Mihika Goyal
GlobalLogic

Mihi Assistant

GENAI Chatbot

Jun 2024 - Jul 2024

OVERVIEW

The Personal Data Assistant or Chatbot called the Mihi Assistant is an intelligent conversational agent designed to answer questions based on personal data uploaded by users. This includes various file formats such as PDFs, Word documents, Excel spreadsheets, images, audio, and video files. Apart from this, the chatbot can also be used to generate images by providing a suitable description of the image required.

The project utilises a robust technical stack to achieve its functionality:

- **Python**: The core programming language used for developing the chatbot and implementing NLP and ML algorithms.
- **LangChain**: A framework for developing applications powered by language models.
    - Modules used for embedding:
        - PyPDFLoader: Used to load PDFs.
        - UnstructuredWordDocumentLoader: Used to load word documents.
        - UnstructuredExcelLoader: Used to load excel documents.
        - GenericLoader: Used to handle a variety of document formats.
        - OpenAIWhisperParser: Used to transcribe and parse audio data.
        - YoutubeAudioLoader: Used to load audio data from YouTube videos.
        - AssemblyAITranscriptLoader: Used to load and process transcripts from AssemblyAI.

    - Model used for question answering: GPT-4
    - Model used for image generation: DALLE-3
- **OpenAI**: Utilised for its state-of-the-art language models to comprehend and generate human-like text.
- **Pytesseract**: Utilised for recognising and extracting text from images
- **Flask**: A lightweight web framework for deploying the chatbot as a web service.
- **HTML, CSS, JavaScript**: Technologies used for creating the user interface, ensuring a seamless and interactive user experience.

INSTALLATION

1. Clone the GitHub repository

   ```
   git clone https://github.com/mikagoyal/mihi_assistant.git
   ```

2. Run the following shell scripts in the same order as mentioned :

   ```
   bash prepared.sh
   ```

   ```
   bash run.sh
   ```

USAGE

1. Run the following command in your terminal to start the chatbot:
   ```
   bash run.sh
   ```
2. Once the server starts, you will see a link in the console:
   ```
   * Running on http://127.0.0.1:8000
   ```
3. Click on this link to open the chat interface in your default web browser.
4. In the chat interface, click on the "Upload File(s)" button located at the bottom right corner to upload your desired files (PDFs, Word documents, Excel spreadsheets, images and audio files).
5. After uploading the files, you can:
    - Ask questions about the content of the uploaded documents.
    - Request the chatbot to display extracted text from an uploaded image.
    - Generate a transcript from an uploaded audio file.
6. For video files, provide the chatbot with a YouTube link to any video and have the bot summarise or answer questions based on the video content.

7. To generate images, simply provide a description of the image desired in the chat interface. The bot responds back with a link for the image generated. Open this link in a new browser window or tab.
8. You can upload new documents at any point while the program is running by clicking the "Upload File(s)" button again.

LIBRARIES AND DEPENDENCIES

Main Libraries Imported

1. logging

Description: Part of Python's standard library, logging is used for tracking events that happen when the software runs. It is helpful for debugging and running diagnostics.

Usage: To log messages that can be used to understand the flow of the program and diagnose issues.

2. os

Description: Another standard library in Python, os provides a way of using operating system-dependent functionality like reading or writing to the file system.

Usage: To interact with the operating system, such as accessing environment variables and file handling.

3. flask

Description: A lightweight WSGI web application framework in Python. Flask is designed with simplicity and flexibility in mind.

Usage: To create and run the web server that hosts the chatbot interface.

4. flask_cors

Description: An extension for Flask that allows you to handle Cross-Origin Resource Sharing (CORS), making it possible to access resources on a web page from different domains.

Usage: To enable CORSt_chain in the Flask application, allowing the chatbot to interact with web clients hosted on different domains.

5. flask_socketio

Description: An extension for Flask that provides WebSocket support, allowing for real-time communication between the client and server.

Usage: To handle real-time messaging in the chatbot interface.

6. langchain.chains

Description: A module in the LangChain framework for creating and managing chains of language model invocations. It helps in structuring complex workflows involving multiple language model calls.

Usage: To build and manage different chains of processing steps for the chatbot, including conversational retrieval and multi-prompt handling.

