL (Python Imaging Library)**

Plays a vital role in image processing in image chatbot and in pdf, docx chatbot. Used for image opening, conversion and processing. It loads sensitive configuration data from a .env file, keeping API keys and other sensitive information secure and separate from the code. This separation of configuration from code is crucial for security and deployment flexibility.

Approach of the problem:

All the chatbot follows the following general workflow with changes implemented for customization of each chatbot.



Chatbot specific Approaches

Pdf based chatbot:

Core Functionality

1. PDF Text Extraction and image extraction
    - Uses both PyPDF2 and PyMuPDF (fitz) for comprehensive PDF proc essing. The images are also extracted.
    - Extracts text content while preserving formatting and structure
    - We also take into consideration of the possibility of multiple documents and each document containin multiple pages and images.

Text Processing Pipeline

1. Text Chunking

- Splits extracted text into manageable chunks and uses overlapping of the chunks to ensure the intrinsic meaning.
- Optimizes chunk size for AI model processing

2. Text Cleaning

- Unwanted whitespaces etc are removed.

AI Integration

- We try to create embeddings for eacha and every chunk of the data. Google's AI model used for the embedding generation. It automatically handles batch processing of large documents.
- The vectors are stored in FAISS index that enables efficient similarity search which is important for response generation.

Query Processing

- We try to identify the vital components od the user query and we map the questions to relevant document sections.

- Multiple chunks are taken into account when needed and FAISS ranks based on relevance.

Response Generation

- Generates answers based on retrieved document context to ensure responses remain grounded in original content.
- Incase the question asked is very deviated from the context we display the appropriate static text to the user.

Similar approach is taken for document and csv chatbots.

2. Image based Chatbot:

Image Processing and Feature Extraction:

- The code handles multiple image formats like jpeg,png,gif and bmp. We also process single images and batches.
- The visual features are extracted from the images and the objects, patterns are identified. The text recognition is done using OCR.

Pre-processing and OCR Processing:

- We resize the images and create optimal brightness. The noise etc are removed and the file is standardised for processing.
- OCR is used to obtain the text from the images. This handles text with various font sizes, multiple languages as well.
- In this code much support is not provided for languages other than English.

AI Integration, query processing and response generation:

- We use the gemini genai AI for the object detection, scene understanding etc.
- We use it to extract the text in the images which are then created as word embeddings to answer the user queries.

- The FAISS library is used to store and retrieve embeddings efficiently. When a user submits a query, the system searches for the most relevant text chunks using FAISS and retrieves them for generating responses.
- Text extraction from images is done using Tesseract OCR. The extracted text is stored along with its embeddings to improve search accuracy and context retention.
- If the image contains charts or graphs, a specialized prompt is used to analyze them separately, extracting key trends, insights, and numerical data. This ensures that both textual and visual data are utilized for answering queries.
- The chatbot processes user questions by first analyzing the query and then matching it with relevant stored embeddings. The closest matching text chunks are retrieved and used to generate an AI-driven response.

Video Based Chatbot:

- The code integrates the Gemini AI model for generating responses from extracted video content. The Gemini API key is securely loaded from environment variables using dotenv.
- The faster-whisper model is used for fast and efficient speech-to-text transcription from video files.
- FAISS is implemented for efficient similarity search and retrieval of relevant text chunks based on user queries. It stores the embeddings of transcribed video content and retrieves the most relevant sections.
- The embedding model, all-MiniLM-L6-v2, from sentence-transformers is loaded to generate vector representations of text chunks. This allows the chatbot to match user queries with relevant transcriptions using FAISS.

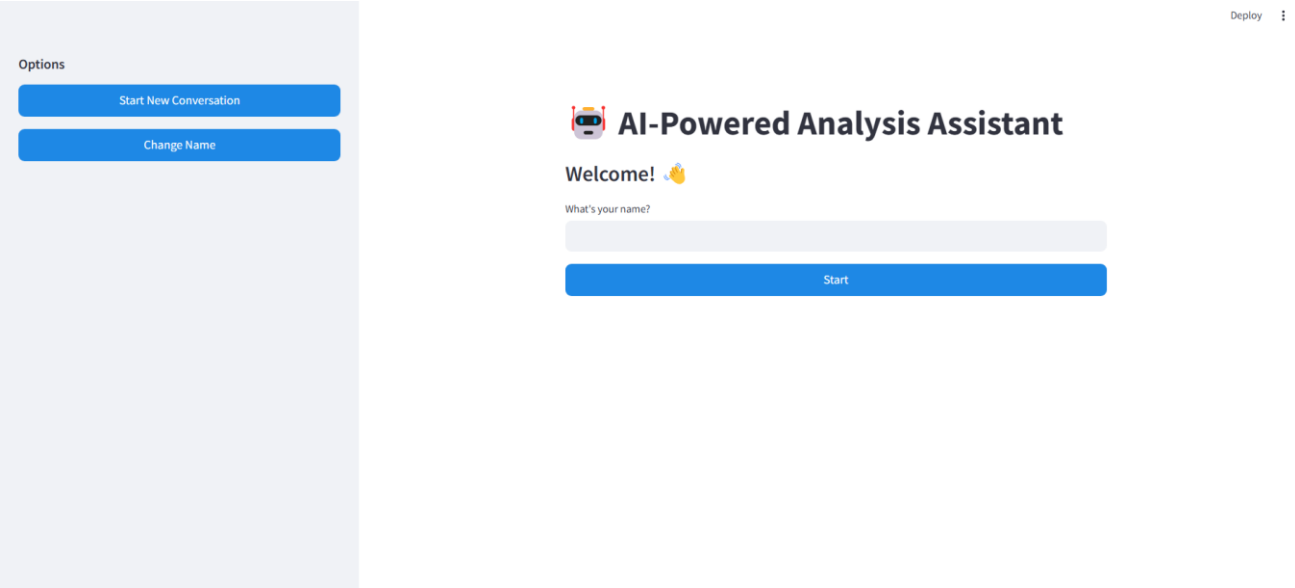
- SQLite is used to store chat history, keeping a record of user queries and AI-generated responses. The chat history is loaded from the database on startup and displayed in an interactive chat interface.
- The extract_audio_from_video function processes uploaded video files by temporarily storing them and preparing them for transcription using Whisper. It ensures error handling by cleaning up temporary files if an error occurs.
- The transcribe_audio function utilizes faster-whisper to convert spoken content into text. It efficiently segments and processes the audio, removing unnecessary pauses and improving transcription accuracy.
- The chatbot interface is built using Streamlit, providing a visually appealing and interactive experience. Users can upload videos, view transcribed text, ask questions, and receive AI-generated responses.
- FAISS indexing ensures that text embeddings are stored efficiently. The chatbot retrieves relevant transcript segments based on query similarity, ensuring accurate responses grounded in the extracted content.
- The retrieve_relevant_chunks function encodes user queries into embeddings and searches the FAISS index to find the closest matching transcript chunks, ensuring responses are contextually relevant.
- The chatbot uses a structured prompt to generate responses from the Gemini AI model. If relevant content is not found in the video, the response informs the user rather than generating unrelated answers.
- Custom CSS is applied to style the chat messages, ensuring a clean and structured UI where user messages and bot responses are visually distinct and well-aligned.
- The chatbot maintains session state in Streamlit, allowing it to store FAISS indexes, text chunks, and embeddings, preventing reprocessing of the video when interacting with the chatbot.

