T. C.
USKUDAR UNIVERSITY
FACULTY OF ENGINEERING AND NATURAL SCIENCES

GRADUATION THESIS

“Friendly AI Powered Voice Assistant”

Thesis Advisor

Dr. Öğr. Üyesi AHMET ŞENOL

SLAYMANE ABDUL HAMID - 200201942

HASSAN ISSA - 200201331

2023 - 2024

T. C.
ÜSKÜDAR ÜNİVERSİTESİ
MÜHENDİSLİK VE DOĞA BİLİMLERİ FAKÜLTESİ

BİTİRME TEZİ

“Friendly Yapay Zeka Sesli Asistan”

Tez Danışmanı
Dr. Öğr. Üyesi AHMET ŞENOL

SLAYMANE ABDUL HAMID - 200201942

HASSAN ISSA – 200201331

2023 - 2024

Slaymane ABDUL HAMID and Hassan ISSA students of Üsküdar University, Faculty of Engineering and Natural Sciences, department of Computer Engineering. Students ID 200201942, 200201331 (respectively), successfully defended the thesis entitled ***"Friendly AI Powered Voice Assistant"***, which they prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below:

Thesis Advisor:Dr. Öğr. Üyesi AHMET ŞENOL
Üsküdar University

Jury Members: Dr. Öğr. Üyesi Gamze USLU
Üsküdar University

Dr. Öğr. Üyesi Belaynesh CHEKOL
Üsküdar University

Date of Submission : June 2024
Date of Defense : June 2024

FOREWORD

In this thesis we have prepared an AI Powered voice assistant. We would like to express special thanks to our advisor, Doctor Lecturer AHMET ŞENOL, for his help and interest in us.

TABLE OF CONTENTS

Page

SUMMARY

Friendly AI Powered Voice Assistant Web Application

This project introduces a voice-activated personal assistant designed to streamline daily tasks and information retrieval. Using their voices, users engage through natural language, leveraging advanced voice-to-text and text-to-speech technologies. The application is built on a modular architecture, incorporating **Deepgram** for voice transcription, **OpenAI's GPT-3** for intelligent response generation, and **ElevenLabs** for lifelike speech synthesis. A user-friendly web interface, powered by **Taipy**, displays conversations, and provides visual feedback. It seamlessly integrates with online information sources, offering up-to-date responses and enhancing productivity.

Key Words: *Voice Assistant, Natural Language Processing, Web Application, AI Integration, Productivity Tool*

ÖZET

Friendly Yapay Zeka Sesli Asistan Web Uygulaması:

Bu proje, günlük görevleri ve bilgi alımını kolaylaştırmak için tasarlanmış sesle etkinleştirilen bir kişisel asistanı tanıtmaktadır. Kullanıcılar, gelişmiş sesli metin ve metinden sese teknolojilerini kullanarak doğal dil aracılığıyla etkileşimde bulunurlar. Uygulama, modüler bir mimari üzerine kurulmuştur ve ses transkripsiyonu için Deepgram, akıllı yanıt üretimi için OpenAI'nin GPT-3'ü ve gerçekçi konuşma sentezi için ElevenLabs'i içermektedir. Taipy tarafından desteklenen kullanıcı dostu bir web arayüzü, konuşmaları görüntüler ve görsel geri bildirim sağlar. Çevrimiçi bilgi kaynaklarıyla sorunsuz bir şekilde entegre olarak güncel yanıtlar sunar ve verimliliği artırır.

Anahtar Kelimeler: *Sesli Asistan, Doğal Dil İşleme, Web Uygulaması, Yapay Zeka Entegrasyonu, Verimlilik Aracı*

1. Introduction:

All of us previously heard about *Alexa*, *Siri*, *Google Assistant* and many others, but for years we used these assistants while they lack the intelligence in their responses. Today, voice-controlled AI assistants represent a revolutionary shift in how users interact with technology. These systems leverage advanced speech recognition and natural language processing to facilitate seamless and intuitive communication between humans and machines. In today’s world, millions of people manage their daily tasks more efficiently using these AI assistants. They serve as a bridge that enables users to connect with various services through simple voice commands.

While day by day we are hearing more and more about *GPT*, *Gemini* and the upcoming *apple AI*, all these are not able to interact with the user and get back to them all through a spoken conversation on a PC, while they are able to do so on a mobile phone. For our project, we chose to integrate several technologies to create a sophisticated voice assistant. We utilized ***Deepgram*** for speech-to-text conversion, ***OpenAI's GPT-3*** API for natural language understanding, and ***ElevenLabs*** for text-to-speech synthesis, all orchestrated within a ***Python*** environment. These technologies are well-supported by modern AI and machine learning frameworks, making them ideal for developing an advanced voice-controlled system.

This thesis describes the development of a voice assistant program that allows users to interact using a microphone. The system processes voice input, converts it to text with ***Deepgram***, generates responses using ***OpenAI’s GPT-3***, and converts these responses back to speech with ***ElevenLabs***. The interaction is displayed in real-time on a web interface built with ***Taipy***. The entire system operates within a ***Python*** environment, ensuring compatibility and performance.

Users can perform various tasks through the assistant, such as asking questions, obtaining information, or just having a **FRIENDLY** chat. Setting up the system involves running **display.py** for the web interface and **main.py** for the voice assistant, providing a smooth user experience from voice input to spoken output.

In summary, by integrating these advanced technologies, we can modularize our system, separating components for different purposes. This approach enhances development and testing, reducing the potential for errors and improving maintainability. We will demonstrate the system's architecture and examine each component to ensure it functions optimally.

A-<u>Literature Review</u>

The development of voice-controlled AI assistants has evolved significantly since the inception of speech recognition technology. In the 1950s, Bell Laboratories introduced "Audrey," an early system that recognized spoken digits, aiming to bridge the gap between human speech and machine interpretation, as discussed by Davis, Biddulph, and Balashek in their 1952 paper, “Automatic Recognition of Spoken Digits.”

In the 1970s and 1980s, advancements like the Hidden Markov Model (HMM) improved speech recognition, as detailed by Rabiner in his 1989 article, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." Rabiner emphasized the importance of modular design, stating that "isolating functional units such as acoustic modeling, language modeling, and decoding improves system accuracy and adaptability."

The 1990s brought significant progress with natural language processing (NLP) algorithms, enabling systems to understand and generate human language more effectively. Jurafsky and Martin's 1996 work, "Speech and Language Processing," highlighted the integration of speech recognition and NLP as crucial for modern voice-controlled systems.

In the 2010s, deep learning revolutionized voice-controlled AI assistants. Hinton et al.'s 2012 paper, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," showcased the accuracy improvements through deep neural networks (DNNs). This period also saw the rise of commercially successful assistants like Siri, Google Assistant, and Alexa, utilizing DNNs for enhanced speech recognition and NLP.

Modern systems employ advanced speech-to-text and text-to-speech technologies integrated with powerful language models. Deepgram, for instance, uses deep learning for high-accuracy speech recognition, while OpenAI's GPT-3, enables sophisticated language understanding and generation. ElevenLabs' text-to-speech technology provides natural and expressive speech synthesis.

Voice-controlled AI assistants operate in a cycle: the user speaks a command, the system processes the speech, generates a response, and delivers it back to the user. This modular architecture ensures a seamless and efficient user experience.

The evolution of voice-controlled AI assistants incorporates advancements in speech recognition, NLP, and deep learning, culminating in the sophisticated systems we use today

2. ***Existing AI Voice Assistants (e.g., Alexa, Google Assistant, Siri)***

Voice-controlled AI assistants, such as Amazon's Alexa, Google Assistant, and Apple's Siri, have revolutionized the way people interact with technology. These assistants use advanced speech recognition and natural language processing to perform a variety of tasks, from setting reminders and answering questions to controlling smart home devices. Since their introduction, these systems have evolved significantly, offering users a more seamless and integrated experience.

The idea of voice assistants was popularized in the early 2010s, transforming how users interact with their devices. The rapid advancement in artificial intelligence and machine learning has enabled these assistants to understand and process natural language more accurately and respond in a more human-like manner. These advancements have made voice assistants an integral part of modern life, enhancing user productivity and convenience.

A- <u>Functionality Comparison</u>

Alexa, Google Assistant, and Siri each offer unique functionalities, but they share core capabilities like voice recognition, natural language understanding, and task execution. Alexa is known for its extensive integration with smart home devices and its vast library of skills developed by third parties. Google Assistant excels in search capabilities and contextual understanding, leveraging Google's extensive search database. Siri is deeply integrated with Apple's ecosystem, providing seamless functionality across Apple devices and services.

While all three assistants can perform basic tasks such as setting reminders, sending messages, and providing weather updates, their performance and user experience vary based on the ecosystem and user preferences. Alexa’s strength lies in home automation, Google Assistant excels in information retrieval, and Siri offers seamless integration with Apple’s hardware and software.

B-<u>Speech Recognition Technologies</u>

The success of voice assistants hinges on their speech recognition technologies. These systems use machine learning algorithms to convert spoken language into text. Alexa uses Amazon's proprietary ASR (Automatic Speech Recognition) technology, Google Assistant relies on Google’s Speech-to-Text API, and Siri uses Apple's in-house developed speech recognition engine. These technologies have evolved to understand various accents, dialects, and languages, improving the accuracy and usability of voice assistants globally.

C-<u>Text-to-Speech Synthesis Techniques</u>

- Text-to-speech (TTS) synthesis is another critical component of voice assistants, enabling them to convert text responses into spoken language.
- Alexa uses Amazon Polly, a service that turns text into lifelike speech.
- Google Assistant employs WaveNet, a neural network-based TTS developed by DeepMind, providing natural and expressive voice responses.
- Siri uses Apple's proprietary TTS engine, which has been refined to deliver a more human-like interaction.
- These TTS technologies have made significant strides in naturalness and expressiveness, enhancing user experience.

D-<u>Python Libraries for AI Development</u>

Python Libraries for AI Development

Several Python libraries are instrumental in developing AI voice assistants. **Deepgram** is a popular choice for speech-to-text processing, offering high accuracy and speed. **ElevenLabs** provides advanced TTS capabilities, producing natural and expressive speech. **Pygame** is a set of Python modules designed for writing video games but can be utilized for multimedia applications in voice assistant interfaces. **SpeechRecognition** is a library for performing speech recognition, supporting multiple engines and APIs.

