

# Youtube Speaker Diarizer and Summarizer Project Tutorial + Report

This project is designed to process a YouTube video, identify different speakers using diarization (speaker separation), transcribe each speaker's audio using a large speech-to-text model (Whisper from Hugging Face), and then summarize the main ideas, opinions, and arguments made by each speaker using GPT-4. The system is especially useful for summarizing talk shows, interviews, debates, or podcast-style videos. This guide will show you how to do all these things and

## Project Objectives

To build an intelligent, privacy aware assistant that was capable of

- Running on web + mobile using React Native
- Storing long term memory using vector embeddings with ChromaDB
- Retrieve and reasoning over important emails, documents, and alerts
- Responding via OpenAI or local models like Ollama
- Uses whisper for potential voice input
- Keeps everything locally hosted

## Technologies and Libraries Used

yt-dlp	Downloads YouTube videos and extracts most relevant audio
transformers	Converts speech to text
pyannote.audio	Identifies and separates different speakers
openai	Summarizes speaker specific text
Miniconda, .env, dotenv	Used for configuring virtual environment and secret configuration management
ffmpeg	Used for audio format handling/wav extraction
torch	CUDA/CPU control

## Folder Structure Breakdown

Top level project folder- youtube\_diarizer\_06\_15\_2025

- .env stores HuggingFace key/OpenAI APIkey/any necessary authentication securely
- Summarize\_youtube.py is main orchestration script(downloads YouTube as wav, runs actual diarization, facilitates speaker summarization)
- Audio.wav is generated audio file from Youtube video which gets diarized
- audio/ is a temp folder used during audio extraction
  - It gets auto deleted after use and created during use
- venv/ creates conda environment folder

- Not committed to version control

#### .env File format

- Ensure that you have a .env file in your project root that has something like the following
  - The purpose of this code is to securely load our secret API keys for HuggingFace and OpenAI

```
HF_TOKEN=your_huggingface_token_here
OPENAI_API_KEY=your_openai_key_here
```

#### Setup Instructions

- To install everything you need to begin, add the following
  - To create a new Conda environment in Miniconda

```
# Create a new Conda environment
conda create -n yt_diarizer python=3.10 -y
conda activate yt_diarizer
```

- To install any packages you'd need in your terminal

```
pip install yt-dlp ffmpeg-python python-dotenv torch torchaudio pyannote.audio scipy
openai transformers pydub
```

- To run the summarizer on any YouTube video link input this in your terminal

```
python summarize_youtube.py "https://www.youtube.com/watch?v=your_video_id"
```

#### Handling document ingestion, embedding, retrieval - memory/

- The memory/ folder is critical — it turns static documents into searchable knowledge using embeddings. Ingested files are transformed into vector embeddings with SentenceTransformers and stored in ChromaDB. This allows the assistant to perform semantic search, not just keyword lookup — meaning it can reason over meaning and context. This is what gives the assistant its “memory” capability, a key differentiator from basic chatbots.
- Create\_database.py reads in relevant files, sends to the openAI AI for embedding, and stores these embeddings in Chroma DB

```
import chromadb
from sentence_transformers import SentenceTransformer
import os

client = chromadb.Client()
collection = client.get_or_create_collection("memory")
model = SentenceTransformer("all-MiniLM-L6-v2")

def store_memory(text: str):
    print(f"Storing text in memory: {text[:50]}...")
    vec = model.encode([text])[0]
    collection.add(documents=[text], embeddings=[vec], ids=[text[:8]])
    print("Memory stored.")
```

```
def ingest_documents(folder_path: str):
    for filename in os.listdir(folder_path):
        if filename.endswith(".txt"):
            with open(os.path.join(folder_path, filename), 'r') as file:
                text = file.read()
                store_memory(text)
```

- Query\_database.py accepts a query from the user, finds relevant text in stored memory, and returns the top 'k' text chunks

```
import chromadb
from sentence_transformers import SentenceTransformer
from chromadb.utils import similarity_search

client = chromadb.Client()
collection = client.get_or_create_collection("memory")
model = SentenceTransformer("all-MiniLM-L6-v2")

def retrieve_context(query: str, top_k: int = 3) -> str:
    query_vec = model.encode([query])[0]
    results = collection.query(embedding=query_vec, top_k=top_k)

    context_chunks = [doc for doc in results['documents']]
    context = "\n".join(context_chunks)

    print(f"Retrieved context for query '{query}': {context}")

    return context
```