Working of the chatbot:

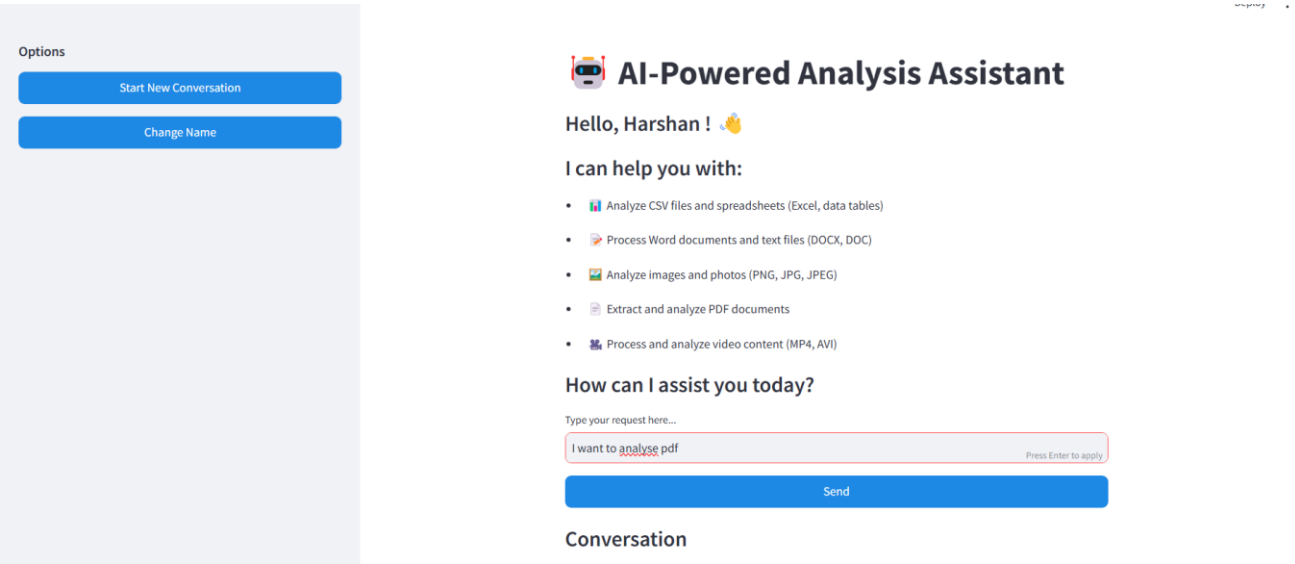
By running the main.py file following output is obtained. The side bar has 2 options :

Start New conversation – clears the chat.

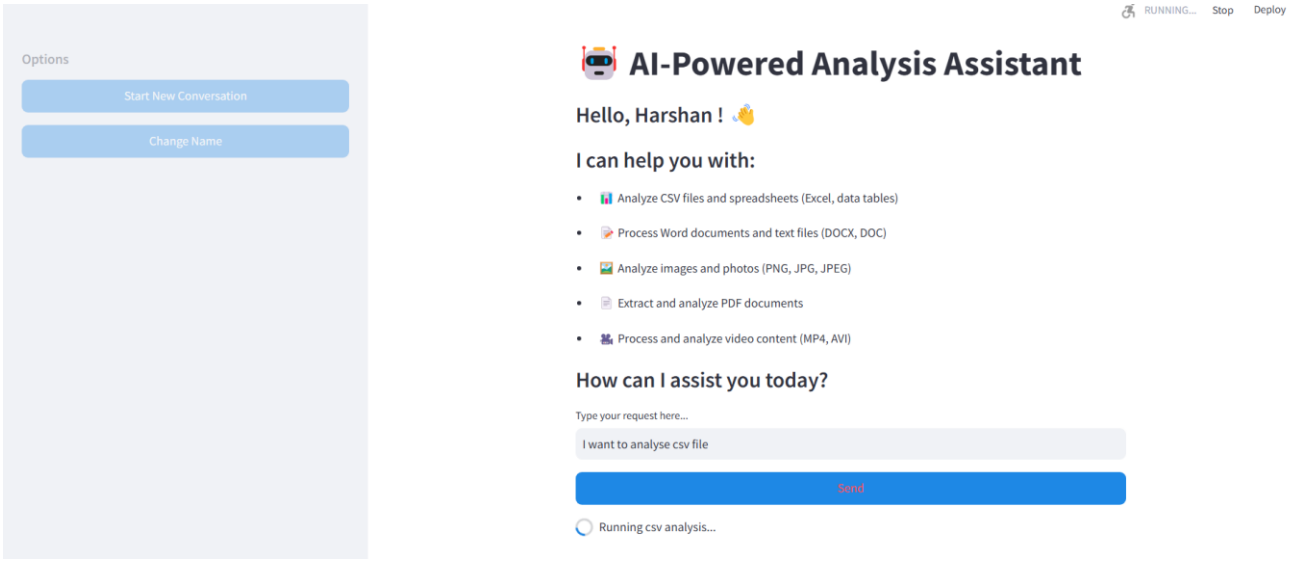
Change Name – The name of the person can be changed.