7. langchain.memory

Description: A module in LangChain that deals with memory management, allowing the chatbot to remember previous interactions within a session.

Usage: To store and retrieve conversation history, enabling context-aware responses in ongoing conversations.

8. langchain_core.callbacks

Description: This module provides callback handlers that can be used to execute custom logic at various stages of the chain execution process.

Usage: To implement custom callback functions for logging, monitoring, or modifying the chain execution process in the chatbot.

9. langchain_core.prompts

Description: A module that provides tools for managing and formatting prompts used in language model interactions.

Usage: To create and manage the different prompts that guide the chatbot's responses, ensuring they are structured and informative.

10. langchain.chains.router

Description: This module includes tools for routing tasks to different chains based on the type of query or context.

Usage: To direct different types of user queries to the appropriate processing chains, enhancing the chatbot's ability to handle diverse tasks.

11. langchain_openai

Description: An integration module that connects LangChain with OpenAI's language models, facilitating the use of OpenAI's powerful NLP capabilities.

Usage: To leverage OpenAI's language models for generating responses and performing NLP tasks within the LangChain framework.

12. langchain_community.vectorstores

Description: A module for managing vector stores, which are used to store and query embeddings efficiently.

Usage: To handle storage and retrieval of vector embeddings, supporting similarity searches and other embedding-based operations in the chatbot.

13. langchain.text_splitter.RecursiveCharacterTextSplitter

Description: A text splitter that recursively splits text based on character count, ensuring that the text segments are manageable for processing by language models.

Usage: To divide large documents into smaller, more manageable chunks for easier processing by the chatbot.

14. langchain.text_splitter.CharacterTextSplitter

Description: A text splitter that divides text based on character count.

Usage: To split documents into smaller, manageable chunks for processing by the chatbot.

15. langchain_community.vectorstores.utils.filter_complex_metadata

Description: A utility function for filtering metadata in complex data structures, often used in conjunction with vector stores.

Usage: To filter and manage metadata associated with embeddings and vector searches, ensuring relevant data is retained.

16. langchain_community.document_loaders.AssemblyAIAudioTranscriptLoader

Description: A document loader that uses AssemblyAI to transcribe audio files into text.

Usage: To convert audio files into text transcripts, enabling the chatbot to process and understand audio content.

17. langchain_community.document_loaders.PyPDFLoader

Description: A document loader specifically designed for extracting text from PDF files.

Usage: To load and process PDF documents, allowing the chatbot to read and extract information from them.

18. langchain_community.document_loaders.UnstructuredWordDocumentLoader

Description: A document loader for extracting text from unstructured Word documents.

Usage: To load and process Word documents, enabling the chatbot to extract and use the text content.

19. langchain_community.document_loaders.UnstructuredExcelLoader

Description: A document loader for extracting data from unstructured Excel spreadsheets.

Usage: To load and process Excel files, allowing the chatbot to read and extract information from spreadsheet data.

20. langchain_community.document_loaders.generic.GenericLoader

Description: A generic document loader that can be customised for various document types.

Usage: To load documents of various formats, providing a flexible tool for handling different kinds of data.

21. langchain_community.document_loaders.parsers.audio.OpenAIWhisperParser

Description: A parser that uses OpenAI's Whisper model to transcribe audio files into text.

Usage: To convert audio content into text transcripts, enabling the chatbot to process spoken language.

22. langchain_community.document_loaders.YoutubeAudioLoader

Description: A document loader that downloads and processes audio from YouTube videos.

Usage: To extract and process audio from YouTube videos, allowing the chatbot to analyse and summarise video content.

23. langchain.schema.document.Document

Description: A schema definition for documents within the LangChain framework, encapsulating document metadata and content.

Usage: To represent and manage documents, ensuring they are processed consistently within the chatbot.

24. re

Description: Python's regular expression library, used for string matching and manipulation.

Usage: To perform pattern matching and text processing, such as extracting specific information from text.

25. cv2

Description: OpenCV (cv2) is an open-source computer vision and machine learning software library.

Usage: To perform image processing tasks, such as reading images and preparing them for text extraction.

26. pytesseract

Description: A Python wrapper for Google's Tesseract-OCR Engine, which can be used to recognize text in images.

Usage: To extract text from images, enabling the chatbot to process and understand visual content.

27. openai

Description: OpenAI's Python client library for accessing OpenAI's language models and other AI services.

