The report of the AI assistant project

Analysis

Scope of the program

This development involves creating a modular program. The deliverables include:

Having a detailed design of the program structure and an evaluation report for the final program

Modular Codebase:

- Well-structured and organized codebase with clearly defined modules.
- Each module should handle specific functionality (e.g., speech recognition, text processing, API interactions).

API Integrations:

- Integration with various APIs (e.g., weather, news, task management).
- Configurable API keys and endpoints.

Error Handling and Logging:

- Robust error handling mechanisms.
- Logging for debugging and monitoring.

Boundaries

- The program will reads the data from the Api request
- Input: the raw input from the microphone and the text input
- Output: A recognized text or an error message that these will be transmitted to speech voice
- Providing highly accurate transcription for noisy or multi-speaker environments.

End-user requirements

- Users should be able to talk to the assistant and receive spoken responses.
- Users should have the option to type questions and commands and receive text responses.
- Users should be able to ask for the current weather in a specific city.
- Users should be able to ask for the latest news headlines.
- Users should be able to say "hello" or "hi" and receive a friendly greeting in return or receive a polite farewell by saying “bye” or “goodbye”.

Functional requirements

Input:

- Commands and Queries
- Text input
- Speech input

Process

Initialization:

- Set up the text-to-speech engine.
- Configure the microphone for speech input.

User Interaction:

- Wait for the user to choose between text and speech interaction.

Receive Input:

- **Text Mode:** The user types a question or command.
- **Speech Mode:** The user speaks a question or command.

Process Input:

- Determine if the user asked for weather, news, or a basic conversation (e.g., greetings).

Fetch Data (if needed):

- **Weather:** Make an API call to get the weather for a specified city.
- **News:** Make an API call to get the latest news headlines.

Generate Response:

- Create a response based on the user's input and any fetched data.

Provide Output:

- **Text Mode:** Display the response.
- **Speech Mode:** Speak the response using the text-to-speech engine.

Exit Condition:

- If the user says "bye" or "goodbye," end the interaction

Output

- If the command is found matching with code, then the data or the response will be displayed to the end-user input.
- If can’t found, a error message will display to the end-user.

Constraints

- Python and other libraries will be used to develop the program
    - speech_recognition
    - pyaudio
    - pyttsx3
    - nltk
    - requests (api)
    - flask
- The working program will run on all the popular operating systems.

Research

Library that has used in the program

- speech_recognition
- pyaudio
- pyttsx3
- nltk
- requests (api)
- flask

speech_recognition

Understanding Speech Recognition

Speech recognition is a subfield of computer science that enables a computer to convert spoken language, typically captured through a microphone, into text. This technology has been in development since before the 1970s.

Often referred to as Automatic Speech Recognition (ASR), Computer Speech Recognition, or Speech-to-Text (STT), we will use "speech recognition" as the general term in this explanation.

How Speech Recognition Works

Most general-purpose speech recognition systems use **Hidden Markov Models (HMMs)**. HMMs are well-suited for speech recognition because a speech signal can be treated as a piecewise stationary signal or a short-time signal. Over short intervals (about 10 milliseconds), speech can be approximated as a stationary process. Since speech can be modeled stochastically, HMMs are a natural fit for recognizing patterns in spoken language.

While there are other methods such as **Dynamic Time Warping (DTW)**, **Neural Networks**, and **End-to-End ASR**, HMM-based systems are still widely used, so we'll focus on this approach here.

Types of Speech Recognition Systems

In addition to HMM-based systems, modern speech recognition can be categorized into three main types:

1. **Dynamic Time Warping (DTW)-based Speech Recognition**
2. **Neural Networks-based Speech Recognition**
3. **End-to-End Automatic Speech Recognition**

However, since HMM is the primary focus of this program, we won't delve into the details of the other approaches.