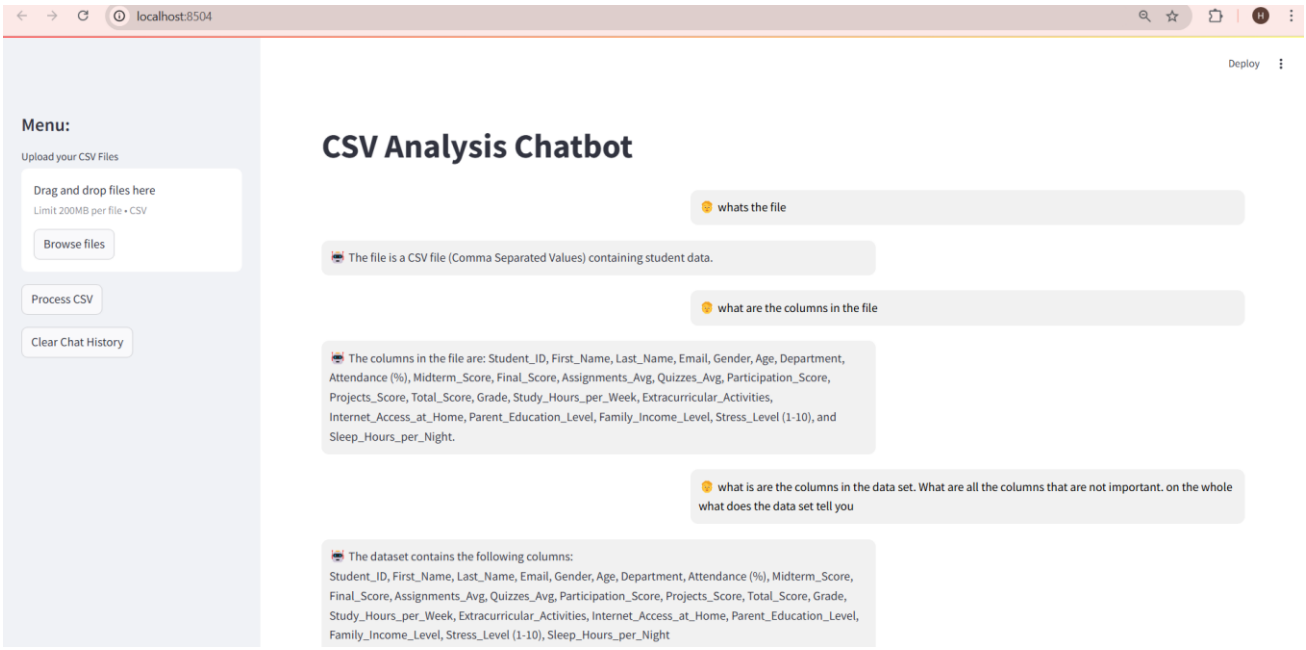
After entering the name, we will be able to see the possible chatbots that we can access.



Based on the query, that particular chat bot will run in a new window.

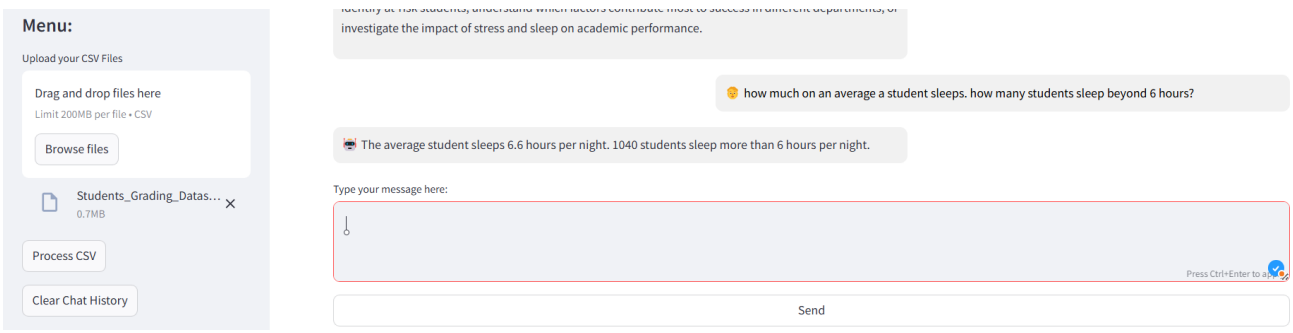


CSV Chatbot:

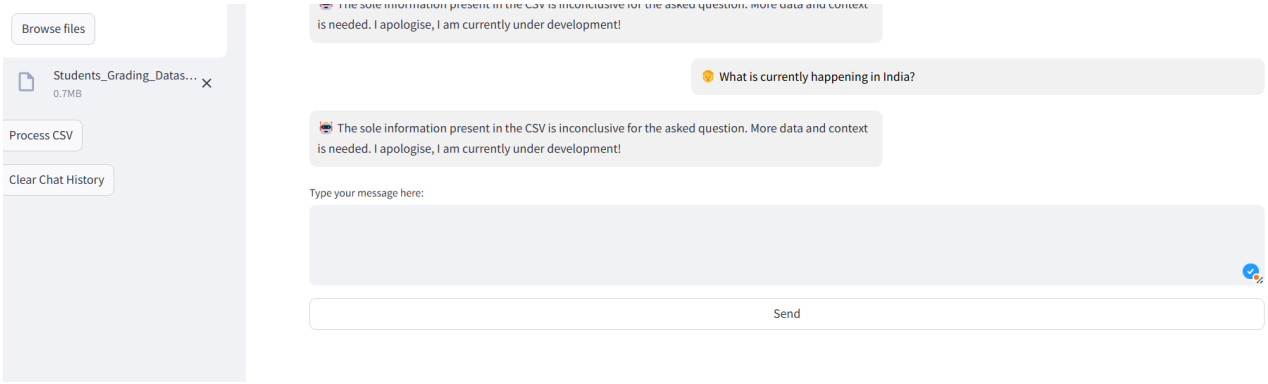


It retrieves the stored old data from the .db file. We can either continue with it or clear the data by clicking the clear chat history button.

