Voice Assistant – Assignment

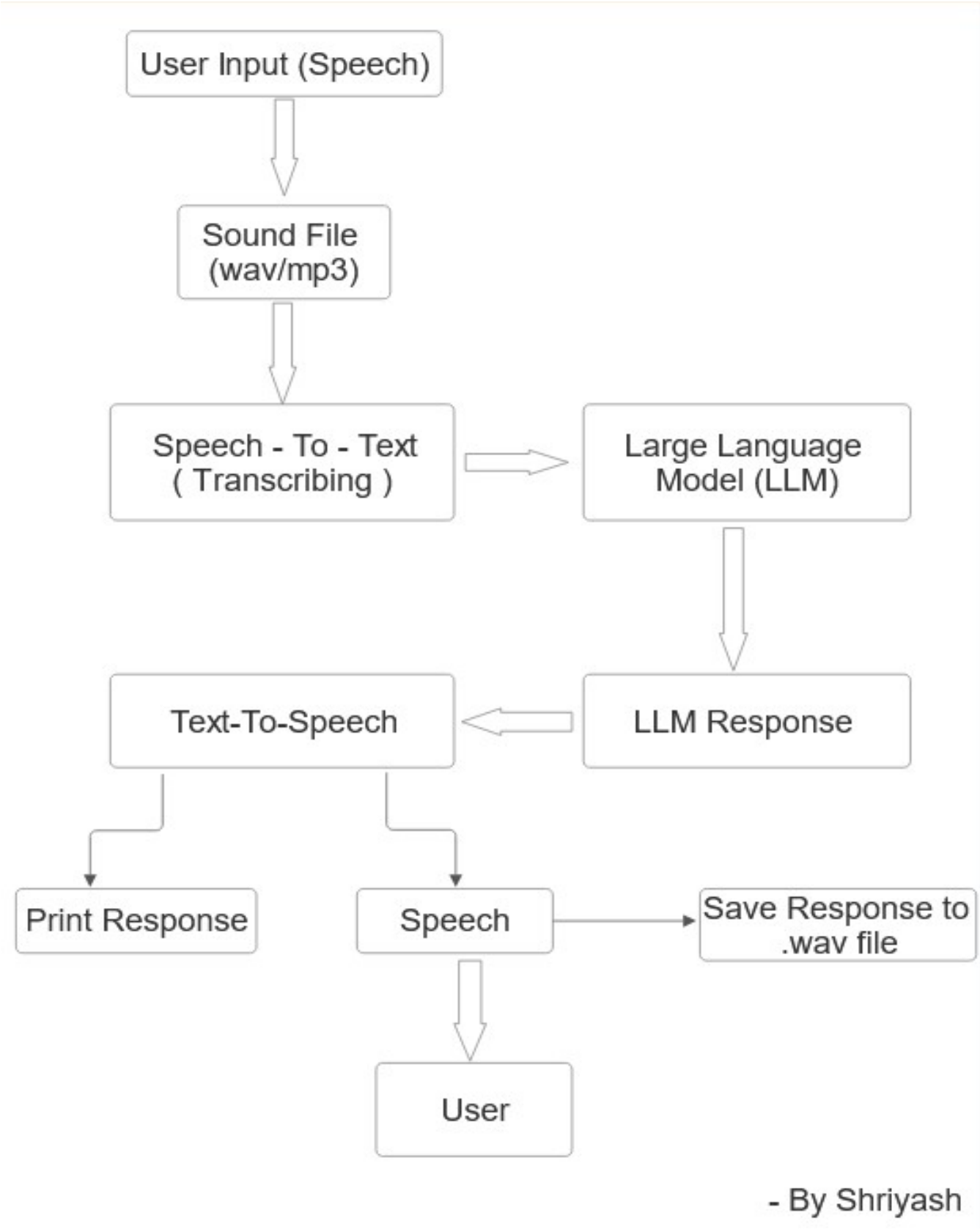
Hello, I am Shriyash Beohar. Here is how I solved the assignment. All the code and requirements are attached with the accompanying email. Please reach out in case of any further queries. Thanks!

Objective:

- To create a program which takes speech input, converts it into text, generate response to the query using LLM and then converts the LLM's response to speech.
- The solution must allow changing voice preferences (male/female), words per minute control (speed) etc.
- Use 16K sampling rate for user speech input.
- Use VAD to detect silence.
- Save the output as .wav.

Approach:

- Firstly, I take the voice query as input, convert it into .wav/.mp3 file and then transcribe it into text. I have used PyAudio to take the voice input and SpeechRecognition library for transcribing.
- The text obtained from STT model is then passed to the Large Language Model (LLM) via Groq API (llama3-70b-8192) to generate a reply to the user query.
- Since response of the LLM is in text format, we'll apply Text-To-Speech model to convert the response back into speech and play it back to the user.
- This response is also stored in a .wav/.mp3 file which is stored locally.
- Additionally, a conversation history file is maintained such that the Voice Assistant can recall context of a conversations while generating the reply.
- Here is a block diagram which explains the pipeline.


User Input (Speech)
Sound File (wav/mp3)
Speech - To - Text (Transcribing)
Large Language Model (LLM)
LLM Response
Text-To-Speech
Print Response
Speech
Save Response to .wav file
User
- By Shriyash

Working :

Step-0: Install the requirements from requirements.txt `pip install –r requirements.txt`. On Windows OSes you might need to install MVSC C++ Build Tools and SDK to build the WebRTCVAD library during pip installation.