Key Skills in Speech Recognition

Here are some key components and techniques I learned while working with speech recognition systems:

Recognition Classes

There are several methods for recognizing speech from an audio source, each using a different API:

- recognize_bing(): Microsoft Bing Speech API
- recognize_google(): Google Web Speech API (This is the API used in the program)
- recognize_google_cloud(): Google Cloud Speech (requires the google-cloud-speech package)
- recognize_houndify(): Houndify by SoundHound
- recognize_ibm(): IBM Speech to Text
- recognize_sphinx(): CMU Sphinx (requires installing PocketSphinx)
- recognize_wit(): Wit.ai

Working with Audio Files

The following file formats are supported for speech recognition:

- WAV
- AIFF

- AIFF-C
- FLAC

When importing an audio file into a speech recognition program, you can apply two important parameters to control which portions of the audio are transcribed: **offset** and **duration**.

- **Duration**: Defines how many seconds of audio should be transcribed after starting.
    - Example: duration = 4 will only transcribe the first 4 seconds of the audio.
- **Offset**: Specifies the number of seconds from the start of the audio that should be ignored.
    - Example: offset = 4 will skip the first 4 seconds of the audio before starting the transcription.

Handling Noisy Audio

Background noise can affect the accuracy of speech recognition. The method adjust_for_ambient_noise() helps reduce background noise and improves transcription accuracy in noisy environments.

NLTK

Introduction to Natural Language Processing (NLP) with NLTK

Natural Language Processing (NLP) is a field that enables computers to understand, interpret, and work with human language. In Python, one of the most widely used libraries for NLP is **NLTK** (Natural Language Toolkit). NLTK is essential for analyzing large amounts of unstructured data, particularly text. In this project, I used NLTK to perform text preprocessing, which involves preparing the text for analysis and creating visualizations based on the results.

Key Features of NLTK in the Project

The primary tasks I performed with NLTK include:

1. **Finding text for analysis**
2. **Processing the text**
3. **Analyzing the text**

These tasks are made possible through several powerful features offered by NLTK, including **tokenization** and **stopword filtering**.

Tokenizing Text

Tokenization is one of the essential skills for text preprocessing. It involves splitting the text into smaller units, either words or sentences, allowing the program to work with manageable pieces of text.

Word Tokenization

To tokenize text by word, we use the function word_tokenize() from the NLTK package. This function breaks the text into individual words. For example, the sentence:

- "Apple products are the best."

When tokenized, becomes:

- ["Apple", "products", "are", "the", "best"]

Sentence Tokenization

To tokenize by sentence, the function sent_tokenize() is used. It splits text into sentences. For example, the following passage:

- *"Muad'Dib learned rapidly because his first training was in how to learn. And the first lesson of all was the basic trust that he could learn. It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult."*

Can be tokenized into:

1. *"Muad'Dib learned rapidly because his first training was in how to learn."*
2. *"And the first lesson of all was the basic trust that he could learn."*
3. *"It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult."*

These tokenization methods break down text into smaller, more manageable components, making it easier to analyze.

Filtering Stopwords

Another essential skill in NLP is **stopword filtering**. Stopwords are common words (such as "the", "is", "in") that carry little meaning in the context of text analysis. Filtering them out enhances the accuracy and effectiveness of the analysis.

To filter stopwords, NLTK provides a built-in **stopwords library**. The code snippet below demonstrates how to remove stopwords from a list of words:

Alternatively, you can create a filtered list in a more concise way using list comprehension:

filtered_list = [word for word in words_in_quote if word.casefold() not in stop_words]

This step is crucial for focusing the analysis on the meaningful words in the text.

Conclusion

The two key skills of **tokenization** and **stopword filtering** form the foundation of the text preprocessing workflow in this project. By splitting text into smaller units and filtering out irrelevant words, these techniques help to improve the analysis of the text or speech entered by the user, making NLTK an invaluable tool for NLP tasks.

Requests

The requests module is a popular Python library used for sending HTTP requests. It's often described as one of the most user-friendly ways to interact with web servers to send or receive data. Whether you need to fetch information from a website, interact with an API, or submit data to a web service, the requests library provides a straightforward and efficient way to do it.