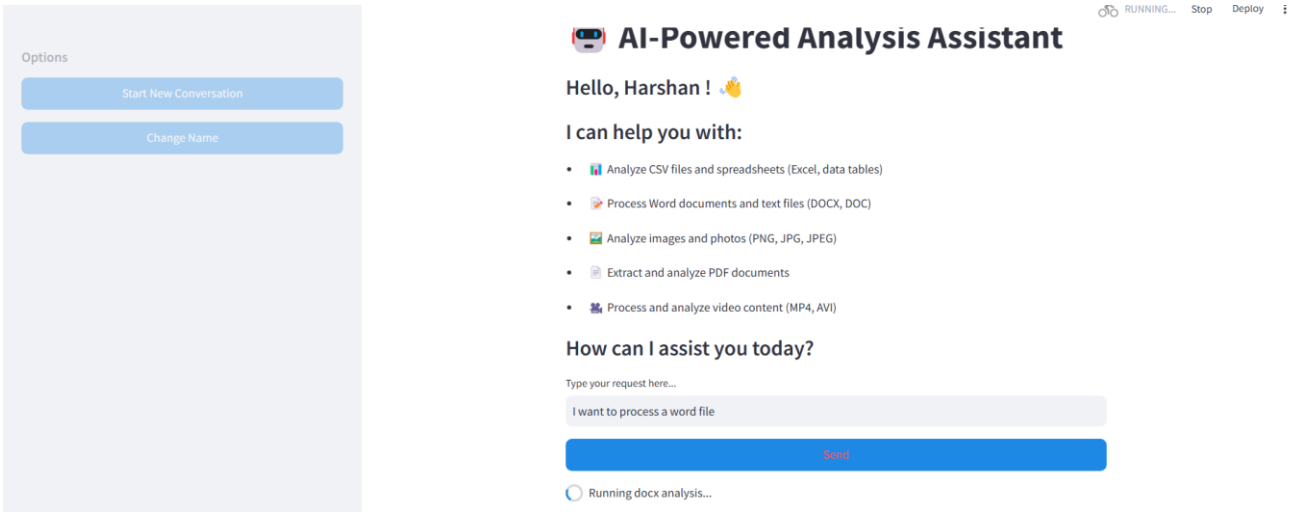
Now I upload the given csv file and we ask queries.



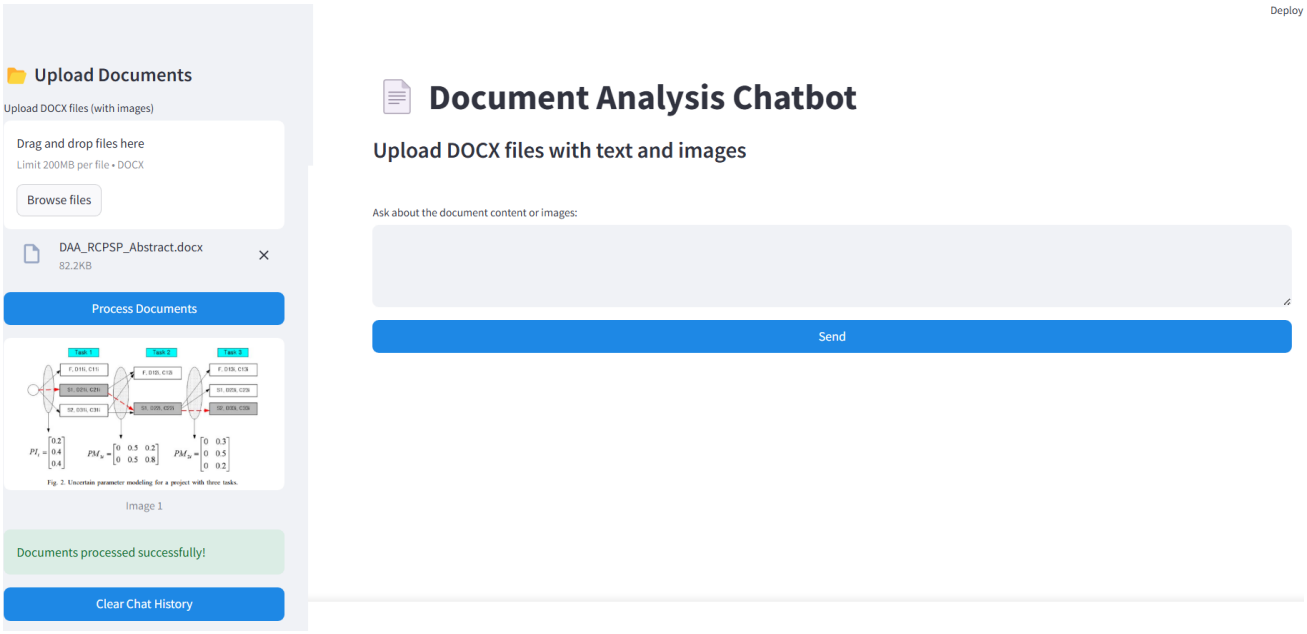
Static message is displayed for unrelated queries:



Docx Chat bot:



I have uploaded a project abstract for the analysis:



The chatbot can correlate with the relevance of the image with the textual content present in the document and make intelligent responses.