Step-1: On running the voiceai.py file (`python voiceai.py`), the start() function will be called which initialized the history and welcomes the user with a message. The start function creates an empty '*history_db.json*' file to store the conversation. Here, system message for the LLM is set which defines the behavior of the LLM throughout the conversation.

```
    with open("history_db.json", 'r') as f:
        history = json.load(f)

    if id not in history:
        next_key = len(list(history.keys()))+1
        with open("history_db.json", 'w') as f:
            history[next_key] = [{
                "role": "system",
                "content": system_Set
            }]
            json.dump(history, f)
        groq(next_key)
```

Step-2: groq() function is called to get response from LLM. The function loads the conversation history using the conversation ID, gets the completion from the API and converts it to speech. The function open the '*history_db.json*' file and passes the messages stored for the conversation's id (that is always set to '1' for convenience but it can be managed for multiple conversations easily) to the LLM. The program prints the reply from the LLM on the console and converts them to speech using `*pyttsx3*` a Text-To -Speech library which uses in built voices. The response of the LLM is printed, played as well as stored in an audio '*output.wav*' file.

```
def groq(id):
    with open('key1.json') as f:
        key = json.load(f)
        keys = key['key']
    client = Groq(api_key=keys)

    with open("history_db.json", 'r') as f:
        chat_history = json.load(f)
    message = chat_history[str(id)]

    completion = client.chat.completions.create(
        model="llama3-70b-8192",
        messages=message,
        temperature=0.8,
        max_tokens=1024,
        top_p=1,
        stream=True,
        stop=None,
    )

    answer = """"""
```

```
    for chunk in completion:
        response = chunk.choices[0].delta.content if hasattr(
            chunk.choices[0].delta, 'content') else None
        answer += response if response is not None else ""
    system = p.init()
    rate = system.getProperty('rate')
    system.setProperty(rate, 170)

    system.say(answer)
    print(answer)

    system.save_to_file(answer, "output.wav")
    system.runAndWait()
```

Step-3: The main loop listens for user input (voice) and send it to the continue_conv() function for chat completion and reply generation. The VAD waits for 1 sec of silence and breaks the input to process the query. This can be improved as per the application. Once the user sends a query and receives the reply the user can simply ask another query to continue the conversation.

```
        try:
            while True:
                data = stream.read(480)
                is_speech = vad.is_speech(data, sample_rate)
                print(f"Contains speech: {is_speech}", end='\r')
                if is_speech:
                    continuous_silence_count = 0
                else:
                    continuous_silence_count += 30  # 480 bytes == 30ms at 16KHz
                    if continuous_silence_count >= SILENCE_THRESHOLD_MS:
                        raise (StopIteration)

                chunks.append(data)
        except (StopIteration, KeyboardInterrupt):
            print("Recording stopped.")
```

Step-4: The continue_conv() functions dumps the user query into the conversation file and calls groq() to produce and play the LLM response.

```
def continue_conv(id, prompt):
    with open("history_db.json", 'r') as f:
        message_prompt = json.load(f)

        if str(id) not in message_prompt:
            message_prompt[str(id)] = []

        x = {"role": "user",
             "content": prompt}
        message_prompt[str(id)].append(x)

    with open("history_db.json", 'w') as f:
        json.dump(message_prompt, f)
    groq(1)
```

Design Choices:

1. Speech to text : SpeechRecognition: https://pypi.org/project/SpeechRecognition/ :
   It is a lightweight Speech to text library which uses Google Speech Recognition API. Whisper and OpenAI API for Whisper can also be used if available. I chose this library as it makes it easy to include STT capabilities without any significant setup required and is quite fast. Alternatives are using a custom Whisper / any other STT model or APIs.
2. LLM API: groq : https://groq.com/ : Groq is free and fast and supports open source models. I used llama-70b, but any other model can be easily used. Alternatives could be OpenAI, Claude, Gemini or even local models.
3. VAD : WebRTCVAD : https://github.com/wiseman/py-webrtcvad/tree/master : Google's RTC VAD, while it only detect noise and not actual speech, it is fast and does the job and is also inexpensive to run. Not to mention that it is also very portable across various platforms as it is written in C++. A better alternative could be silero-vad but it runs slow on CPU. VAD helps in handoff interaction by the user, which would otherwise require user to press a button after speaking his query and degrade the user experience.
4. TTS : pyttsx3 : https://pypi.org/project/pyttsx3/ : It runs offline and is quite fast as well. A better and more engaging alternative could be using speech generation models.

Changing Parameters and Additional Requirements:

1. The voice (gender, pitch, speed) can be controlled by using different settings available in PyTTSx.
2. The most recent speed input and output is saved in .wav files.
3. VAD is implemented using WebRTCVAD.
4. LLM output is short and can be further tweaked using the system message in the start() function.

Alternative Attempts:

Due to lack of external GPU, Speech Recognition with faster-Whisper was not implement. Although , I tried running on Google Colab but again I faced a little problem in enabling microphone as Google Colab doesn't support *'pyaudio'* . I had to manually upload voice queries, which affects the latency and UX. Thus the main advantage of '*faster-whisper'* was unachievable for me in this setting.
Code : https://colab.research.google.com/drive/1trsxvHilNO5PN2nPTlFMOd2A56ClqASu?usp=sharing

Possible next steps / improvements:

1. Improve voice model / Speech to text. Currently the voice is quite robotic, latest models can greatly enhance user experience, but it will add significant latency and compute requirements.
2. Improve VAD . WebRTC VAD is quite old (but capable). It doesn't actually detect voice activity but rather checks for noise and hence susceptible to false positives. silero-vad is a better alternative but it requires GPU for lower latency.
3. Use better LLM based on requirements.
4. Improve control loop. The control flow with VAD can be refined along with LLM's response especially with exit phrases and conversation nuances. The LLM can be instructed to return response in a JSON with special fields to manage the main loop.
5. Bug fixes.