These libraries simplify the development of voice-controlled AI systems by providing robust tools for speech recognition and synthesis. They enable developers to create sophisticated, responsive voice assistants that can understand and respond to a wide range of user inputs, making AI more accessible and effective for various applications.

3. <u>Project Content</u>

This project content section provides an in-depth look at the various advanced libraries and tools that will be utilized to build robust audio processing, transcription, and artificial intelligence applications. Each tool is essential for different aspects of the project, and this section aims to elaborate on their features, applications, and significance.

A-<u>PyAudio Library</u>

PyAudio is an essential library that provides Python bindings for PortAudio, a cross-platform audio input and output library. It is widely recognized for its simplicity and flexibility in handling audio streams, making it a cornerstone for projects that require real-time audio processing.

Core Features and Benefits:

Audio Capture: PyAudio facilitates the seamless recording of audio from various input devices, including microphones and line-in sources. This functionality is crucial for applications that need to capture live audio data, such as voice recognition systems, sound analysis tools, and audio logging applications.

Audio Playback: The library supports playback capabilities, allowing the project to output audio through speakers or headphones. This feature is vital for applications requiring audio feedback, such as interactive voice response systems, alert systems, and multimedia applications.

Stream Handling: PyAudio provides advanced stream management features, including non-blocking streams

that allow audio data to be processed in real-time without halting other operations. This is particularly important for applications that need to perform simultaneous tasks, such as real-time audio effects processing and live audio streaming.

Cross-Platform Compatibility: One of PyAudio’s key advantages is its ability to function across multiple operating systems, including Windows, MacOS, and Linux. This ensures that applications built with PyAudio can reach a broad audience without compatibility issues.
Use Cases in the Project:

In this project, PyAudio is utilized for its robust audio handling capabilities. For instance, in a voice-controlled application, PyAudio captures the user’s voice commands, which are then processed and interpreted by other integrated tools. Additionally, PyAudio’s playback functionality can be used to provide auditory feedback to users, enhancing interactivity and user engagement.

B-DeepGram

Deepgram is an advanced automatic speech recognition (ASR) platform that utilizes deep learning models to offer high-precision transcription services. Its sophisticated algorithms and efficient processing capabilities make it an invaluable tool for converting spoken language into text.

Key Features:

High Accuracy: Deepgram’s deep learning models are trained on vast amounts of diverse audio data, enabling them to achieve high
accuracy rates in transcription. This is particularly beneficial in noisy environments or when dealing with speakers who have various accents and speech patterns.

Real-Time Transcription: The platform supports real-time transcription, which is essential for applications that require live audio processing, such as customer service bots, live captions for events, and real-time analytics.

Scalability: Deepgram's infrastructure is designed to handle large volumes of audio data efficiently. This scalability is crucial for enterprise applications where high throughput and reliability are necessary.

Custom Models: Users can train custom speech models to improve accuracy for specific terminologies, industries, or dialects. This customization is valuable for applications in specialized fields like healthcare, legal, and technical support, where domain-specific language is common.

Applications in the Project:

Within this project, Deepgram is integrated to provide precise and efficient speech-to-text conversion. For example, in a voice-activated note-taking application, Deepgram transcribes the user’s spoken words into text in real-time, allowing for seamless interaction and accurate documentation. This integration significantly enhances the user experience by providing reliable and instant transcription.

C-__OpenAI__

OpenAI is a leading artificial intelligence research lab and deployment company known for its cutting-edge AI models, including the widely recognized GPT-4. These models excel in natural language processing (NLP) and generation,

offering a wide range of applications from chatbots to advanced data analysis.

Notable Features and Capabilities:

Natural Language Processing (NLP): OpenAI's models are capable of understanding and generating human-like text, making them ideal for applications that require nuanced language understanding, such as virtual assistants, customer support bots, and content creation tools.

Contextual Understanding: These AI models can comprehend context and maintain coherent conversations, which is crucial for applications that need to handle complex queries or provide detailed explanations.

Code Generation and Debugging: OpenAI's models can assist developers by generating code snippets, providing debugging suggestions, and explaining code functionality. This feature can streamline the development process and improve productivity.

Creative Content Generation: The ability to generate creative content such as articles, stories, and marketing copy is another strength. This capability can be used to automate content creation, thereby saving time and resources.

Integration in the Project:

In this project, OpenAI's language models are leveraged to enhance various functionalities. For instance, in a chatbot application, OpenAI provides intelligent and context-aware responses, making interactions more natural and engaging. Additionally, OpenAI's models can be used to analyze user input and generate insights, which can be valuable for data-driven decision-making processes.

D-__ElevenLabs__

ElevenLabs is a technology company specializing in creating high-quality synthetic voices using advanced text-to-speech (TTS) technology. Their platform provides realistic and natural-sounding voice generation, essential for applications requiring human-like speech synthesis.

Key Advantages:

Natural Sounding Voices: ElevenLabs uses cutting-edge algorithms to produce voices that closely mimic human speech. This naturalness enhances the user experience in applications such as virtual assistants, audiobooks, and automated announcements.

Customization: Users can customize various aspects of the generated voices, including tone, pitch, and speaking style. This customization allows for tailored voice outputs that suit specific brand personalities or user preferences.

Multi-Language Support: The platform supports multiple languages, making it versatile for global applications and allowing for the creation of multilingual interactive systems.

Efficiency: ElevenLabs' technology is designed for quick and efficient voice generation without compromising on quality. This ensures that applications requiring real-time voice synthesis can operate smoothly.

Role in the Project:

In this project, ElevenLabs is used to generate synthetic speech, adding an auditory dimension to the applications. For example, in an educational app, ElevenLabs can convert text-based lessons into spoken word, making the content accessible to users who prefer auditory learning or have visual impairments. This integration enhances inclusivity and user engagement.

E- PyGame

PyGame is a comprehensive library for writing video games in Python, but its capabilities extend far beyond gaming. It provides tools for handling graphics, sound, and user input, making it a versatile choice for multimedia applications.

Key Features:

Game Development: PyGame simplifies the creation of games with its extensive collection of modules

F- <u>TaiPy</u>

TaiPy is an open-source Python library designed to simplify the creation of production-ready applications with intuitive interfaces and powerful backend capabilities. It offers tools for developing user-friendly dashboards and interactive web applications, making it an excellent choice for projects that require robust data presentation and interaction.

Key features of TaiPy include:

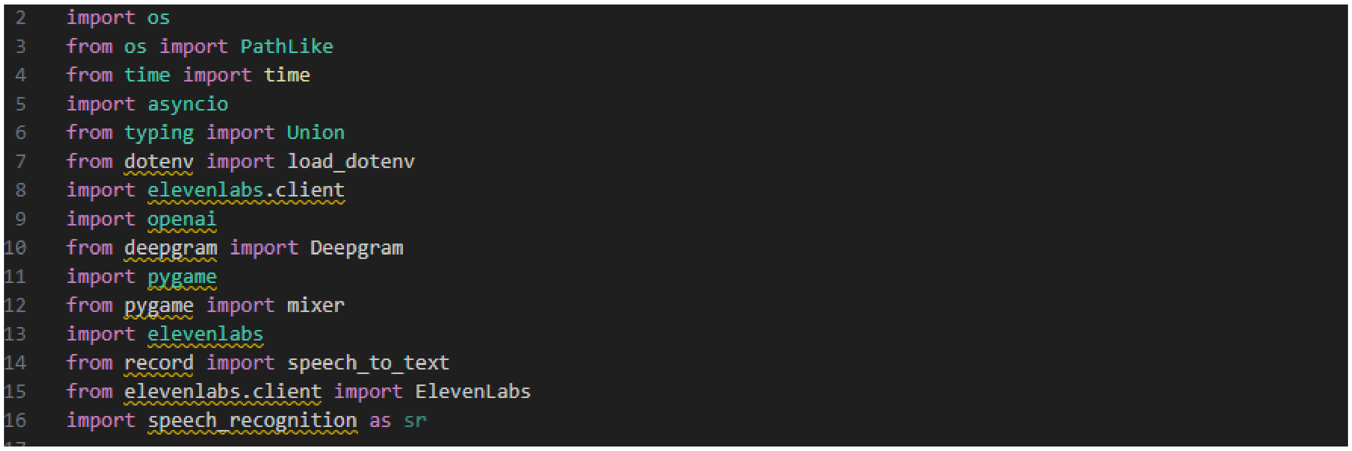
User Interface Design: TaiPy provides a range of components and tools to design intuitive and aesthetically pleasing user interfaces.
Data Visualization: The library supports advanced data visualization techniques, enabling developers to create insightful and interactive data representations.
Ease of Use: TaiPy is designed with simplicity in mind, allowing developers to quickly prototype and deploy applications without extensive coding.
Integration Capabilities: TaiPy can easily integrate with other Python libraries and tools, making it a versatile addition to the project.
By incorporating TaiPy, this project gains the ability to present data and results in a user-friendly and visually appealing manner, improving user engagement and comprehension

4. Code Explanation:

A- Main.Py:

1. Import:

Various libraries and modules are imported, including os, asyncio, dotenv, elevenlabs, openai, deepgram, pygame, and speech_recognition.



```python
import os
from os import PathLike
from time import time
import asyncio
from typing import Union
from dotenv import load_dotenv
import elevenlabs.client
import openai
from deepgram import Deepgram
import pygame
from pygame import mixer
import elevenlabs
from record import speech_to_text
from elevenlabs.client import ElevenLabs
import speech_recognition as sr
```

❖ Standard Libraries:

- **OS**: Provides functions to interact with the operating system, such as accessing environment variables and file paths.
- **time**: Provides time-related functions, used here to measure execution time.
- **asyncio**: Supports asynchronous programming, allowing the program to run transcriptions concurrently.
- **typing**: Provides type hints for better code readability and maintenance, specifically Union for type hinting.