#### Handling audio transcription - voice\_input/

- The voice\_input/ module allows me to test Whisper transcription locally before exposing it via the Flask route. While simple, this is a powerful capability — users could speak to the assistant and get natural language responses. In future versions, this would be tightly integrated with the React Native app using voice-to-text buttons or wake-word detection.
- Whisper\_recorder.py works to test local audio

```
import whisper

model = whisper.load_model("base")

def transcribe_audio(file_path: str):
    result = model.transcribe(file_path)
    print("Transcription result:")
    print(result['text'])
    return result['text']

if __name__ == "__main__":
    transcribe_audio("test_audio.wav")
```

- Flask's /transcribe route accepts .wav input and calls Whisper model

```

    @app.route('/transcribe', methods=['POST'])
    def transcribe():
        audio = request.files['file']
        result = whisper_model.transcribe(audio)
        return jsonify({'text': result['text']})

```

## Frontend React Native app using expo - mobile\_app/

- [App.js](#) used for main frontend ui and handles text input, submit button, assistant response, and displays relevant memory graph

```

import React, { useState } from 'react';
import { View, TextInput, Button, Text, Image } from 'react-native';
import axios from 'axios';

const FLASK_BASE_URL = 'http://localhost:5000';

export default function App() {
    const [input, setInput] = useState('');
    const [response, setResponse] = useState('');

    const sendMessage = async () => {
        try {
            const res = await axios.post(`${FLASK_BASE_URL}/`, { user_input: input });
            setResponse(res.data.reply);
        } catch (error) {
            console.error("Error sending message:", error);
        }
    };

    return (
        <View style={{ padding: 20 }}>
            <TextInput
                placeholder="Ask your assistant..."
                value={input}
                onChangeText={setInput}
                style={{ borderWidth: 1, padding: 10, marginBottom: 10 }}
            />
            <Button title="Send" onPress={sendMessage} />
            <Text style={{ marginVertical: 20 }}>{response}</Text>
            <Image
                source={{ uri: `${FLASK_BASE_URL}/static/memory_graph.png` }}
                style={{ width: 300, height: 200 }}
            />
        </View>
    );
}

```

Backend logic - ui/[app.py](#)

- Receives user input, retrieves memory from ChromaDB, builds prompt to send to LLM, and returns final message

```
from flask import Flask, request, jsonify, send_file
from memory.query_database import retrieve_context
from llm_engine.openai_engine import get_openai_response
import whisper

app = Flask(__name__)

# Load Whisper model for audio transcription
whisper_model = whisper.load_model("base")

@app.route('/', methods=['POST'])
def query():
    user_input = request.json['user_input']
    print(f"Received user input: {user_input}")

    context = retrieve_context(user_input)
    print(f"Retrieved context: {context}")

    prompt = f"Context: {context}\n\nUser: {user_input}"
    response = get_openai_response(prompt)

    return jsonify({'reply': response})

@app.route('/transcribe', methods=['POST'])
def transcribe():
    audio = request.files['file']
    result = whisper_model.transcribe(audio)
    return jsonify({'text': result['text']})


@app.route('/static/memory_graph.png')
def memory_graph():
    return send_file('static/memory_graph.png', mimetype='image/png')






if __name__ == "__main__":
    app.run(debug=True)
```


## Personal LLM Assistant

Can you summarize my emails from yesterday and tell me if there's

### Assistant Response:

 Here's a summary of your emails from June 3:






1. Prof. Chen confirmed your research meeting (Friday 2 PM).  No reply needed.
2. Lockheed Martin followed up about your internship — deadline Thursday.  Reply recommended.
3. Penn State University Library sent an overdue notice (due May 31st).  Consider renewing.
4. Mom sent a grocery list for the weekend — she wants you to pick up almond milk, eggs, and avocados.  Optional.
5. Amazon confirmed your textbook delivery — arriving by Wednesday.  No action needed.

-  You have 1 urgent email to respond to (Lockheed Martin).

