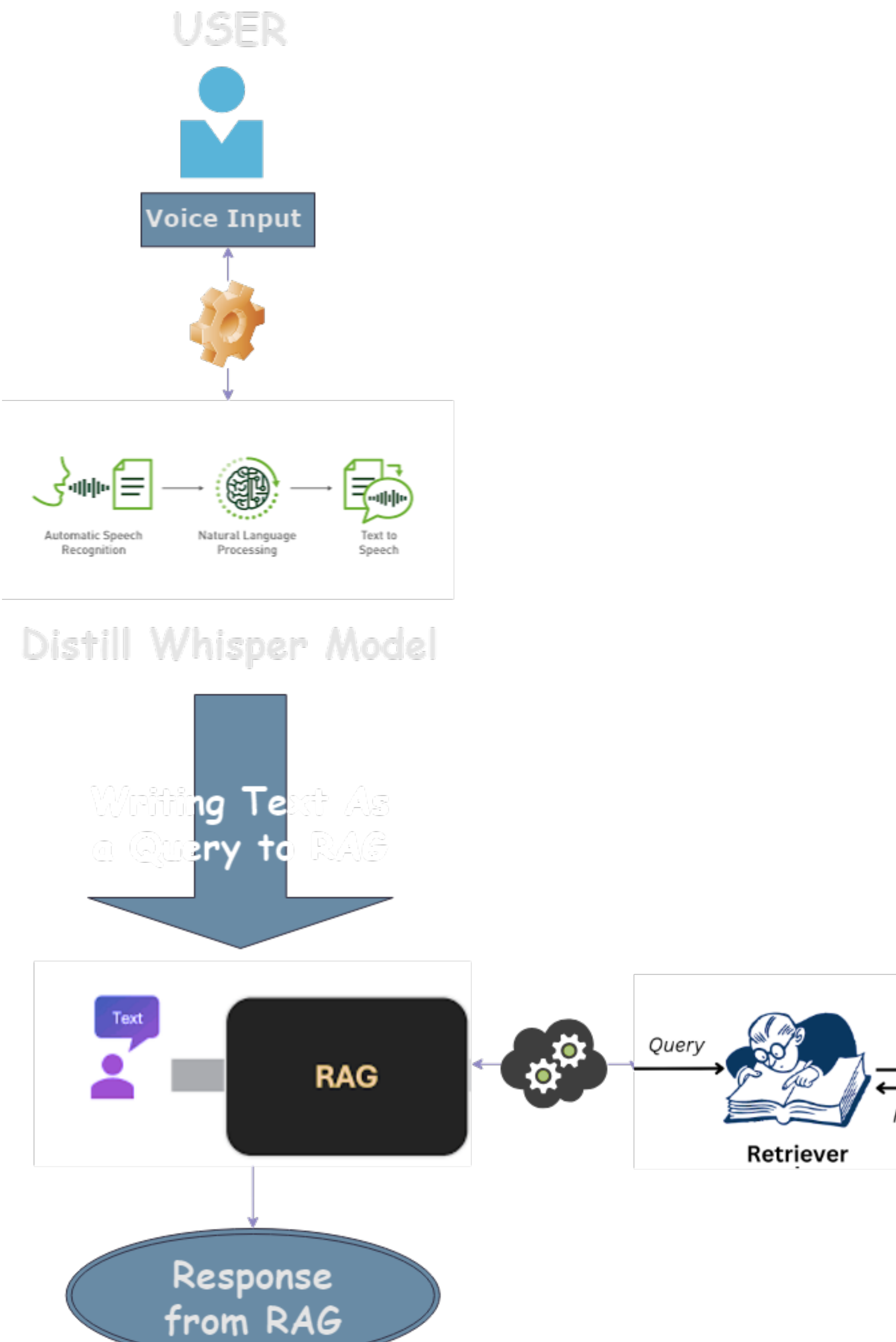
This project involves designing an End-to-End AI Voice Assistance Pipeline

Objective:

Design a pipeline that takes a voice query command, converts it into text, uses a Large Language Model (LLM) to generate a response, and then converts the output text back into speech. The system should have low latency, Voice Activity Detection (VAD), restrict the output to 2 sentences, and allow for tunable parameters such as pitch, male/female voice, and speed.

Architecture:



USER
Voice Input
Automatic Speech Recognition
Natural Language Processing
Text to Speech
Distill Whisper Model
Writing Text As a Query to RAG
Text
RAG
Query
Retriever
Response from RAG

AIVA drawio

Approach

The goal is to implement VOICE-TO-VOICE Assistance Pipeline using various open source models, combing their power building a voice Assistance Tool.

Step 1: Voice-to-Text Conversion

Here we are using `distil-whisper/distil-medium.en` for conversion of speech to text Efficiently.

[distll-whisper]()

Why only distill-whisper?

1. Able to Convert the text faster and with less memory compared to Wishper AI and its quantized models
2. Comsumes less computation power
3. State of Art WRE of 8 for `medium-en` model

Here is the code walkaround of Step-1

Defining a function to convert the voice-to-text data

```
def convert_text(self,input_file):
    logging.info('Audio file entered conversion method')

    try:

        device = "cuda:0" if torch.cuda.is_available() else "cpu"
        torch_dtype = torch.float16 if torch.cuda.is_available()
else torch.float32

        model_id = "distil-whisper/distil-medium.en"

        model = AutoModelForSpeechSeq2Seq.from_pretrained(
            model_id, torch_dtype=torch_dtype,
low_cpu_mem_usage=True, use_safetensors=True
        )

        model.to(device)

        processor = AutoProcessor.from_pretrained(model_id)
        logging.info("Model is set")
```

```
        pipe = pipeline(
            "automatic-speech-recognition",
            model=model,
            tokenizer=processor.tokenizer,
            feature_extractor=processor.feature_extractor,
            max_new_tokens=128,
            chunk_length_s=15,
            batch_size=16,
            torch_dtype=torch_dtype,
            device=device,
        )


        logging.info('Converison Initiated...')


        result = pipe(input_file)
        text_data = result['text']
        logging.info('Voice successfully Converted to text')


        folder_path = self.file_path.output_file_path

        # Create the folder if it doesn't exist
        if not os.path.exists(folder_path):
            os.makedirs(folder_path)

        # Save the file in the created folder
        with open(os.path.join(folder_path, "output.txt"), "w")
as file:
            file.write(text_data)

        logging.info('Text file saved successfully')

        return text_data

    except Exception as e:
        raise CustomException(e,sys)
```