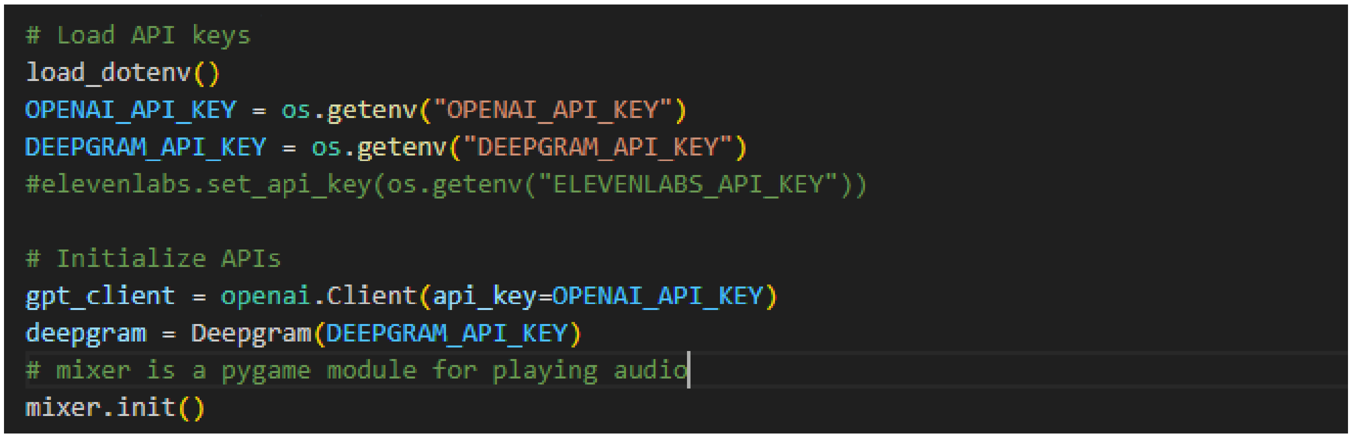
❖ External Libraries:

- **dotenv**: Used to load environment variables from a .env file into the application, ensuring API keys and other sensitive information are managed securely.
- **elevenlabs**: Client library for the ElevenLabs API, used to convert text to speech.
- **openai**: Client library for the OpenAI API, used to generate responses using GPT-3.
- **deepgram**: Client library for the Deepgram API, used for speech-to-text transcription.
- **pygame**: A set of Python modules designed for writing video games, used here for audio playback.
- **speech_recognition**: A library for performing speech recognition with various engines and APIs, used in record.py.

❖ Module Imports

- **speech_to_text**: Imports the speech_to_text function from record.py, which is responsible for recording and processing audio input.
- **ElevenLabs**: Imports the ElevenLabs client directly, which is used to interact with the ElevenLabs API for text-to-speech conversion.
- **mixer**: Specifically imports the mixer module from Pygame, which handles audio playback.

2. *APIs Setup:*



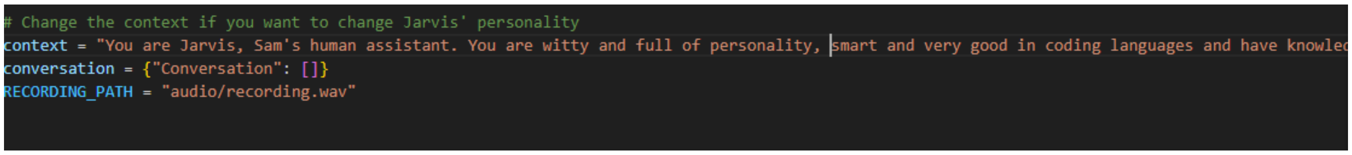
```
# Load API keys
load_dotenv()
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
DEEPGRAM_API_KEY = os.getenv("DEEPGRAM_API_KEY")
#elevenlabs.set_api_key(os.getenv("ELEVENLABS_API_KEY"))

# Initialize APIs
gpt_client = openai.Client(api_key=OPENAI_API_KEY)
deepgram = Deepgram(DEEPGRAM_API_KEY)
# mixer is a pygame module for playing audio
mixer.init()
```

This section of the script is responsible for:

- **Loads** environment variables from a .env file.
- **Retrieves** API keys for OpenAI and Deepgram from the environment variables.
- **Initializes** the OpenAI API client.
- **Initializes** the Deepgram API client.
- **Initializes** the Pygame mixer module for audio playback.

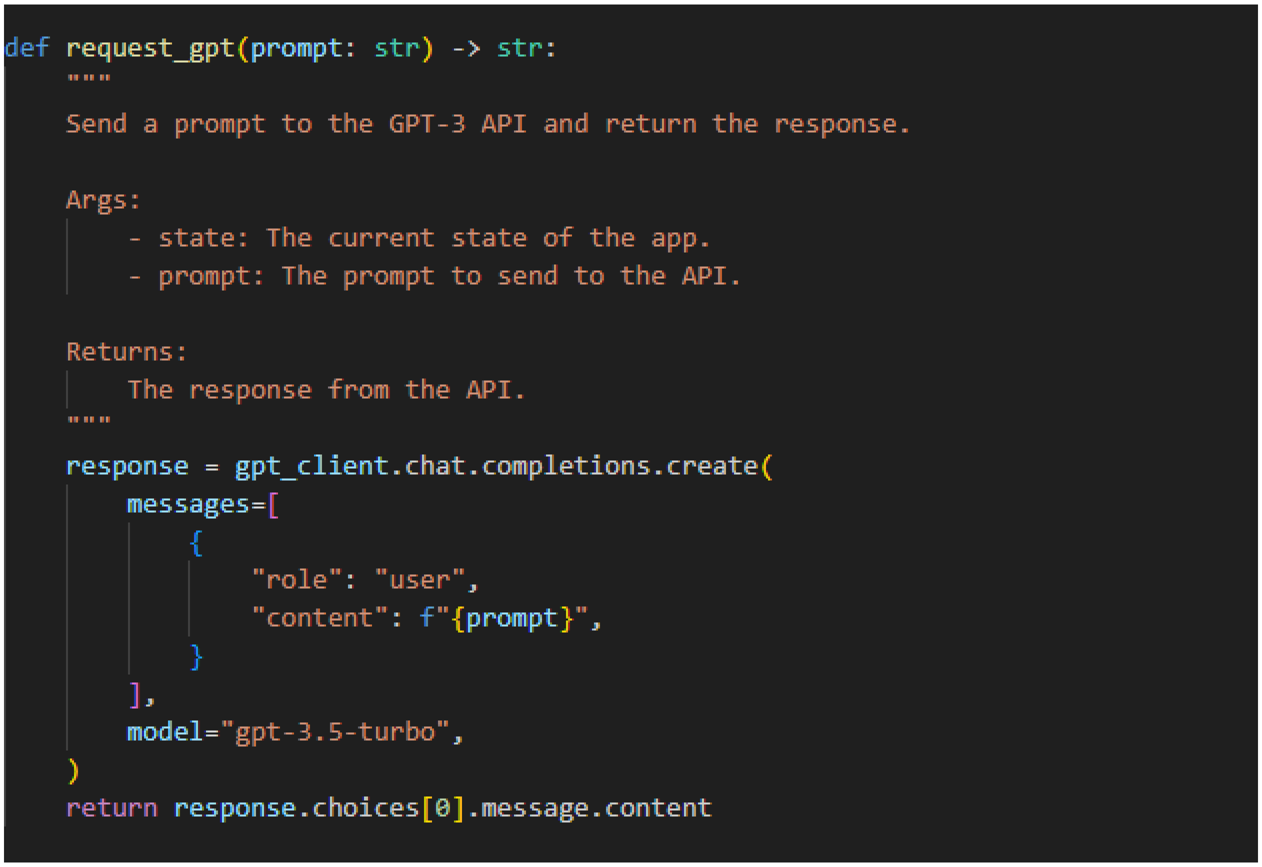
3. ***Set Context and Configuration:***



```
# Change the context if you want to change Jarvis' personality
context = "You are Jarvis, Sam's human assistant. You are witty and full of personality, smart and very good in coding languages and have knowled
conversation = {"Conversation": []}
RECORDING_PATH = "audio/recording.wav"
```

- **context**: Sets the initial context and personality for Jarvis. This defines how Jarvis will respond to user inputs.
- **conversation**: Initializes a dictionary to store the conversation history.
- **RECORDING_PATH**: Defines the file path where audio recordings will be saved.

4. ***Function request LLM***



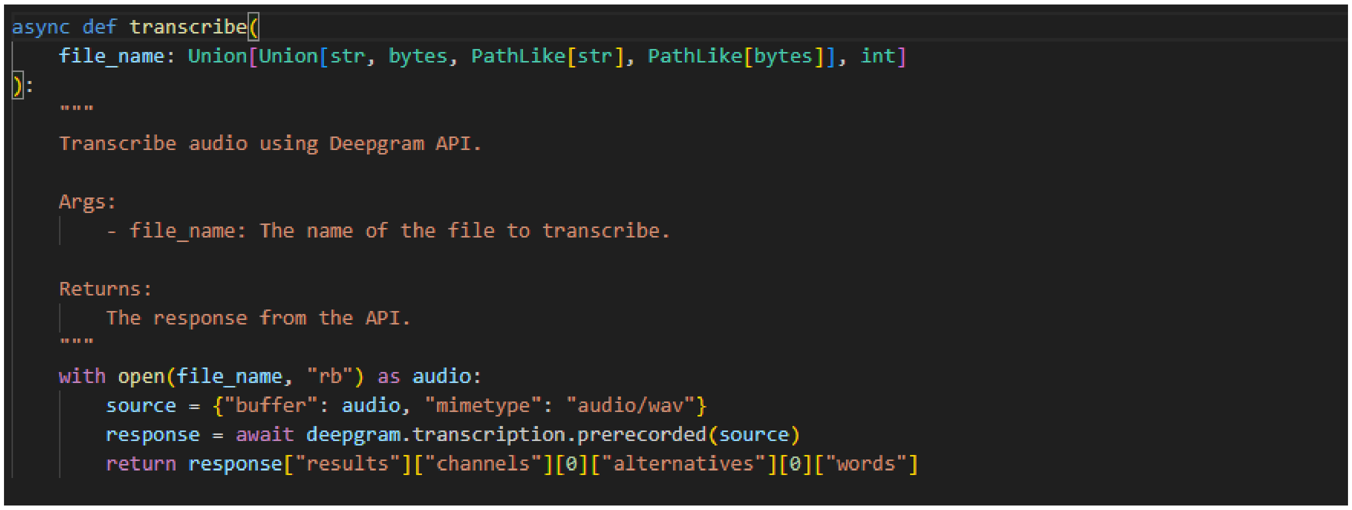
```python
def request_gpt(prompt: str) -> str:
    """
    Send a prompt to the GPT-3 API and return the response.

    Args:
        - state: The current state of the app.
        - prompt: The prompt to send to the API.

    Returns:
        The response from the API.
    """
    response = gpt_client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": f"{prompt}",
            }
        ],
        model="gpt-3.5-turbo",
    )
    return response.choices[0].message.content
```

- **Description**: Sends a user prompt to GPT-3 and returns the generated response.
- **prompt (str):** The user input or conversation context to send to GPT-3.
- **Returns**: The response text from GPT-3.