In my project, I used the requests module primarily to fetch data from web-based APIs. This included getting real-time weather updates and fetching the latest news headlines. The beauty of using requests is in its simplicity and effectiveness, making it a preferred choice for tasks involving network communication.

Making HTTP Requests

One of the main tasks you often perform when using the requests library is sending HTTP requests. HTTP (Hypertext Transfer Protocol) is the foundation of any data exchange on the Web, and requests makes it easy to perform these operations. In my project, I used the GET method to retrieve information from external APIs.

Here's an example of how a basic GET request works with requests:

```
import requests url = "http://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY&units=metric" response = requests.get(url)
```

In this code snippet, a GET request is sent to the OpenWeatherMap API to get the current weather for London. The URL includes the city name, an API key for authentication, and a parameter to specify that the temperature should be returned in metric units.

Handling Responses

Once the request is made, the server sends back a response. The requests library makes it easy to handle these responses. You can check if the request was

successful, access the data returned by the server, and handle any potential errors.

- **Accessing Data**: You can get the content of the response as raw text using response.text, or as a JSON object if the server returns JSON data, using response.json(). This is particularly useful when working with APIs, as most of them return data in JSON format.

```
data = response.json()
```

- **Checking Status Codes**: Every response comes with a status code. A status code of 200 means the request was successful, while codes like 404 (not found) or 500 (server error) indicate problems. Using these codes, you can build logic to handle different scenarios.

```
if response.status_code == 200:
    print("Request was successful!")
else: print(f"Request failed with status code:
    {response.status_code}")
```

- **Error Handling**: The requests library also includes error handling capabilities. For instance, you can handle scenarios where the server is unreachable, the request times out, or invalid responses are returned. This is crucial for building robust applications that can handle network issues without crashing.

```
try:
    response = requests.get(url)
    response.raise_for_status()  # This will raise an error for bad responses
(like 4XX or 5XX)
    data = response.json()
except requests.exceptions.HTTPError as http_err:
    print(f"HTTP error occurred: {http_err}")
except requests.exceptions.ConnectionError:
```

```
        print("Failed to connect to the server.")
    except requests.exceptions.Timeout:
        print("The request timed out.")
    except requests.exceptions.RequestException as req_err:
        print(f"An error occurred: {req_err}")
```

How requests is Used in This Project

In this project, requests is a key component for accessing real-time information. For instance, when a user wants to check the weather, the get_weather() function uses requests.get() to connect to a weather API. It then processes the JSON response to pull out details like temperature, humidity, and a short description of the current weather.

Similarly, for fetching news, the get_news() function uses a GET request to connect to a news API and retrieve the top headlines. By looping through the first few articles, it gathers titles and descriptions to present to the user.

The use of requests in this project shows how important it is for interacting with web services. Its simplicity and reliability make it an ideal choice for fetching data over the internet and integrating that data into applications.

Design

At the design stage of the project I have used pseudocode to design the framework of my code will be look like and mostly how it will be works

1. Initialize the system
    - Set up speech recognition engine
    - Set up text-to-speech engine
    - Load default settings for voice, rate, volume, and language

2. Main Function

- Loop until user exits
    - Ask user if they want to interact using text or speech
    - If choice is 'text':
        - Call text_interaction()
    - If choice is 'speech':
        - Call speech_interaction()
    - If choice is 'exit':
        - End the loop

3. Function: text_interaction()

- Loop until user types 'back'
    - Get user input text
    - If input is 'back':
        - Exit the loop
    - Call process_user_input(input)
    - Call respond_to_user(response)

4. Function: speech_interaction()

- Loop until user says 'back'
    - Prompt user to say something
    - Capture user speech input
    - If input is 'back':