Usage: To integrate OpenAI's language models for processing and generating responses in the chatbot.

28. dotenv

Description: A library for loading environment variables from a .env file, making it easy to manage configuration settings.

Usage: To load configuration variables (e.g., API keys) from a .env file.

29. chromadb

Description: A database designed for managing embeddings and vector search, often used in conjunction with machine learning applications.

Usage: To store and query vector embeddings for efficient similarity searches.

30. chroma

Description: Part of the LangChain community, this library deals with vector stores, providing tools for working with embeddings and vector searches.

Usage: To handle vector storage and retrieval operations, supporting the chatbot's functionality in managing and searching embeddings.

31.time

Description: The sleep function suspends execution of the current thread for a specified number of seconds.

Usage: To pause for a stipulated number of seconds.

Custom Libraries Imported

1. prompt_definition

Description: This is a python file imported which mainly deals with setting up a prompt template, LLM Chain and running the same.

Usage: This file directly invokes the LLM chain made to get the bot to answer user questions.

2. service

Description: This is a folder which contains the various services provided by the chatbot like image embedding, document embedding, image generation, audio embedding and video embedding.

Usage: This folder is used to access the various services provided by the chatbot.

3. constants

Description: This is a folder containing a python file ‘app_constants.py’.

Usage: This python file is used to import constants used throughout the code.

4. embeddings

Description: This is a python file that creates various objects of the classes present in the service folder and calls methods using these objects. It also has functions to initiate llm embeddings and chroma_db

Usage: This python file is used to initiate the llm embeddings and chroma_db needed to embed various documents along with calling the methods required to embed documents.

CLASSES AND FUNCTIONS

CLASSES

1. TranscriptProcessor

Description: This class exists in the services subfolder, ‘audio_embedding.py’ python file. It processes audio transcripts using AssemblyAI and manages them in a Chroma database.

```
class TranscriptProcessor:
    def __init__(self, chroma_db):
        ...
    def load_transcripts(self, audio_files):
        ...
    def get_transcripts(self):
        ...
```

2. DocumentProcessor

Description: This class exists in the services subfolder, ‘document_embedding.py’ python file. It processes various document types (PDF, DOCX, XLSX) and manages them in a Chroma database.

```
class DocumentProcessor:
    def __init__(self, chroma_db):
        ...
    def process_documents(self, document_paths):
        ...
```

3. ImageProcessor

Description: This class exists in the services subfolder, ‘image_embedding.py’ python file. It processes images using OpenCV and extracts text using pytesseract, storing results in a Chroma database.

```
class ImageProcessor:

    def __init__(self, chroma_db):

        ...

    def add_images(self, collection_name, image_paths):

        ...
```

4. VideoProcessor

Description: This class exists in the services subfolder, ‘video_embedding.py’ python file. It embeds and processes videos from YouTube links using document loaders and stores them in a Chroma database.

```
class VideoProcessor:

    def __init__(self, chroma_db):

        ...

    def load_videos(self, user_prompt):

        ...
```

5. ImageGenerator

Description: This class exists in the services subfolder, ‘image_generation.py’ python file. It generates images based on prompts using the OpenAI DALL-E-3 model.

```
class ImageGenerator:

    def __init__(self, api_key):
```

```
        ...

    def generate_image(self, prompt):

        ...
```

6. Streaming

Description: This class exists in the ‘prompt_definition’ python file. It handles streaming responses from the language model during interaction.

```
class Streaming(BaseCallbackHandler):

    def on_llm_new_token(self, token: str, **kwargs) ->
None:

        ...
```

FUNCTIONS

7. get_answer_from_chain

Description: Retrieves answers from a conversational chain based on user input.

```
def get_answer_from_chain(question):

    ...

    return result
```

8. init_llm

Description: Initialises the language model (LLM) and its embeddings using the OpenAI API key.

```
def init_llm():

    ...
```

9. init_chroma_db

Description: Initialises the Chroma database using existing documents or creates a new one.

```
def init_chroma_db():
    ...
```

10. process_message_route

Description: Processes user messages, handling text extraction, transcript generation, image generation, and interaction with the conversational chain.

```
def process_message_route():
    ...
```