5. ***Function: transcribe***



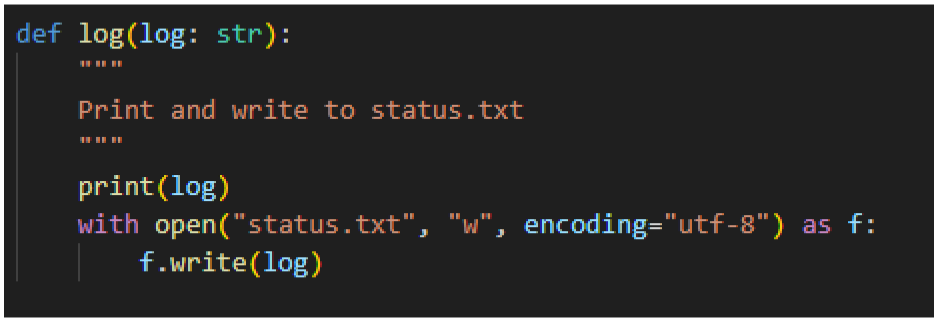
```python
async def transcribe(
    file_name: Union[Union[str, bytes, PathLike[str], PathLike[bytes]], int]
):
    """
    Transcribe audio using Deepgram API.

    Args:
        - file_name: The name of the file to transcribe.

    Returns:
        The response from the API.
    """
    with open(file_name, "rb") as audio:
        source = {"buffer": audio, "mimetype": "audio/wav"}
        response = await deepgram.transcription.prerecorded(source)
        return response["results"]["channels"][0]["alternatives"][0]["words"]
```

- **Description**: Asynchronously transcribes the given audio file using Deepgram API. **file_name:** The path to the audio file to transcribe.
- **Returns**: The transcribed words from the audio.

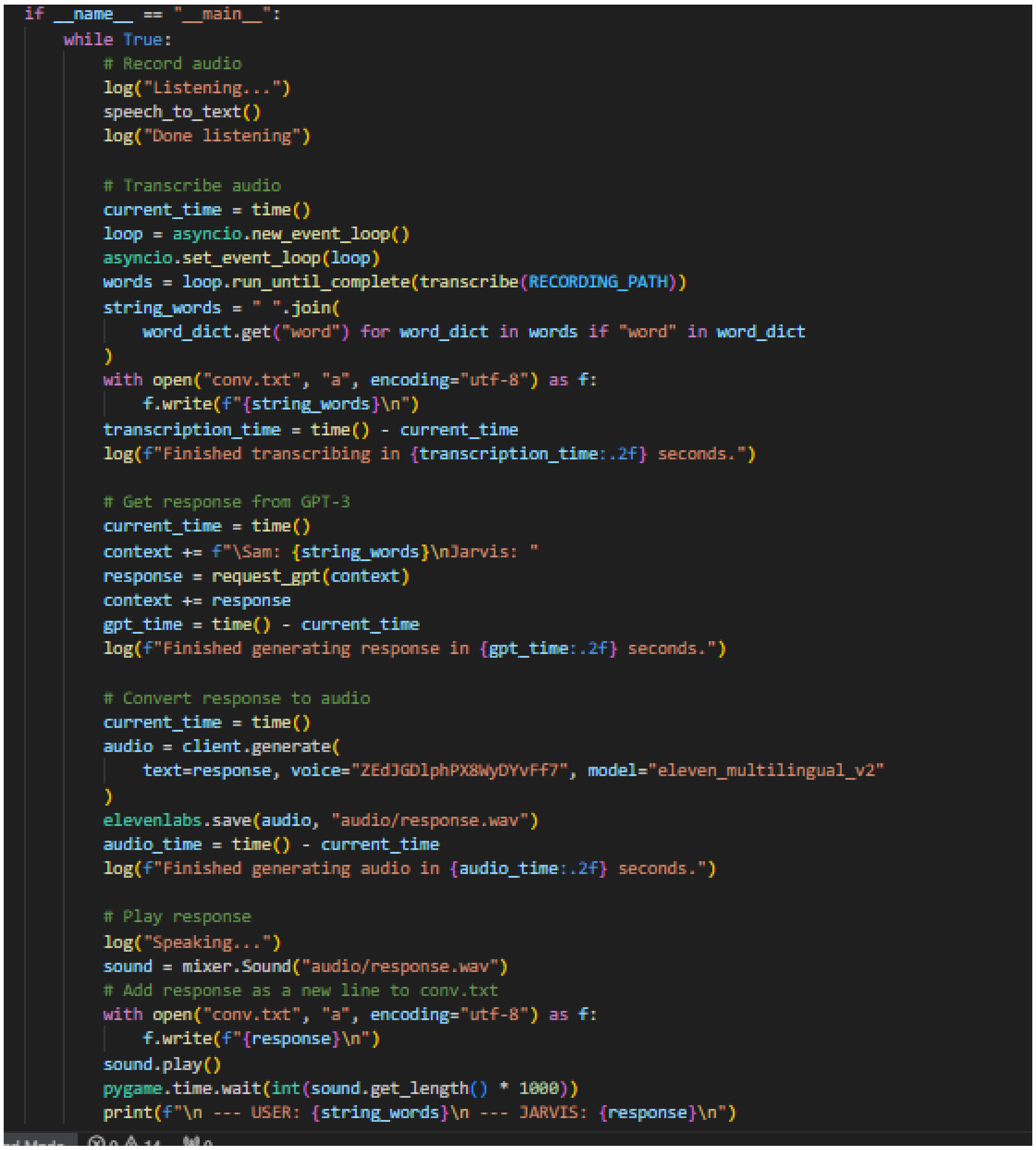
6. ***Function: log***



```python
def log(log: str):
    """
    Print and write to status.txt
    """
    print(log)
    with open("status.txt", "w", encoding="utf-8") as f:
        f.write(log)
```

- **Description**: Logs messages to both the console and writes them to a status file.
- **log (str):** The log message to record.

7. ***Main Execution Loop***



```python
if __name__ == "__main__":
    while True:
        # Record audio
        log("Listening...")
        speech_to_text()
        log("Done listening")

        # Transcribe audio
        current_time = time()
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        words = loop.run_until_complete(transcribe(RECORDING_PATH))
        string_words = " ".join(
            word_dict.get("word") for word_dict in words if "word" in word_dict
        )
        with open("conv.txt", "a", encoding="utf-8") as f:
            f.write(f"{string_words}\n")
        transcription_time = time() - current_time
        log(f"Finished transcribing in {transcription_time:.2f} seconds.")

        # Get response from GPT-3
        current_time = time()
        context += f"\Sam: {string_words}\nJarvis: "
        response = request_gpt(context)
        context += response
        gpt_time = time() - current_time
        log(f"Finished generating response in {gpt_time:.2f} seconds.")

        # Convert response to audio
        current_time = time()
        audio = client.generate(
            text=response, voice="ZEdJGDlphPX8WyDYvFf7", model="eleven_multilingual_v2"
        )
        elevenlabs.save(audio, "audio/response.wav")
        audio_time = time() - current_time
        log(f"Finished generating audio in {audio_time:.2f} seconds.")

        # Play response
        log("Speaking...")
        sound = mixer.Sound("audio/response.wav")
        # Add response as a new line to conv.txt
        with open("conv.txt", "a", encoding="utf-8") as f:
            f.write(f"{response}\n")
        sound.play()
        pygame.time.wait(int(sound.get_length() * 1000))
        print(f"\n --- USER: {string_words}\n --- JARVIS: {response}\n")
```

Description: The main loop continuously performs the following steps:

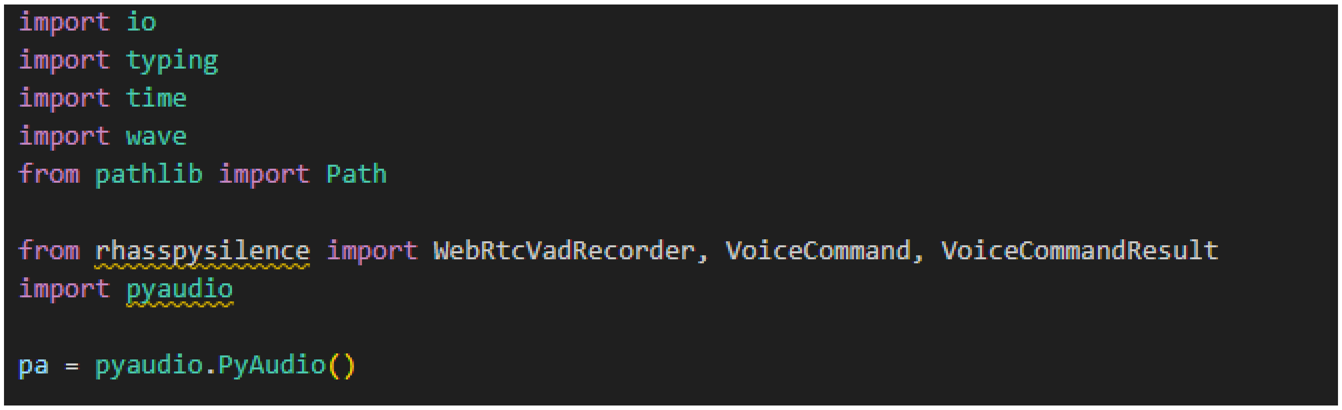
- **Record Audio**: Logs "Listening..." and calls speech_to_text to record audio input.
- **Transcribe Audio**: Logs "Done listening", then transcribes the audio using transcribe.
- **Generate GPT-3 Response**: Logs the transcription time, sends the transcribed text to GPT-3, and logs the time taken to generate a response.
- **Convert Text to Audio**: Converts the GPT-3 response to an audio file using ElevenLabs and logs the time taken.
- **Play Audio Response**: Logs "Speaking...", plays the generated audio using Pygame's mixer, and logs the user input and response.