        - Exit the loop
    - Call process_user_input(input)
    - Call respond_to_user(response)
5. Function: process_user_input(input)
    - Convert input to lowercase
    - Determine the user's intent:
        - If input contains greeting keywords (e.g., 'hello', 'hi'):
            - Set response to greeting message
        - If input contains 'news':
            - Fetch top 5 news headlines using get_news()
            - Set response to news headlines
        - If input contains 'weather':
            - Ask for city name
            - Fetch weather information using get_weather(city)
            - Set response to weather information
        - If input contains 'time' or 'date':
            - Get current time or date
            - Set response to time/date
        - If input contains 'joke' or 'quote':
            - Set response to a predefined joke or motivational quote
        - If input contains 'bye' or 'goodbye':
            - Set response to farewell message
        - For other inputs, set response to default message

6. Function: respond_to_user(response)
    - Display the response
    - Use text-to-speech to say the response

7. Function: get_weather(city)

- Call weather API with the city name
- Parse and return the weather information

8. Function: get_news()

- Call news API
- Parse and return the top 5 news headlines

9. Function: update_settings(setting_type, value)

- Update the relevant setting (voice, speed, volume, language)
- Apply changes to text-to-speech engine

Implementation

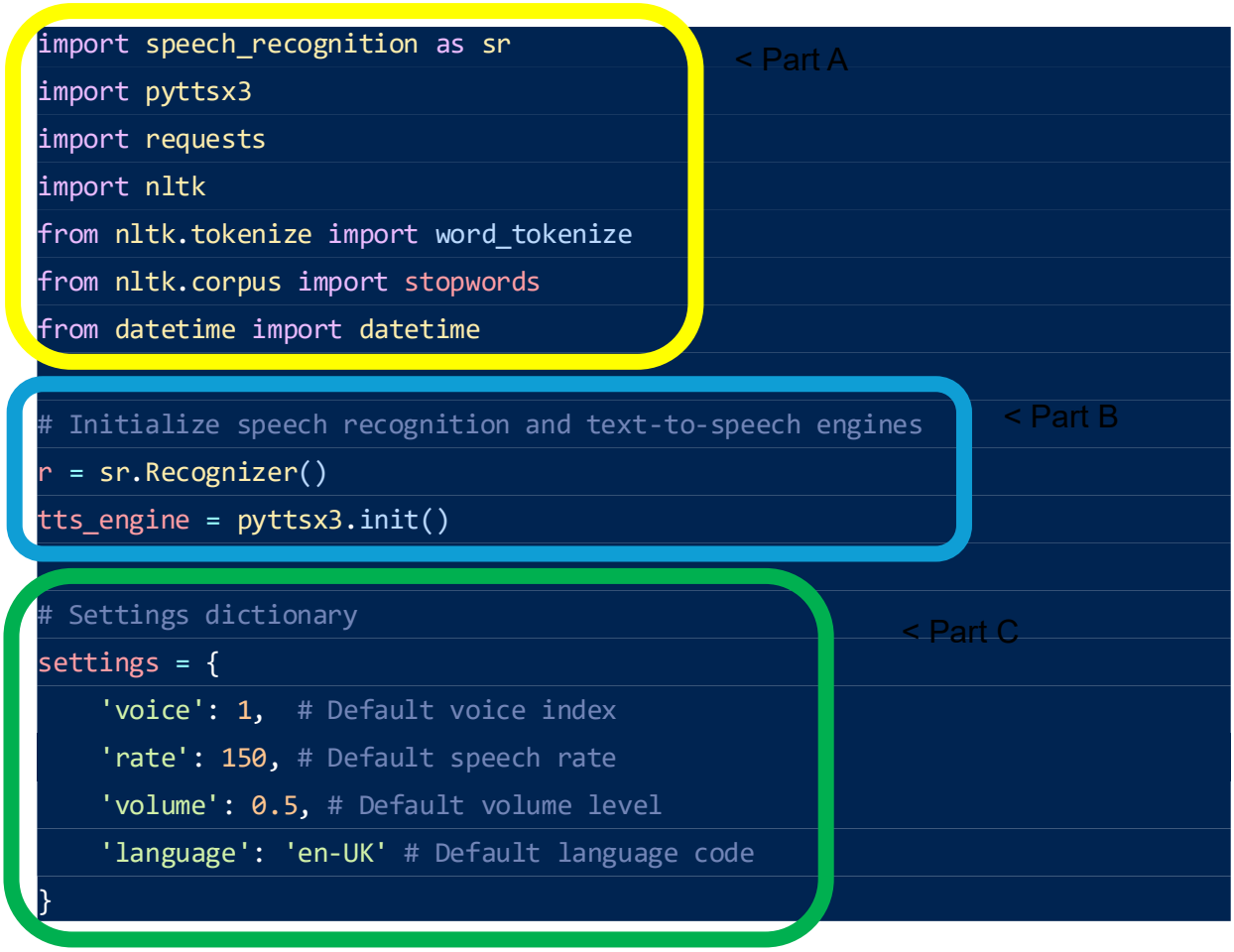
Main program part

In the main program can be separated in to 7 different parts based on their functionalities.

- Imports and Initialization
- Utility Functions for TTS and Speech Recognition
- Weather and News API Interaction
- Text Processing and Reactions
- Settings Modification Functions
- Interaction Modes (Text and Speech)
- Main Function

I will explain the purpose and how I implemented that part of code in the program, also the error that I encountered while I am coding it.

Imports and Initialization



Purpose: This section includes all necessary imports and initializations for external libraries and settings.

For Part A is implemented for importing all the python libraries that is needed in the program, for example pyttsx3, nltk, requests etc. libraries.

For Part B is initializing the speech recognition and text-to-speech engines into a variable, “r” for the speech recognition and “tts_engine” is for the text-to-speech engines.

For Part C is setting up a dictionary for the setting of the text-to-speech engines, voice, the speech rate, volume of the speech and the language that it speak.

Utility Functions for TTS and Speech Recognition

Inside of this part there is 3 section of code included, “update_tts_engine”, “text_to_speech”, “receive_speech”.

```
def update_tts_engine():