### UI and Axios integration

- This part of the React Native app is intentionally kept minimal, focusing on core functionality. However, even this simple Axios integration demonstrates the full loop — user types a message, the backend retrieves context, the LLM responds, and the UI displays it. The memory graph visualization provides an extra touch — showing the assistant is using more than just the current chat input.
- Uses react hooks, takes a *textInput* for user messaging, the *button* sends query to flask, and finally displays the final result and resulting memory graph

### Working input/output

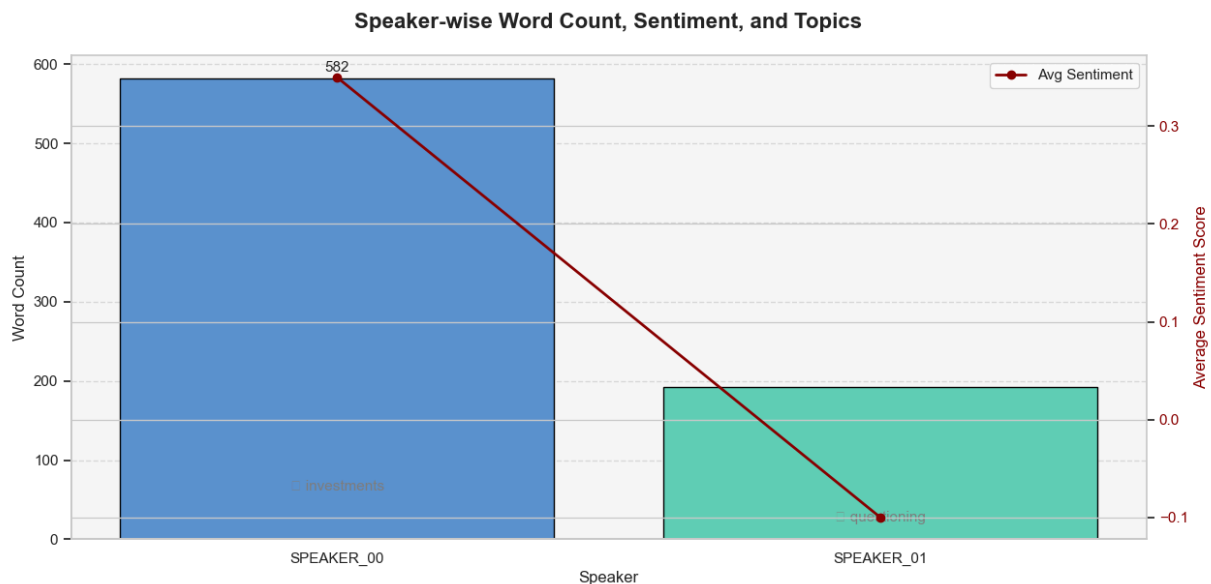
- Input: Can you summarize my emails from yesterday and tell me if there's anything I should respond to?
- Output:
  -  Prof. Chen replied — no action needed
  -  Lockheed Martin follow-up — REPLY recommended
  -  Overdue notice from PSU Library
  -  Grocery list from mom
  -  Amazon delivery notice

### Possible future enhancements

- Chat history and threading
- Voice-to-text UI toggle
- Live memory graph update
- Secure login by user

These are just a few of the many directions this assistant could evolve in. Adding chat history and threading would give the assistant true conversational memory, helping it maintain context across multiple interactions. A voice-to-text UI toggle would make hands-free use much easier, especially on mobile. A live-updating memory graph could give users a visual sense of what the assistant "remembers" and how it connects information. Finally, adding secure user login would allow for personalizing the

assistant experience while keeping sensitive data protected — an essential step for any production-ready version.



- This chart breaks down the key metrics from the diarizer output — how much each speaker talked, the tone of what they said, and their main topic. In this case, SPEAKER\_00 dominated the conversation with a mostly positive sentiment, discussing topics like investments and properties. SPEAKER\_01, on the other hand, had fewer contributions and came across more neutral or questioning. Visuals like this make it way easier to quickly understand a conversation without digging through raw transcripts or full summaries. You can instantly see who's driving the discussion, what kind of tone they're bringing, and where the focus is. Super useful for breaking down interviews, podcasts, debates — anything long-form with multiple voices.

## Final Thoughts

- While on the surface this project might just seem like the average AI chatbot project, it's not. It's actually
  - A cross-platform assistant
  - Capable of long term memory retrieval
  - Voice-interactive
  - Has a deployable demo of how it could work today