Summary

This part of the code sets up the conversation context, defines functions for interacting with GPT-3 and Deepgram APIs, and implements the main loop for recording, transcribing, generating, and playing back responses. The process is meticulously logged at each step for monitoring and debugging purposes.

B-Record.py

1. Import:



```
import io
import typing
import time
import wave
from pathlib import Path

from rhasspysilence import WebRtcVadRecorder, VoiceCommand, VoiceCommandResult
import pyaudio

pa = pyaudio.PyAudio()
```

❖ Standard libraries:

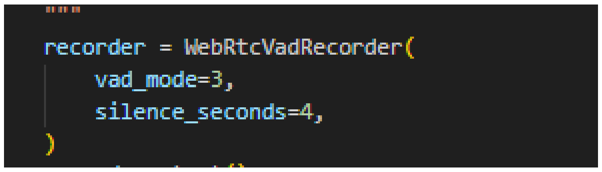
- **io**: Provides core tools for working with streams, used here for handling audio buffers.
- **typing**: Used for type hinting to ensure better code readability and maintenance.
- **time**: Provides time-related functions, used to generate timestamps for audio file names.
- **wave**: Provides a convenient interface to the WAV sound format, used for writing audio data to WAV files.
- **pathlib**: Provides an object-oriented interface for file system paths, used to handle file paths for saving audio.

❖ External libraries:

- **rhasspysilence**: Specifically *WebRtcVadRecorder*, *VoiceCommand*, and V*oiceCommandResult* are used for detecting voice activity and handling voice commands.
- **pyaudio**: Provides bindings for *PortAudio*, used here for accessing and managing audio hardware.

2. ***Function: speech_to_text:***

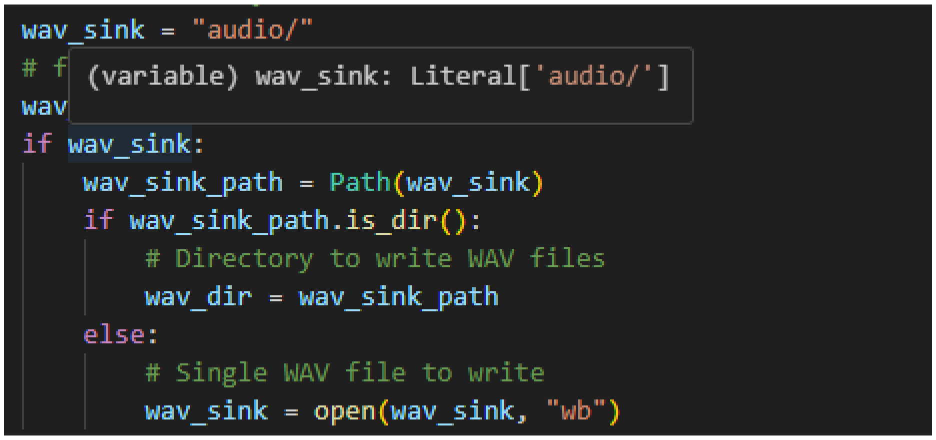
❖ Initialize Recorder:



```
"""
recorder = WebRtcVadRecorder(
    vad_mode=3,
    silence_seconds=4,
)
```

Configures the voice activity detector to use a high sensitivity mode and detects 4 seconds of silence to stop recording.

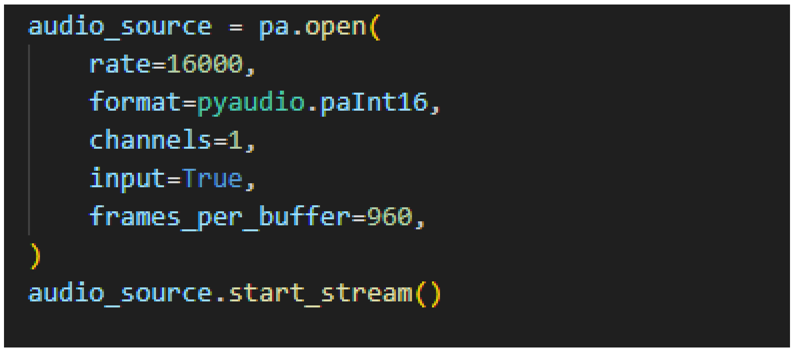
❖ Setup File Paths:



```
wav_sink = "audio/"
# f (variable) wav_sink: Literal['audio/']
wav
if wav_sink:
    wav_sink_path = Path(wav_sink)
    if wav_sink_path.is_dir():
        # Directory to write WAV files
        wav_dir = wav_sink_path
    else:
        # Single WAV file to write
        wav_sink = open(wav_sink, "wb")
```

Configures the directory and file name where the audio will be saved.

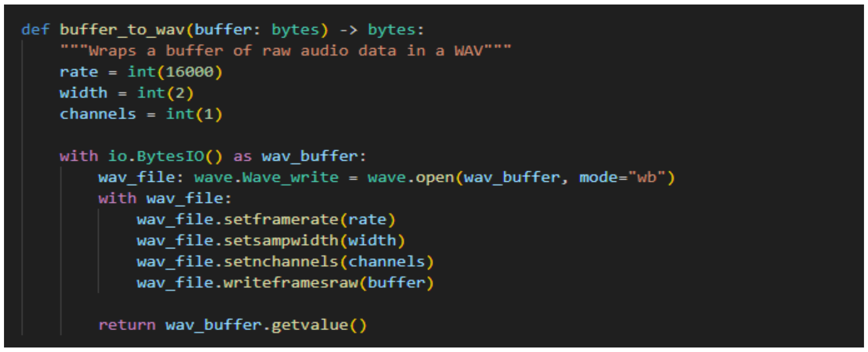
❖ Set Audio source:



```
audio_source = pa.open(
    rate=16000,
    format=pyaudio.paInt16,
    channels=1,
    input=True,
    frames_per_buffer=960,
)
audio_source.start_stream()
```

Opens an audio stream with a sample rate of 16,000 Hz, 16-bit samples, mono channel, and a buffer size of 960 frames.

❖ <u>Buffer to wav conversation:</u>



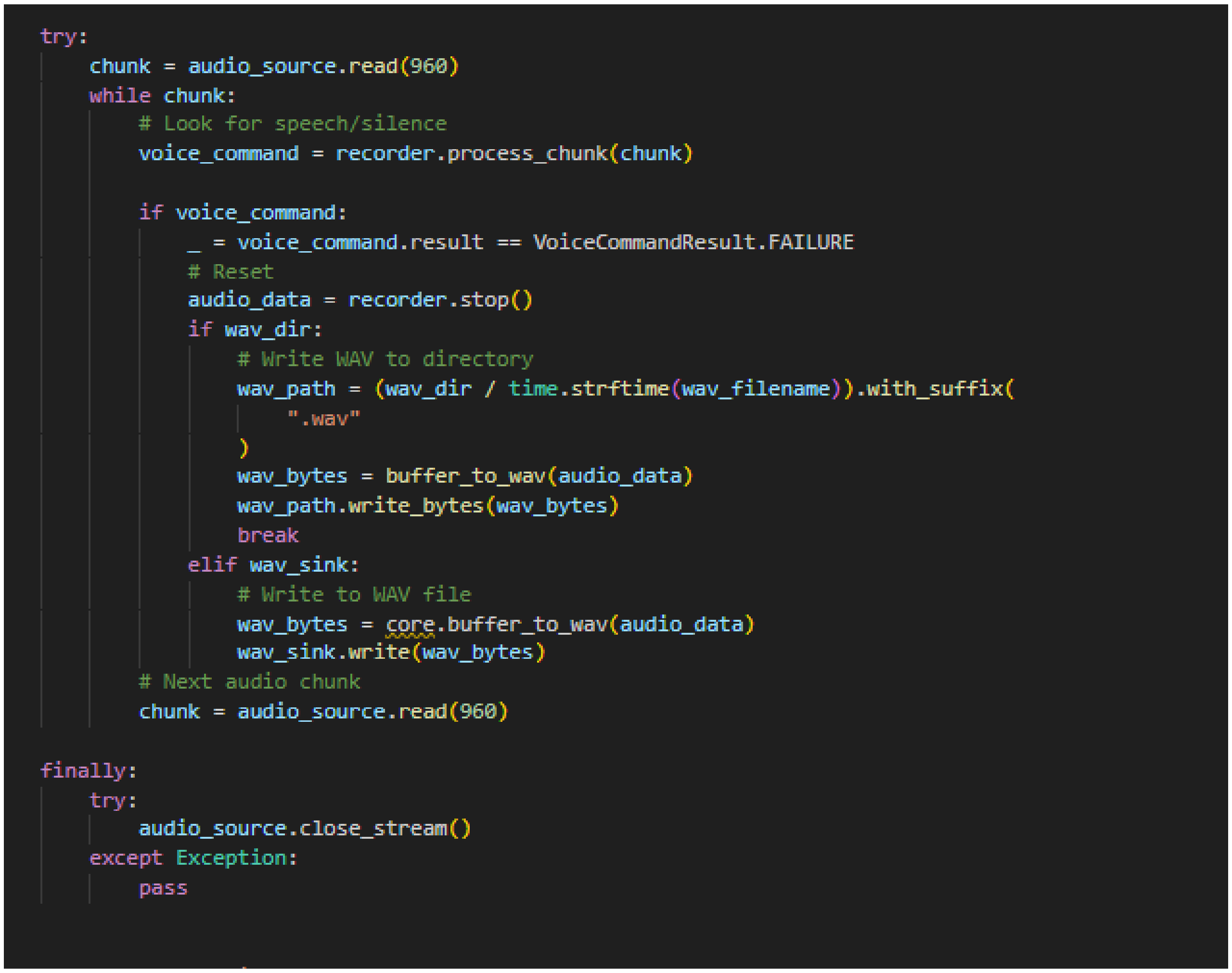
```python
def buffer_to_wav(buffer: bytes) -> bytes:
    """Wraps a buffer of raw audio data in a WAV"""
    rate = int(16000)
    width = int(2)
    channels = int(1)

    with io.BytesIO() as wav_buffer:
        wav_file: wave.Wave_write = wave.open(wav_buffer, mode="wb")
        with wav_file:
            wav_file.setframerate(rate)
            wav_file.setsampwidth(width)
            wav_file.setnchannels(channels)
            wav_file.writeframesraw(buffer)

        return wav_buffer.getvalue()
```

Converts raw audio data to WAV format.