    """Update the TTS engine settings based on user preferences."""
    voices = tts_engine.getProperty('voices')
    tts_engine.setProperty('voice', voices[settings['voice']].id)
    tts_engine.setProperty('rate', settings['rate'])
    tts_engine.setProperty('volume', settings['volume'])
```

“update_tts_engine”

Purpose: The purpose of this section is to update the tts_engine and changes the way the virtual assistant speaks based on user preferences. It adjusts the Voice, rate, volume of the assistant speak.

Implementation: at the 3 lines of setProperty code, normally will use numbers to set the property but I have change the value to variable that is used at the “Settings Modification Functions” at that part can change the value of the variable that will leads to the change in the property of the tts_engine.

```
def text_to_speech(text):
    """Convert the given text to speech."""
    try:
        update_tts_engine()  # Apply the current settings
        tts_engine.say(text)
        tts_engine.runAndWait()
    except Exception as e:
        print(f"An error occurred during text-to-speech: {e}")
```

“text_to_speech”

Purpose: This function is used to convert a given text to spoken words using a TTS engine. The function provides a simple way to convert and play text as speech while handling potential errors that might occur during the process.

Implementation: I have used “try ” in this part of the code, “try” is used to define a block of code in which Python will attempt to execute the operations specified (in this case, converting text to speech and playing it). This is good for handling errors the program won’t crash because of a error. And for the last two lines in this part is used to notices the user about the error during the TTS.

```
def receive_speech():
    """Listen to the user's speech and return the recognized text."""
    print("Listening...")
    try:
        with sr.Microphone() as source:
            r.adjust_for_ambient_noise(source)
            audio = r.listen(source)
            try:
                text = r.recognize_google(audio,
language=settings['language'])
                print(f"Recognized text: {text}")
                return text
            except sr.UnknownValueError:
                print("Sorry, I didn't catch that. Could you repeat?")
            except sr.RequestError as e:
                print(f"Could not request results from Google Speech
Recognition service; {e}")
    except Exception as e:
        print(f"An unexpected error occurred while listening: {e}")
    return None
```

“receive_speech”

Purpose : This function is designed to interact with users by capturing their speech and converting it into text, which could then be used for further processing or commands within an application. It is useful in applications that need voice input, like voice assistants or speech-to-text transcription tools.