To save the file in a specific folder we have defined a class variable:

```
output_file_path = str=os.path.join('query_data')
```

Step 2: Building RAG using llama-2 and llamaIndex

meta-llama-2

Why RAG?

We could have directly used any text generation llms such as langchain or mistral-7b but llms results in irrelevant response that may be out of context.
Due to this our responses may to not be up to the mark. By implementing RAG using LLamaIndex (optimal for retrival and indexing operations) We make sure thwe outputs are accurate to the query

RAG document: **human_01**

Employing LlamaIndex for Retrival operation combining both llama-2 and langchain

The use of two model results in better and efficient query results

We wrap both the models using Service Context

Service Context

The input to the ** RAG ** are populated to the both Langchain and Llama-2 usign Embedding

Embedding play a major role: Embedding are vectorized form of the textual data that are populated as input to the llm

Here’s the code overview for RAG Application usign ’ Llama_index framework, Llama-2 and Langchain model `

```
def __init__(self):
    self.output_path = RagConfig()
```

```
def get_query_output(self):

    try:
        logging.info("Entered RAG playground")

        with open("query_data/output.txt",'r') as file:
            user_query = file.readlines()

        documents = SimpleDirectoryReader(input_dir="RAG_data")

        logging.info("Document loading...")
        documents = documents.load_data()
        logging.info("Document Successfully loaded")

        system_prompt = """
            You are a Question&Answer assistant, your goal is to
answer questions based on instructions and context provided.
                            """

        prompt_format = SimpleInputPrompt("<|USER|>{user_query}<|
ASSISTANCE|>") #format for LLamaIndex



        # uploading the model Here we can use any model, each
model has its own parameters
        llm = HuggingFaceLLM(
            context_window=4096,
            max_new_tokens=256,
            generate_kwargs={"temperature": 0.0, "do_sample":
False},
            system_prompt=system_prompt,
            query_wrapper_prompt=prompt_format,
            tokenizer_name="meta-llama/Llama-2-7b-chat-hf",
            model_name="meta-llama/Llama-2-7b-chat-hf",
            device_map="auto",
        )

        """
         Embeddding is important as it transforms the textual
data in vectors efficiently
         we import embedding for langchain and llama-2
         Service Context to wrap both the llm making it more
efficient
```

```
        """


        # successfully added embedding to model
        embed_model = LangchainEmbedding(
            HuggingFaceEmbeddings(model_name="sentence-
transformers/all-mpnet-base-v2"))
        # this  model efficiently maps sentences to para -->
dense vector space

        # These Sparse vectors are populated with information and
can be efficiently stored
        service_context = ServiceContext.from_defaults(
            chunk_size=1024,
            llm=llm,
            embed_model=embed_model
        )

        logging.info("llms")
        index = VectorStoreIndex.from_documents(documents,
service_context=service_context)

        query_engine = index.as_query_engine()

        logging.info("Preparing the query output")

        response = query_engine.query("hey man how are you
doing?")

        logging.info("Result generated successfully")

        return response

    except Exception as e:
        raise CustomException(e,sys)
```

Here comes the final step Step 3: Converting RAG output to Speech

This is implementing by using Parler-Text-to-Speech model

Parler-tts

We are building this stage, Partial implemented

```
def convert_to_speech(self, input_filepath:str, tone:str):

    try:

        input_file = open_file(input_filepath)
        device = "cuda:0" if torch.cuda.is_available() else "cpu"

        model =
ParlerTTSForConditionalGeneration.from_pretrained("parler-tts/
parler-tts-mini-v1").to(device)
        tokenizer = AutoTokenizer.from_pretrained("parler-tts/
parler-tts-mini-v1")

        tone = str(tone.lower())
        if tone == "female":
            description = "A female speaker delivers a slightly
expressive and animated speech with a moderate speed and pitch.
The recording is of very high quality, with the speaker's voice
sounding clear and very close up."
        elif tone == "male":
            description = "A male speaker delivers a slightly
expressive and animated speech with a moderate speed and pitch.
The recording is of very high quality, with the speaker's voice
sounding clear and very close up."

        else:
            print(f'Invalid Input {tone}')


        input_ids = tokenizer(description,
return_tensors="pt").input_ids.to(device)
        prompt_input_ids = tokenizer(input_file,
return_tensors="pt").input_ids.to(device)

        generation = model.generate(input_ids=input_ids,
prompt_input_ids=prompt_input_ids)
        audio_arr = generation.cpu().numpy().squeeze()
```

```
        audio_data = sf.write("speech.wav", audio_arr,
model.config.sampling_rate)

        return audio_data

    except Exception as e:
        raise CustomException(e,sys)
```

This is the partial code.

Things to update

1. **Resampling of the audio file**

```
<li> Efficient code for selecting mulitple voice-overs <br>
</li>
<li> Saving the processed file <br> </li>
```

Things to cover

1. **Building Flask Server to receive/send GET and PUT requests**
2. **Interactive UI**
3. **Deploying the Entire project on cloud (AWS,AZURE,GCP)**