❖ <u>Record and Process Audio:</u>



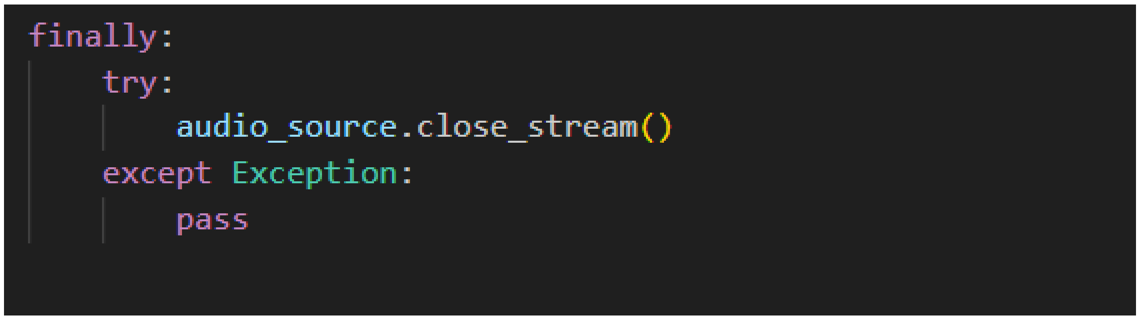
```python
try:
    chunk = audio_source.read(960)
    while chunk:
        # Look for speech/silence
        voice_command = recorder.process_chunk(chunk)

        if voice_command:
            _ = voice_command.result == VoiceCommandResult.FAILURE
            # Reset
            audio_data = recorder.stop()
            if wav_dir:
                # Write WAV to directory
                wav_path = (wav_dir / time.strftime(wav_filename)).with_suffix(
                    ".wav"
                )
                wav_bytes = buffer_to_wav(audio_data)
                wav_path.write_bytes(wav_bytes)
                break
            elif wav_sink:
                # Write to WAV file
                wav_bytes = core.buffer_to_wav(audio_data)
                wav_sink.write(wav_bytes)
        # Next audio chunk
        chunk = audio_source.read(960)

finally:
    try:
        audio_source.close_stream()
    except Exception:
        pass
```

Reads audio chunks, processes them for voice activity, and saves them as WAV files when speech is detected.

❖ Main execution:

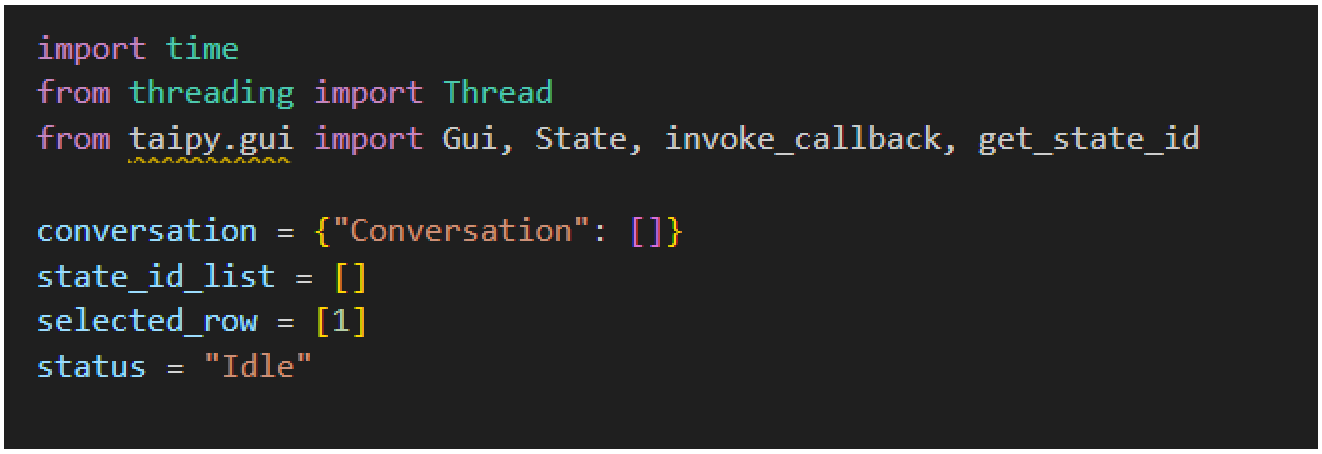


```
finally:
    try:
        audio_source.close_stream()
    except Exception:
        pass
```

If the script run directly, it will call the speech_to_text function to start recording audio.

C-Display.py

1. Import:



```
import time
from threading import Thread
from taipy.gui import Gui, State, invoke_callback, get_state_id

conversation = {"Conversation": []}
state_id_list = []
selected_row = [1]
status = "Idle"
```

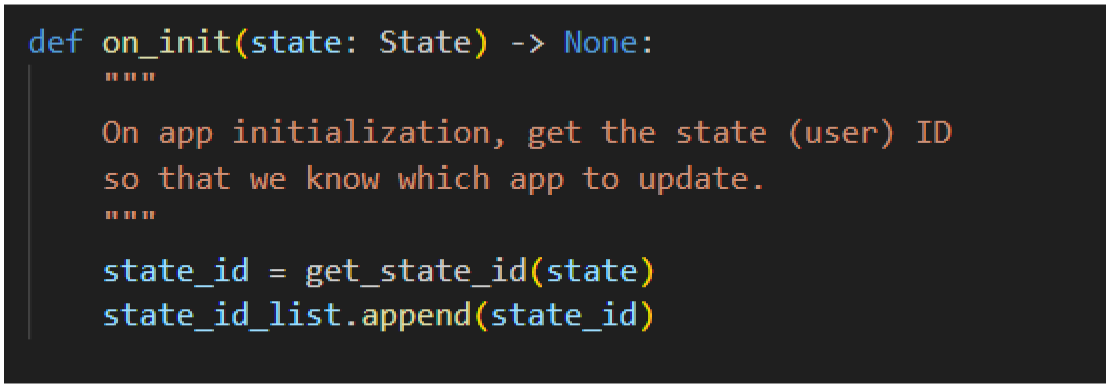
❖ Imports:

- time: Provides various time-related functions.
- Thread from threading: Allows the execution of code in separate threads, facilitating concurrent execution.
- Gui, State, invoke_callback, get_state_id from taipy.gui: Components from the Taipy GUI framework used to build and manage the application's user interface.

❖ Initializations:

- conversation: A dictionary to store conversation data.
- state_id_list: A list to keep track of state (user) IDs for updating the app.
- selected_row: A list to keep track of the currently selected row in the conversation table.
- status: A string to hold the current status of the application.

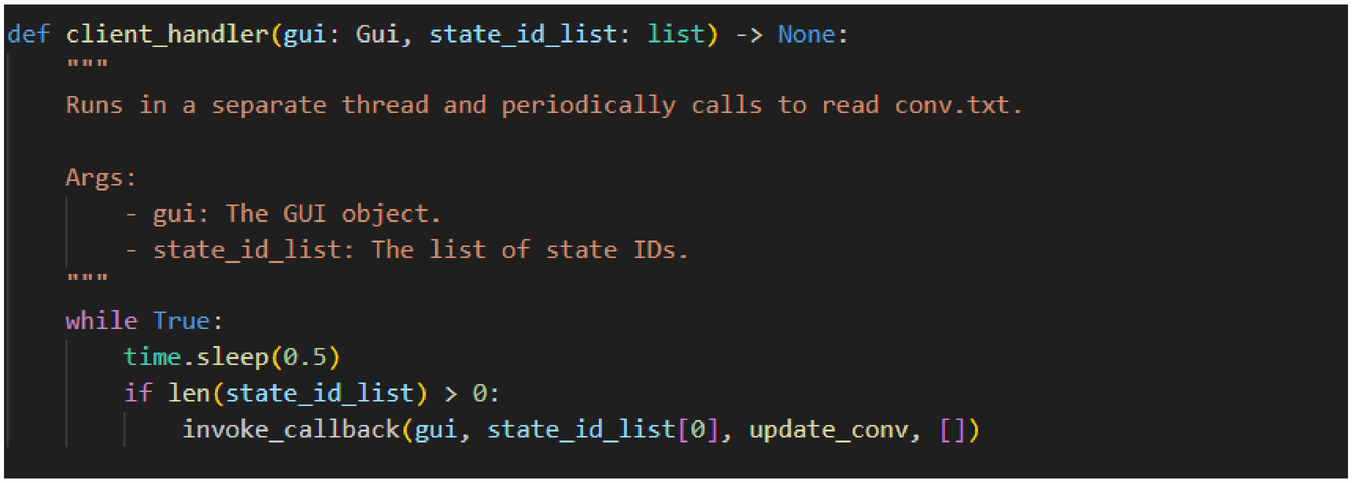
2. _on_init_ Function:



```python
def on_init(state: State) -> None:
    """
    On app initialization, get the state (user) ID
    so that we know which app to update.
    """
    state_id = get_state_id(state)
    state_id_list.append(state_id)
```

This function runs during the initialization of the app. It retrieves the state (user) ID and appends it to the state_id_list. This helps in identifying which instance of the app to update.