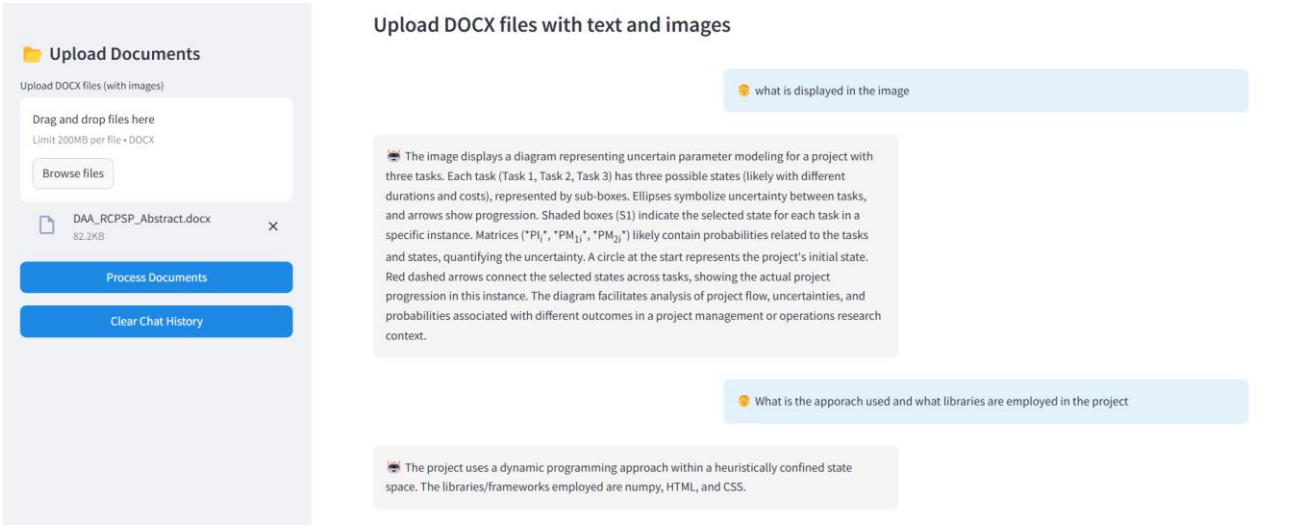
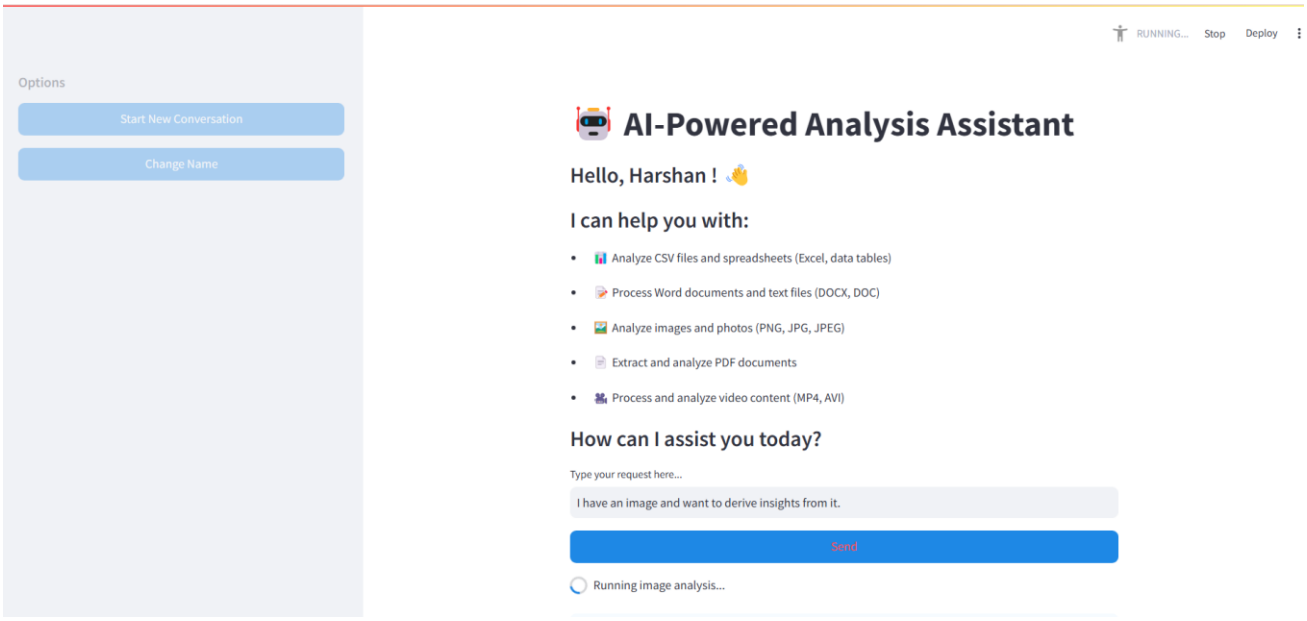