11. process_document_route

Description: Processes uploaded documents, extracting text from PDFs, DOCX, XLSX, and storing them in the Chroma database.

```
def process_document_route():
    ...
```

12. handle_start_stream

Description: Handles WebSocket connections and streams responses based on user messages (transcript generation, video embedding, image generation, conversational responses).

```
@socketio.on('start_stream')
def handle_start_stream(data):
    ...
```

CODE EXPLANATION

prompt_definition.py

`from langchain.chains import LLMChain` `from langchain.memory import ConversationBufferMemory` `from langchain_core.callbacks import BaseCallbackHandler` `from langchain_core.prompts import PromptTemplate` `import embeddings` `from flask_socketio import emit` `import time` `from constants import app_constants`	Adding all the necessary imports
`chat_history = []`	Initialising global variables
`class Streaming(BaseCallbackHandler):` `def on_llm_new_token(self, token: str, **kwargs) -> None:` `emit('stream_response', {'data': token})` `time.sleep(app_constants.TIME)`	Class defined to permit streaming of data on the frontend
`template = """You are a friendly AI assistant and your name is `Mihi Assistant`. Your responsibility is to answer questions based on documents knowledge base. If you did not find the answer from the documents say 'I'm sorry I'm not able to find a relevant answer as per your question'.`	Defining the template used

If the user asks you your name, you only respond with: "Hi, I'm Mihi Assistant. How can I assist you today?". Always respond with short but complete answers unless the user specifically asks to elaborate on something. If the user says "tell me more about this" or something similar, look at the chat history and refer to the provided documents to give more details about the previously asked question. Here is a question about the document: {input} Answer: """	
prompt = PromptTemplate(input_variables=['input'], template=template)	Defining the prompt
memory = ConversationBufferMemory(memory_key='chat_history', return_messages=True, output_key='text')	Adding conversation memory
retriever = embeddings.chroma_db.as_retriever(search_type="similarity", search_kwargs={"k": app_constants.K})	Creating a retriever interface from the vector store
chain = LLMChain(llm=embeddings.llm, prompt=prompt, memory=memory, callbacks=[Streaming()])	Creating an LLM chain
def get_answer_from_chain(question): global chat_history context_str = " ".join(	Function used to get answers by invoking the chain. This first retrieves relevant documents from the chroma_db based

map(lambda x: f"User: {x[0]} Bot: {x[1]}", chat_history)) + " User: " + question retriever = embeddings.chroma_db.as_retriever(search_type="similarity", search_kwargs={"k": app_constants.K}) docs = retriever.get_relevant_documents(question) context_str += " " + " ".join([str(doc) for doc in docs]) result = chain.run(input=context_str) chat_history.append((question, result)) return result	on user questions. A context string is defined which combines chat history, user question and relevant documents. This context string is passed as input to the chain. Finally the result is returned.

server.py

import logging import os from flask import Flask, render_template, request, jsonify from flask_cors import CORS import prompt_definition import embeddings from flask_socketio import SocketIO, emit	Adding all necessary imports

`app = Flask(__name__)` `socketio = SocketIO(app, cors_allowed_origins="*")` `cors = CORS(app, resources={r"/*": {"origins": "*"}})` `app.logger.setLevel(logging.ERROR)`	Initialise Flask app and CORS
`@app.route('/', methods=['GET'])` `def index():` `  return render_template('index.html')`	Define the route for index page
`@app.route('/process-message', methods=['POST'])` `def process_message_route():` `# Extract the user's message from the request` `  user_message = request.json['userMessage']`	Define the route for processing messages
`# Check if the user is requesting transcript generation from the uploaded audio` `elif "transcript" in user_message.lower():` `# Get the generated transcript from the embeddings module` `  generated_transcript = embeddings.get_transcripts()` `# Return the generated transcript as the bot's response` `return jsonify({` `  "botResponse": generated_transcript[0],` `  "generatedTranscript": generated_transcript[0]` `}), 200` `# Check if the user is requesting image generation`	Explained each loop using comments

elif "generate an image of" in user_message.lower(): # Generate the image using the prompt_definition module image_url = prompt_definition.generate_image(user_message) # Return the image URL as the bot's response return jsonify({ "botResponse": f"Here is the image you requested: ", "imageUrl": image_url }), 200 # Process the user's message using the prompt_definition module bot_response = prompt_definition.get_answer_from_chain(user_message) # Return the bot's response along with the extracted text as JSON return jsonify({ "botResponse": bot_response }), 200	
@app.route('/process-document', methods=['POST']) def process_document_route(): if 'files' not in request.files: return jsonify({	Define the route to process documents uploaded by user Check if files were uploaded