3. client handler Function:

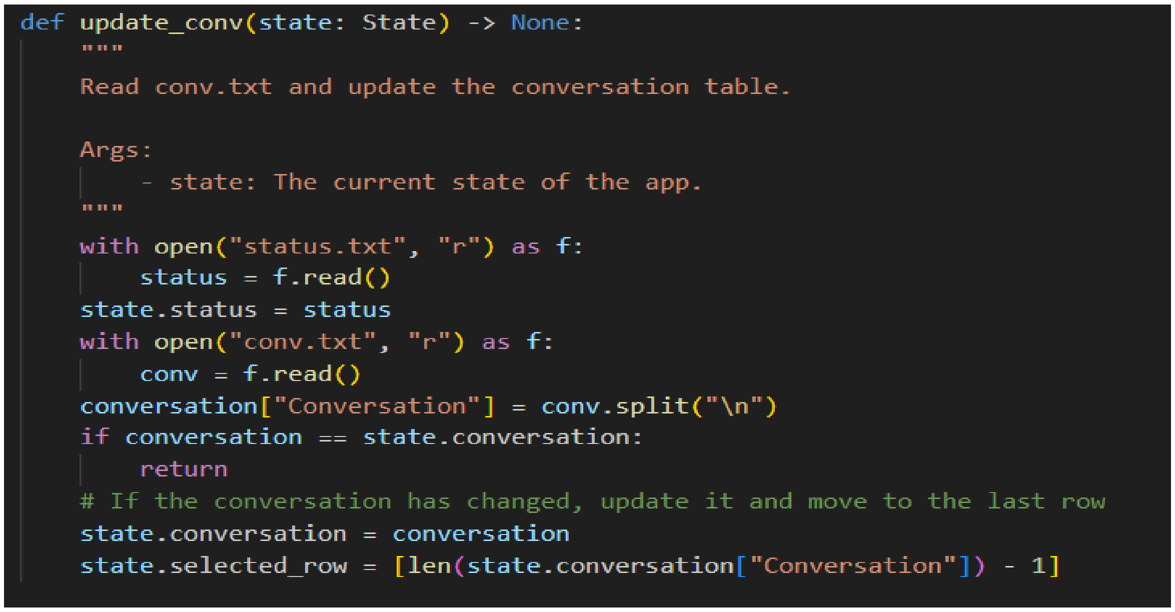


```python
def client_handler(gui: Gui, state_id_list: list) -> None:
    """
    Runs in a separate thread and periodically calls to read conv.txt.

    Args:
        - gui: The GUI object.
        - state_id_list: The list of state IDs.
    """
    while True:
        time.sleep(0.5)
        if len(state_id_list) > 0:
            invoke_callback(gui, state_id_list[0], update_conv, [])
```

This function runs in a separate thread, periodically checking (every 0.5 seconds) to read and update the conv.txt file. If there are state IDs present, it calls the update_conv function to refresh the conversation in the GUI.

4. ***Function update conversation:***

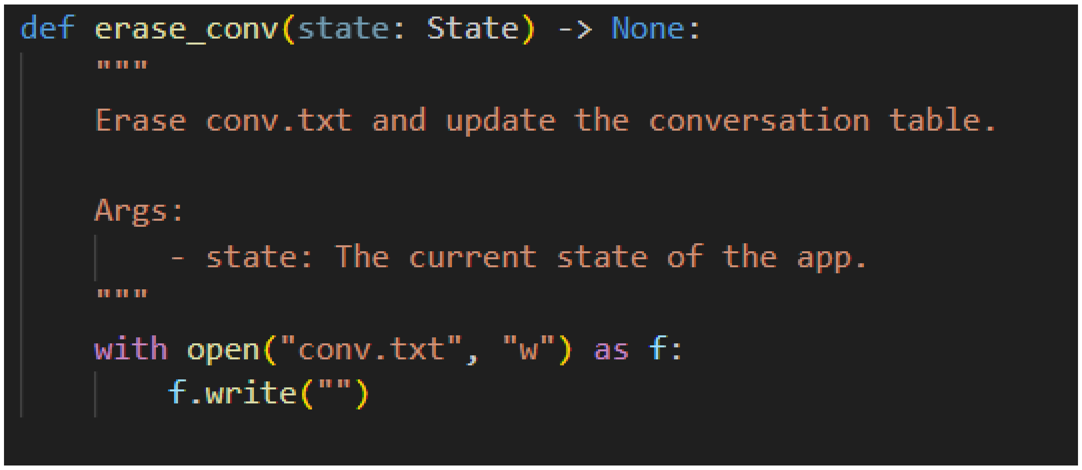


```python
def update_conv(state: State) -> None:
    """
    Read conv.txt and update the conversation table.

    Args:
        - state: The current state of the app.
    """
    with open("status.txt", "r") as f:
        status = f.read()
    state.status = status
    with open("conv.txt", "r") as f:
        conv = f.read()
    conversation["Conversation"] = conv.split("\n")
    if conversation == state.conversation:
        return
    # If the conversation has changed, update it and move to the last row
    state.conversation = conversation
    state.selected_row = [len(state.conversation["Conversation"]) - 1]
```

This function reads the conv.txt file and updates the conversation table in the app. It first reads the status.txt file to get the current status, then reads the conv.txt file to update the conversation. If there is any change in the conversation, it updates the state.conversation and moves the selection to the last row.

5. ***Erase function:***

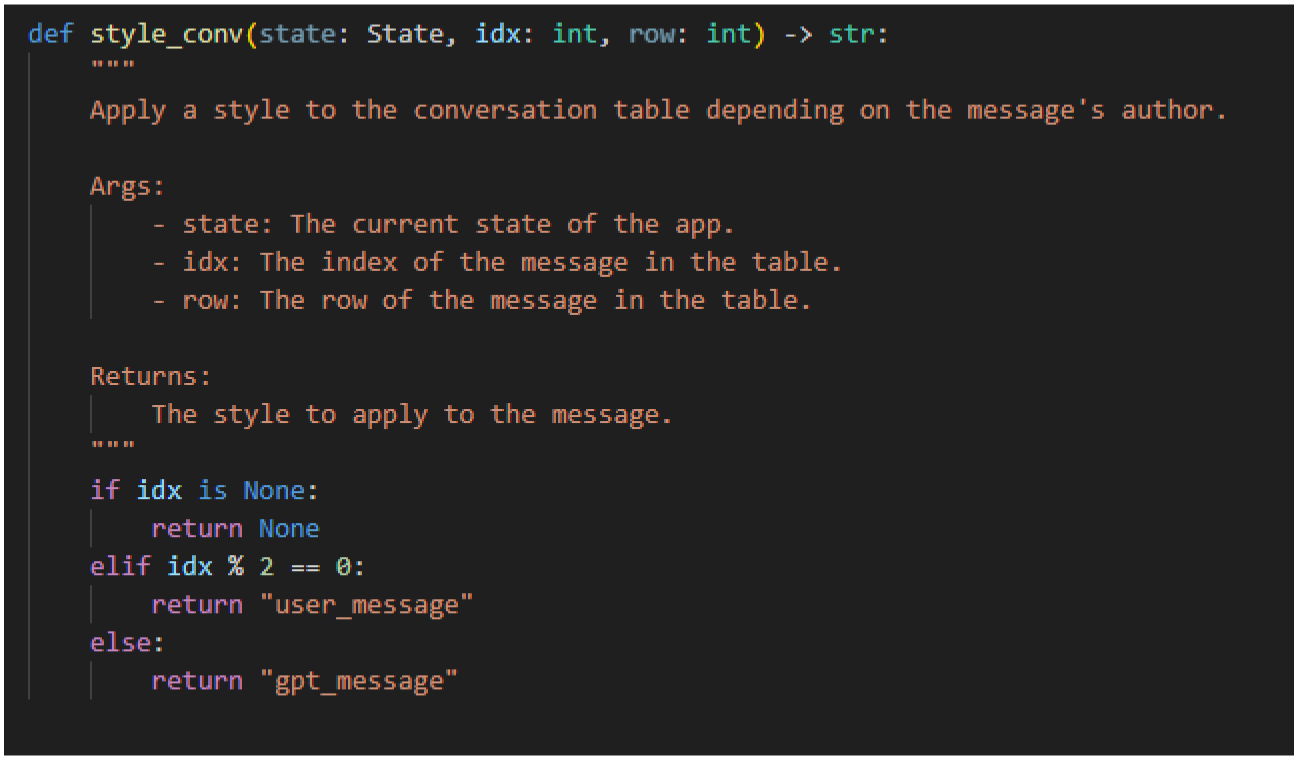


```python
def erase_conv(state: State) -> None:
    """
    Erase conv.txt and update the conversation table.

    Args:
        - state: The current state of the app.
    """
    with open("conv.txt", "w") as f:
        f.write("")
```

This function clears the contents of the conv.txt file. It can be invoked to reset the conversation.

6. ***Function: style_conv:***



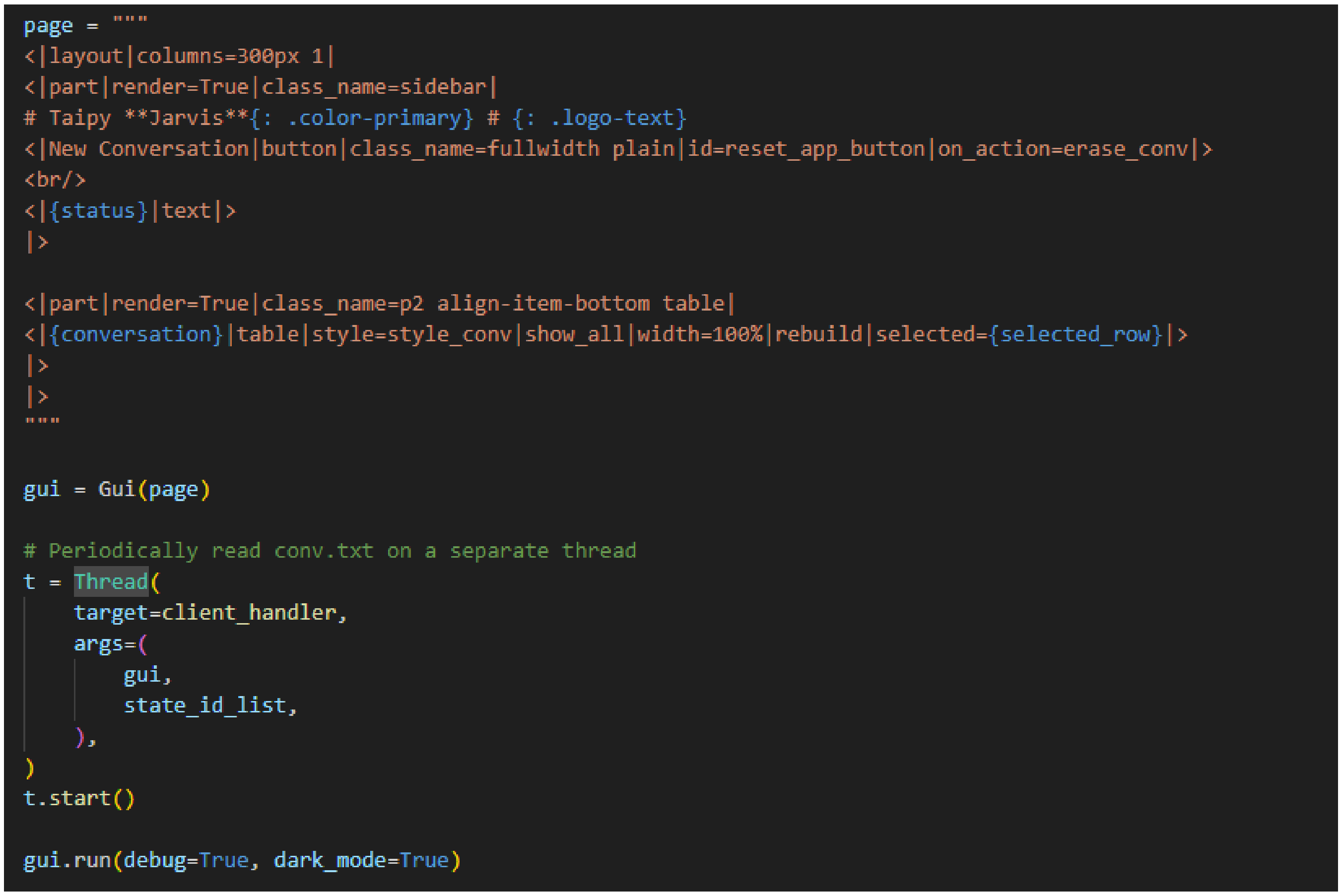
```python
def style_conv(state: State, idx: int, row: int) -> str:
    """
    Apply a style to the conversation table depending on the message's author.

    Args:
        - state: The current state of the app.
        - idx: The index of the message in the table.
        - row: The row of the message in the table.

    Returns:
        The style to apply to the message.
    """
    if idx is None:
        return None
    elif idx % 2 == 0:
        return "user_message"
    else:
        return "gpt_message"
```