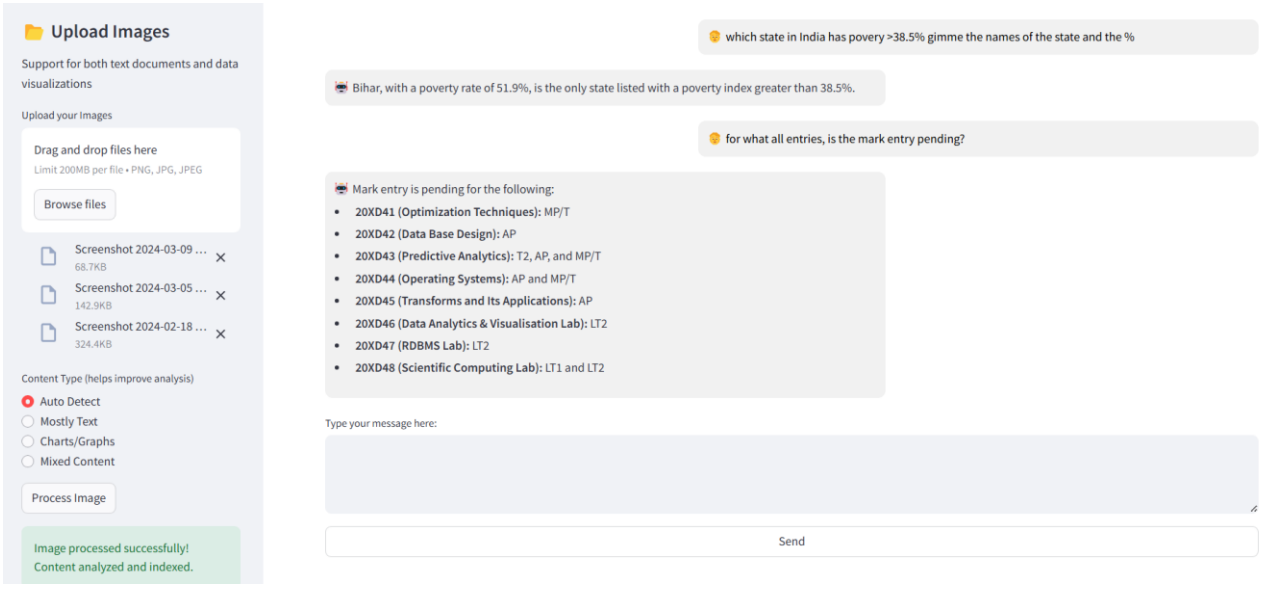
Image based Chatbot:



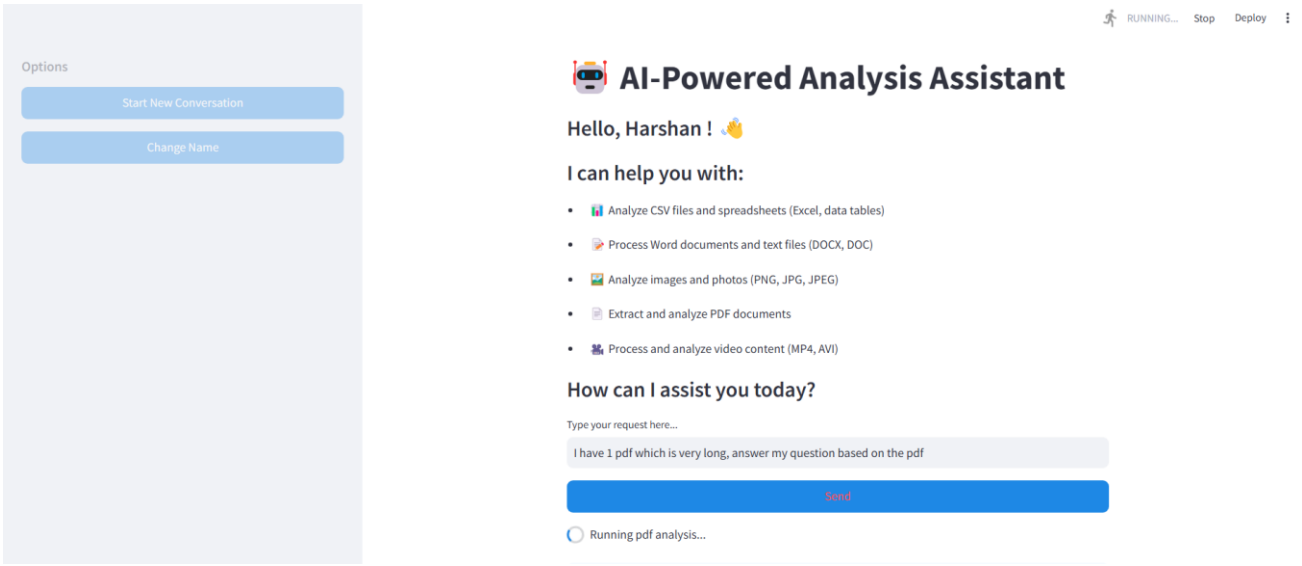
3 images are uploaded-

First containing an India map.
Second containing a screenshot of a table dataset
lastly the mark entry of various subject in PSG tech portal.

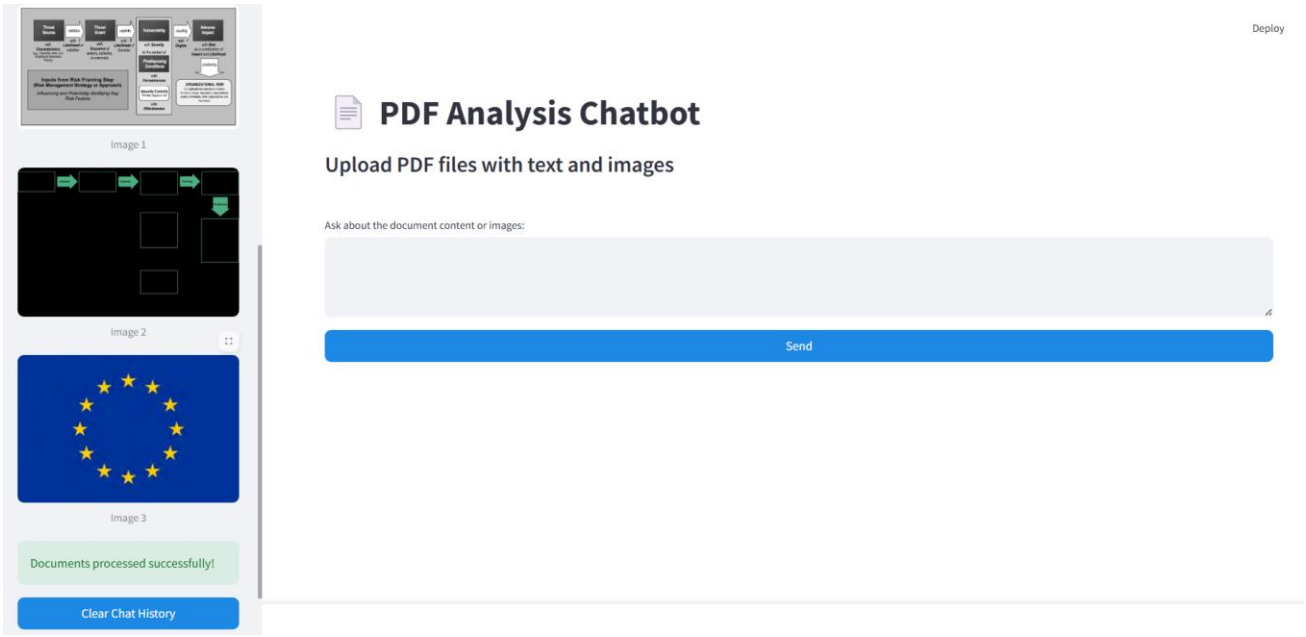
Based on the above the following queries are asked:

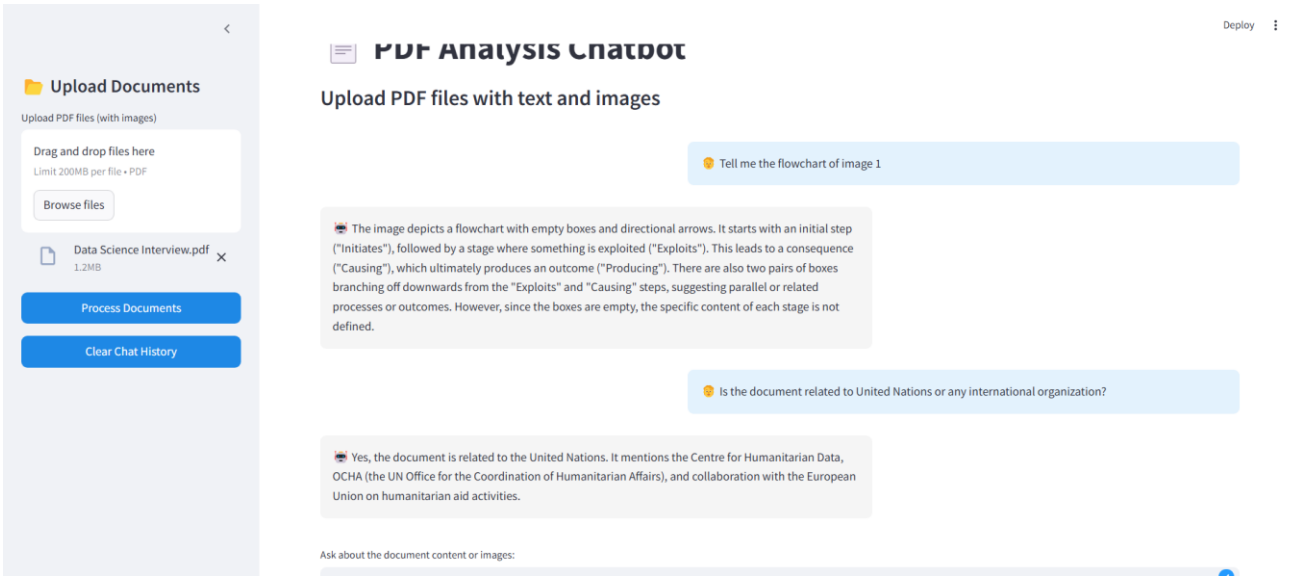


PDF Chatbot:

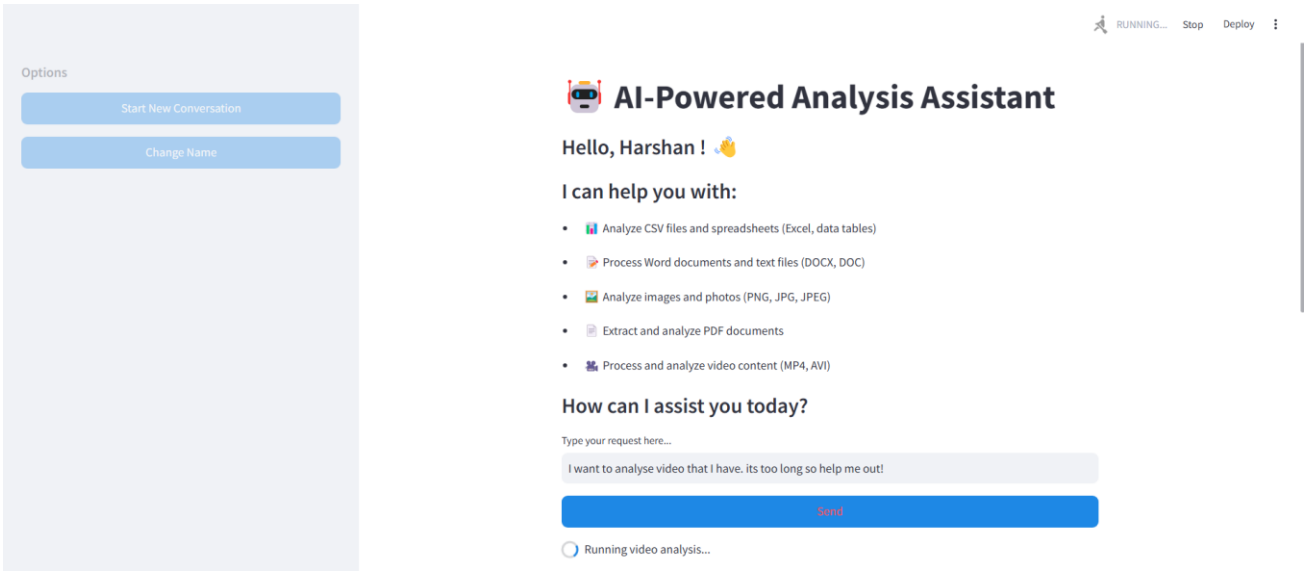


The pdf related to the UN which was sent is uploaded and analysed. It extracts images present in the document. And answers the aquestions:

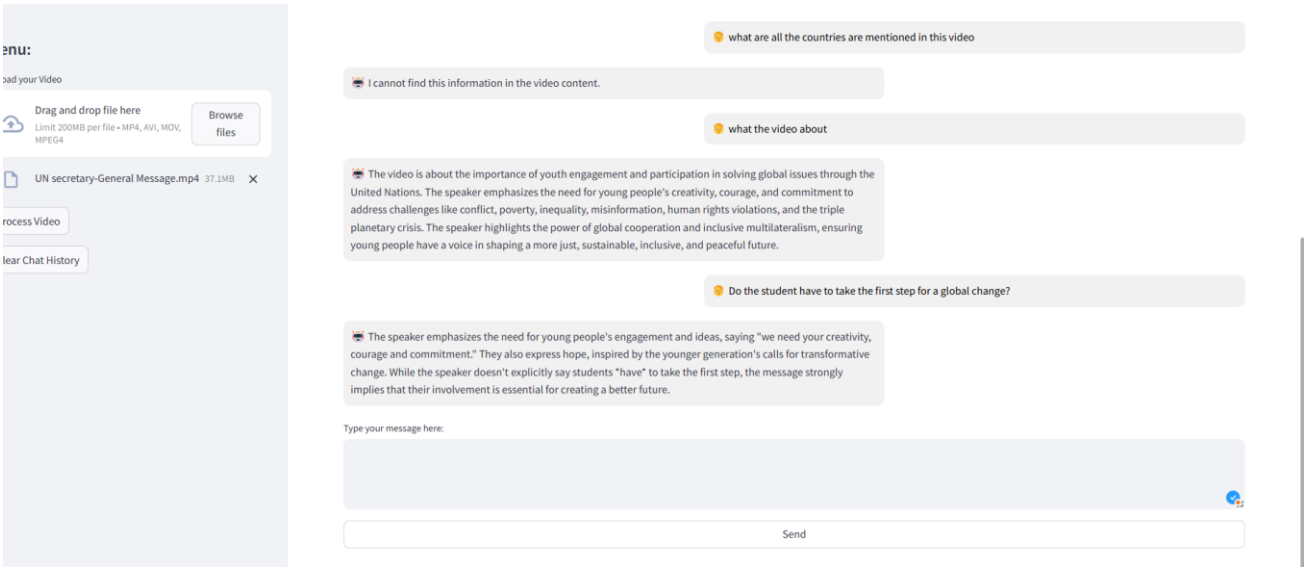




Video Chatbot:



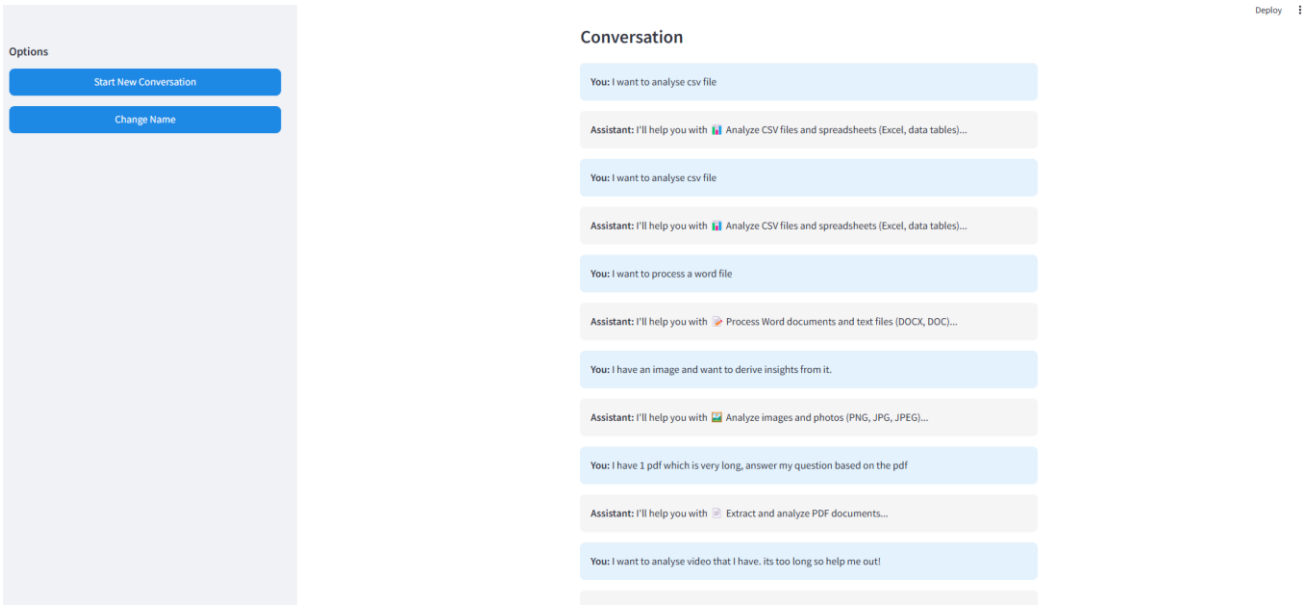
The video uploaded is the speech of the United Nation Secretary General address the world.



The first question is irrelevant since he is address the world.

Main.py:

The history of the requests made the response of the AI in the main home page is also displayed:



Conclusion and recommendations:

This is a complete end-to-end project making use of databases and retrieval operations to integrating AIs to create a phenomenal Rag based chatbot. There are certain pitfalls as well, high end llama models, colpali can be used to enhance the performance and the retrieval of the data from the documents/files provided. These advanced models are heavy weight models that require higher computational capacity of the systems.

End notes:

This code gives just the pinch of what rags can do. True power of RAGs can be experienced by using complex models.