`"botResponse": "No files uploaded. Please upload a PDF, DOCX, XLSX, or image file."` `}), 400`	
`files = request.files.getlist('files')` `file_paths = []` `file_extensions = []`	Extract the uploaded files from the request
`for file in files:` `file_path = os.path.join("uploads", file.filename)` `file.save(file_path)` `file_paths.append(file_path)` `_, file_extension = os.path.splitext(file_path)` `file_extension = file_extension.lower()` `file_extensions.append(file_extension)`	Define the path where the file is saved Save the file Extract the file extension of uploaded file
`doc_files = [file_paths[i] for i in range(len(file_paths)) if file_extensions[i] in {'.pdf', '.docx', '.xlsx'}]`	Extract all docx files
`image_files = [file_paths[i] for i in range(len(file_paths)) if file_extensions[i] in {'.png', '.jpg', '.jpeg'}]`	Extract all image files
`audio_files = [file_paths[i] for i in range(len(file_paths)) if file_extensions[i] in {'.mp3'}]`	Extract all audio files

`if doc_files:` `  embeddings.doc.process_documents(doc_files)` `if image_files:` `  embeddings.image.add_images(image_files)` `if audio_files:` `  embeddings.audio.load_transcripts(audio_files)` `return jsonify({` `  "botResponse": "Thank you for providing your files. They have been analysed, and you can now ask any questions regarding them!"` `}), 200`	Process the documents using embeddings module Process images using embeddings module Process audio using embeddings module Return a success message as JSON
`@socketio.on('start_stream')` `def handle_start_stream(data):` `  user_message = data.get('userMessage')` `if "transcript" in user_message.lower():` `  transcripts = embeddings.audio.get_transcripts()` `  for transcript in transcripts:` `  for token in transcript.split():` `  stream.on_llm_new_token(token)` `  emit('stream_response', {'data': '\n'})` `if "http" in user_message.lower():` `  embeddings.video.load_videos(user_message)`	Define a route to handle Websocket connections and stream responses Check if user requires a transcript Stream the transcript generated Check if the user has provided a youtube link. If yes, load the video

str = "" if "generate an image of" in user_message.lower(): str = embeddings.generate_img.generate_image(user_message) else: str = prompt_definition.get_answer_from_chain(user_message) stream = prompt_definition.Streaming() for token in str.split(): stream.on_llm_new_token(token=token) emit('stream_response', {'data': '\n'})	Check if user has asked to generate an image If none of the cases match, simply get response from bot Stream the response
if __name__ == "__main__": # Ensure the upload directory exists os.makedirs("uploads", exist_ok=True) socketio.run(app, debug=True, port=8000, host='0.0.0.0')	Run the flask app

embeddings.py

import os from langchain_openai import OpenAIEmbeddings, ChatOpenAI from langchain_community.vectorstores import Chroma from dotenv import load_dotenv from service import audio_embedding, document_embedding, generate_image, image_embedding, video_embedding	Adding all the necessary imports

`OPENAI_API_KEY = "sk-proj-..."` `load_dotenv()`	Load environment variables
`llm = None` `llm_embeddings = None` `chroma_db = None`	Initialise global variables
`def init_chroma_db():` `  global chroma_db` `  # Check if Chroma DB already exists` `  persist_directory = "./data"` `  if os.path.exists(persist_directory):` `  # Load existing Chroma DB` `  chroma_db = Chroma(persist_directory=persist_directory, embedding_function=llm_embeddings)` `  else:` `  # Create a new Chroma DB if it does not exist` `  chroma_db = Chroma(embedding_function=llm_embeddings)`	Function to initialise chroma db from existing documents
`def init_llm():` `  global llm, llm_embeddings` `  # Initialise the language model with the OpenAI API key` `  OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")` `  llm = ChatOpenAI(streaming=True, model_name="gpt-4", openai_api_key=OPENAI_API_KEY)` `  # Initialise the embeddings for the language model`	Function to initialise language model and its embeddings