This function applies different styles to the conversation table based on the message's author. Messages at even indices are styled as user messages, and messages at odd indices are styled as GPT messages.

7. GUI:



```
page = """
<|layout|columns=300px 1|
<|part|render=True|class_name=sidebar|
# Taipy **Jarvis**{: .color-primary} # {: .logo-text}
<|New Conversation|button|class_name=fullwidth plain|id=reset_app_button|on_action=erase_conv|>
<br/>
<|{status}|text|>
|>

<|part|render=True|class_name=p2 align-item-bottom table|
<|{conversation}|table|style=style_conv|show_all|width=100%|rebuild|selected={selected_row}|>
|>
|>
"""

gui = Gui(page)

# Periodically read conv.txt on a separate thread
t = Thread(
    target=client_handler,
    args=(
        gui,
        state_id_list,
    ),
)
t.start()

gui.run(debug=True, dark_mode=True)
```

❖ GUI Page Definition:

Defines the layout and components of the GUI page using Taipy's templating syntax. It includes:

- A sidebar with a button to start a new conversation and displays the current status.
- The main part shows the conversation in a table format with alternating styles for user and GPT messages.

❖ GUI Initialization:

- The Gui object is initialized with the defined page layout.
- A separate thread is started to periodically read the conv.txt file and update the GUI.
- The gui.run method is called to start the GUI in debug mode with dark mode enabled.

Summary

display.py is responsible for displaying the conversation in a Taipy GUI. It periodically reads the conv.txt file to update the conversation, applies styles to differentiate between user and GPT messages, and provides a button to reset the conversation. The application runs a separate thread to handle the periodic updates without blocking the main thread. The GUI is initialized with a defined layout, and the app runs with continuous updates.

5. ***Application Demonstration:***

A- Overview:

The application demonstrates a voice-based assistant named "Jarvis," designed to assist users through spoken interactions. It leverages various APIs and libraries to provide functionalities such as speech recognition, natural language understanding, and text-to-speech conversion. The assistant can transcribe spoken words, understand and process the user's intent, generate appropriate responses, and speak these responses back to the user.

B- Components and Workflow:

❖ Voice Recording:

- The user initiates interaction by speaking into the microphone.
- The application uses the speech_to_text function to record the audio until silence is detected.
- The recorded audio is saved as a WAV file for further processing.

❖ Transcription:

- The recorded audio is transcribed into text using the Deepgram API.
- This transcription process converts spoken language into written text, making it understandable for further processing by the application.

❖ Natural Language Processing:

- The transcribed text is sent to the OpenAI GPT-3.5 API for generating a response.
- GPT-3.5 processes the input text, understands the context, and generates a relevant response based on its training and the given prompt.

❖ Text-to-Speech Conversion:

- The generated text response from GPT-3.5 is converted into spoken language using the ElevenLabs API.
- This process involves synthesizing the text into an audio file that mimics human speech.

❖ Audio Playback:

- The synthesized speech audio is played back to the user using the pygame mixer.
- The application ensures that the audio response is clear and audible, providing a seamless interaction experience.

❖ Graphical User Interface (GUI):

- The application includes a GUI built with Taipy to display the conversation history.
- The GUI updates in real-time, showing the user's input and the assistant's responses.
- It also provides a button to clear the conversation history, allowing users to start a new interaction session.

C-User Interaction Flow:

❖ Initiation:

- The user starts the application and begins speaking to Jarvis.
- Jarvis listens for the user's speech, recording it until the user stops speaking.

❖ Processing:

- The recorded speech is transcribed into text.
- The transcribed text is sent to GPT-3.5, which generates a response.
- The response text is converted into speech audio.

❖ Response Delivery:

- Jarvis plays the generated speech audio, delivering the response to the user.
- The interaction is logged and displayed in the GUI, providing a visual representation of the conversation.

❖ Continuous Interaction:

- The application loops continuously, allowing the user to interact with Jarvis in real-time.
- The GUI updates dynamically, reflecting the ongoing conversation.

D- Key Features:

- Real-Time Speech Recognition:

Efficiently captures and processes user speech in real-time.

- Accurate Transcription:

Uses advanced AI models to transcribe speech with high accuracy.

- Intelligent Response Generation:

Leverages GPT-3.5 to provide contextually appropriate and informative responses.

- Natural Sounding Text-to-Speech:

Utilizes ElevenLabs technology to generate natural and engaging speech audio.

- Interactive GUI:

Displays conversation history and provides user controls for managing interactions.

E- Practical Applications:

- Personal Assistants:

Enhance user productivity by providing instant answers and performing tasks through voice commands.

- Accessibility:

Assist individuals with disabilities by providing voice-controlled interactions.

- Education:

Aid in learning by answering questions and providing explanations in an interactive manner.

Conclusion

In this project, we successfully developed an AI-driven voice assistant named "Jarvis" that integrates multiple advanced technologies to facilitate seamless and interactive voice-based communication.

The assistant leverages speech recognition for capturing and transcribing user speech, natural language processing via GPT-3.5 for generating contextually relevant responses, and text-to-speech synthesis for delivering responses in a natural and engaging manner. Additionally, a user-friendly GUI provides real-time updates and interaction controls, enhancing the overall user experience.

The project demonstrates the potential of combining various AI services to create a robust and versatile application that can be used in multiple domains such as personal assistance, customer support, accessibility, and education. By effectively utilizing APIs from Deepgram, OpenAI, and ElevenLabs, the assistant provides accurate and efficient responses, showcasing the significant advancements in AI and its practical applications in improving user interactions and accessibility.

This project serves as a testament to the power of modern AI technologies in transforming everyday tasks and interactions into more intuitive and efficient processes.

References

1. **OpenAI GPT-3**:
    - OpenAI. "GPT-3: Language Models are Few-Shot Learners." 2020. Available at: https://arxiv.org/abs/2005.14165
    - OpenAI. "API Documentation." OpenAI, 2023. Available at: https://beta.openai.com/docs/
2. **Deepgram**:
    - Deepgram. "Deepgram Speech Recognition API." Deepgram, 2023. Available at: https://deepgram.com/docs/
    - Deepgram. "Deepgram API Overview." Deepgram, 2023. Available at: https://developers.deepgram.com/
3. **ElevenLabs**:
    - ElevenLabs. "Text-to-Speech API." ElevenLabs, 2023. Available at: https://elevenlabs.io/docs/
    - ElevenLabs. "API Reference." ElevenLabs, 2023. Available at: https://docs.elevenlabs.io/
4. **Pygame**:
    - Pygame Community. "Pygame Documentation." Pygame, 2023. Available at: https://www.pygame.org/docs/
    - Shinners, Pete. "Introduction to Pygame." Pygame, 2023. Available at: https://www.pygame.org/wiki/GettingStarted
5. **pyaudio**:
    - PortAudio. "PyAudio: PortAudio v19 Python Bindings." PyAudio, 2023. Available at: https://people.csail.mit.edu/hubert/pyaudio/
    - PortAudio. "PortAudio API Overview." PortAudio, 2023. Available at: http://portaudio.com/docs/v19-doxydocs/
6. **Rhasspy Silence**:
    - Rhasspy. "Rhasspy Silence Library." Rhasspy, 2023. Available at: https://rhasspy.readthedocs.io/en/latest/
    - Rhasspy. "WebRTC VAD Recorder." Rhasspy, 2023. Available at: https://github.com/rhasspy/rhasspy-silence
7. **Dotenv**:
    - Theskumar. "python-dotenv: Read key-value pairs from a .env file and set them as environment variables." GitHub, 2023. Available at: https://github.com/theskumar/python-dotenv

8. **Pathlib**:

- Python Software Foundation. "pathlib — Object-oriented filesystem paths." Python 3.8.5 documentation, 2023. Available at: https://docs.python.org/3/library/pathlib.html

9. **Taipy**:

- Taipy. "Taipy Documentation." Taipy, 2023. Available at: https://docs.taipy.io/
- Taipy. "Getting Started with Taipy." Taipy, 2023. Available at: https://docs.taipy.io/en/latest/getting_started/

10. **Miscellaneous**:

- Python Software Foundation. "Python 3.8.5 Documentation." Python, 2023. Available at: https://docs.python.org/3/
- Real Python. "Working with Files in Python – Real Python." Real Python, 2023. Available at: https://realpython.com/working-with-files-in-python/

These references provide comprehensive documentation and resources for understanding and utilizing the various technologies and APIs integrated into the project.

Slaymane ABDUL HAMID - 200201942
Hassan ISSA – 20021331