`llm_embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)`	
`init_llm()` `init_chroma_db()`	Initialising chroma db, llm and its embeddings
`audio = audio_embedding.TranscriptProcessor(chroma_db)` `video = video_embedding.VideoProcessor(chroma_db)` `doc = document_embedding.DocumentProcessor(chroma_db)` `image = image_embedding.ImageProcessor(chroma_db)` `generate_img = generate_image.ImageGenerator(OPENAI_API_KEY)`	Creating objects

service module

audio_embedding.py

`from langchain.text_splitter import RecursiveCharacterTextSplitter` `from langchain_community.vectorstores.utils import filter_complex_metadata` `from langchain_community.document_loaders import AssemblyAIAudioTranscriptLoader`	Adding all the necessary imports
`transcripts = []`	Initialising global variable
`class TranscriptProcessor:` `def __init__(self, chroma_db):`	Defining class TranscriptProcessor Defining constructor

`self.chroma_db = chroma_db` `self.transcripts = []`	
`def load_transcripts(self, audio_files):` `global transcripts` `self.transcripts = []` `for audio_file in audio_files:` `try:`	Defining method to embed text from audio
`transcript_loader = AssemblyAIAudioTranscriptLoader(file_path=audio_file)` `docs = transcript_loader.load()`	Loads the transcript from an audio file
`docs = filter_complex_metadata(docs)`	Filters complex metadata
`texts = RecursiveCharacterTextSplitter(chunk_size=app_constants.CHUNK_SIZE,chunk_overlap=app_constants.CHUNK_OVERLAP).split_documents(docs)`	Splits text into chunks
`for text_segment in texts:` `self.chroma_db.add_documents([text_segment])` `self.chroma_db.persist()`	Adds to chroma db and saves it
`for text_segment in texts:` `self.transcripts.append(text_segment.page_content.strip())`	Appending the text into transcript list

print(f"Processed and added transcript for audio file: {audio_file}") except Exception as e: print(f"Error processing audio file {audio_file}: {e}") return self.transcripts def get_transcripts(self): global transcripts return transcripts	Defining a method to get returned transcripts

document_embedding.py

import os from langchain_community.document_loaders import PyPDFLoader, UnstructuredWordDocumentLoader, UnstructuredExcelLoader from langchain.text_splitter import CharacterTextSplitter from constants import app_constants	Adding all the necessary imports
class DocumentProcessor: def __init__(self, chroma_db):	Defining class DocumentProcessor Defining constructor

```
    self.chroma_db = chroma_db


    def process_documents(self, document_paths):
        for document_path in document_paths:
            try:
            _, file_extension =
os.path.splitext(document_path)
            file_extension = file_extension.lower()
            if file_extension == '.pdf':
                loader = PyPDFLoader(document_path)
            elif file_extension == '.docx':
                loader =
UnstructuredWordDocumentLoader(document_path)
            elif file_extension == '.xlsx':
                loader =
UnstructuredExcelLoader(document_path)
            else:
                print(f"Unsupported file type:
{file_extension}")
                continue


            if file_extension in {'.pdf', '.docx',
'.xlsx'}:
```

Defining method to embed text from documents

Extract file extension

documents = loader.load() print(f"document: {documents}") text_splitter = CharacterTextSplitter(chunk_size=app_constants.CHUNK_SIZE, chunk_overlap=app_constants.CHUNK_OVERLAP) texts = text_splitter.split_documents(documents) print(f"text: {texts}") self.chroma_db.add_documents(texts) self.chroma_db.persist() print(f"Processed and added document: {document_path}") except Exception as e: print(f"Error processing document {document_path}: {e}") print("-----------")	Load the document Splits text into chunks Adds to chroma db and saves it

image_embedding.py

import cv2 import pytesseract import chromadb	Adding all the necessary imports

`from langchain.text_splitter import CharacterTextSplitter` `from langchain.schema.document import Document` `from constants import app_constants`	
`class ImageProcessor:` `  def __init__(self, chroma_db):` `  self.chroma_db = chroma_db` `  def add_images(self, image_paths):` `  for image_path in image_paths:` `  try:` `  text = pytesseract.image_to_string(img)` `  text_splitter = CharacterTextSplitter(chunk_size=app_constants.CHUNK_SIZE, chunk_overlap=app_constants.CHUNK_OVERLAP)` `  texts = [Document(page_content=x) for x in text_splitter.split_text(text)]` `  # Add new documents to the Chroma DB` `  self.chroma_db.add_documents(texts)` `  self.chroma_db.persist()` `  print(f"Processed and added image: {image_path}")` `  except Exception as e:` `  print(f"Error processing image {image_path}: {e}")`	Defining class ImageProcessor Defining constructor Defining method to embed text from images Extract text from image using pytesseract Split text into chunks Creating Document object Adding to chroma db

video_embedding.py

`import re` `from langchain_community.document_loaders.generic import GenericLoader` `from langchain_community.document_loaders.parsers.audio import OpenAIWhisperParser` `from langchain_community.document_loaders import YoutubeAudioLoader`	Adding all the necessary imports
`class VideoProcessor:` `    def __init__(self, chroma_db):` `        self.chroma_db = chroma_db` `    def load_videos(self, user_prompt):` `        try:` `            video_link = re.search(r'https?://\S+', user_prompt).group()` `            save_dir="uploads"` `            loader = GenericLoader(YoutubeAudioLoader(` `                                [video_link], save_dir),` `                                OpenAIWhisperParser()` `            )` `            docs = loader.load()`	Defining class VideoProcessor Defining constructor Defining method to embed text from video Extract video link from user message Loading video

`` for doc in docs: self.chroma_db.add_documents([doc]) self.chroma_db.persist() print("Video embedded successfully") except Exception as e: print(f"Error embedding video: {e}") ```	Adding to chroma db

generate_image.py

`` from openai import OpenAI from constants import app_constants ```	Adding all the necessary imports
`` class ImageGenerator: def __init__(self, api_key): self.api_key = api_key self.client = OpenAI(api_key=api_key) def generate_image(self, prompt): try: response = self.client.images.generate(model="dall-e-3", prompt=prompt, n=app_constants.N, ```	Defining class ImageGenerator Defining constructor Defining method to generate image

size="1024x1024") image_url = response.data[0].url return image_url except Exception as e: print(f"Error generating image: {e}") return None	Returning image url generated

FUTURE IMPROVEMENTS

1. Error Handling

Enhance error handling across modules to provide more informative messages and handle edge cases gracefully.

2. Security

Implement security best practices such as input validation, sanitization of user inputs, and secure handling of API keys.

3. Processing Efficiency

Improve text processing and image/video handling algorithms for better performance.

4. Scalability

- Evaluate and enhance the scalability of the application architecture, especially concerning the Chroma database and handling of large datasets.
- Consider using containerization (e.g., Docker) and orchestration tools (e.g., Kubernetes) for easier deployment and scaling.

5. Documentation

- Expand and improve inline comments and docstrings to enhance code readability and maintainability.

- Create comprehensive developer documentation and API references for easy onboarding of new contributors.

6. Testing

- Develop unit tests and integration tests to verify the functionality of each module and ensure robustness across different scenarios.
- Implement continuous integration (CI) pipelines to automate testing and deployment processes.

7. Feature Enhancements

- Introduce additional features such as multi-language support, sentiment analysis, or integration with more AI models for richer interactions.
- Extend capabilities for handling different file formats and media types beyond the current scope.

CONTRIBUTING GUIDELINES

If you're interested in contributing to this project, consider the following guidelines:

1. Familiarise Yourself

Understand the existing codebase, its architecture, and how different modules interact.

2. Identify Areas

Look for areas where improvements or new features can be added based on the project's roadmap or existing issues.

3. Follow Coding Standards

Adhere to the coding style and guidelines used in the project. Maintain consistency with existing code.

4. Discuss Changes

Before making significant changes, discuss them with the project maintainers to ensure alignment with project goals and avoid duplication of efforts.

5. Write Tests

When adding new features or modifying existing ones, include appropriate tests to validate the functionality and prevent regressions.

6. Document Your Code

Ensure that your contributions are well-documented, including inline comments, docstrings, and updates to the project's documentation if necessary.

7. Submit Pull Requests

Fork the repository, make your changes in a separate branch, and submit a pull request. Provide a clear description of your changes and reference any related issues.

8. Participate in Discussions

Engage in discussions on issues and pull requests, offer constructive feedback, and collaborate with other contributors.