Implementation: I have set up the Microphone that is used at first, then store in the audio that is received from the Microphone, after that I use the speech recognition to translate the audio into text. After these preparations, the program will return the

reply, but if the program can catch what the user is saying then it will reply a sentence to let the user knows and request the user to repeat what they said, also if the google recognition can't have result there is also a reply for it.

Weather and News API Interaction

```
def get_weather(api_key, city_name):
    """Fetch and return weather information for a given city."""
    url =
f"http://api.openweathermap.org/data/2.5/weather?q={city_name}&appid={api_key}
&units=metric"
    try:
        response = requests.get(url)
        response.raise_for_status()
        data = response.json()

        if "main" in data:
            main = data['main']
            wind = data['wind']
            weather_description = data['weather'][0]['description']
            weather_info = (f"Temperature: {main['temp']}°C\n"
                            f"Humidity: {main['humidity']}%\n"
                            f"Pressure: {main['pressure']} hPa\n"
                            f"Wind Speed: {wind['speed']} m/s\n"
                            f"Weather Description: {weather_description}")
            return weather_info
```

```
def get_news(api_key):
    """Fetch and return the top 5 news headlines."""
    url = f"https://newsapi.org/v2/top-headlines?country=us&apiKey={api_key}"
    try:
        response = requests.get(url)
        response.raise_for_status()
        data = response.json()

        if "articles" in data:
            articles = data['articles']
            news = ""
            for article in articles[:5]:
                news += f"Title: {article['title']}\nDescription:
{article['description']}\n\n"
            return news
```

Purpose : The first half part of weather is used to provide a user-friendly summary of the current weather conditions for a specified city. It interacts with the OpenWeatherMap API to fetch live weather data, handles possible errors, extracts relevant weather details, and returns them in a structured and readable format. And it's the same for the news part, but just not that complex that won't needed to enter in what city's news. In summary, it's uses the same method but just fetching different things

Implement : I have set up both the url for fetching the weather and news data at the start of the function. And store the respond that is get from the url, also convert the responds data in to json format and store into the "data" variable. For the next half part of the code that is show is to print out the list of data by getting the data in the "data" and list out oraganised.

```
else:
            return "Unexpected response format. Please try again later."
    except requests.exceptions.HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
        return "Unable to fetch news or invalid API key."
    except requests.exceptions.ConnectionError:
        print("Failed to connect to the news service. Please check your
internet connection.")
        return "Unable to connect to the news service."
    except requests.exceptions.Timeout:
        print("The request to the news service timed out.")
        return "The news service is taking too long to respond. Please try
again later."
    except requests.exceptions.RequestException as req_err:
        print(f"An error occurred while requesting news data: {req_err}")
        return "An error occurred while fetching the news."
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return "An unexpected error occurred. Please try again later."
```

Purpose : The purpose of this code is to provide robust error handling for the these two function. By catching different types of exceptions, it ensures that the function can handle various issues, such as connectivity problems, invalid API responses, and unexpected runtime errors. This results in a more resilient function that can provide helpful feedback to the user in case of failure, rather than crashing or providing misleading information.

Text Processing and Reactions

```
def process_text(text):
    """Process the input text by tokenizing and removing stop words."""
    text = text.lower()
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word not in stop_words]
    return filtered_tokens
```

Purpose : The “process_text” function helps in preparing the text for further analysis by:

- Converting it to lowercase (to ignore case differences),
- Breaking it down into individual words (tokenizing),
- Removing common, unimportant words (stop words).

This preprocessing makes it easier to analyze the text for keywords or phrases that have significant meaning.

Implementation : First I have Convert Text to Lowercase by doing it by this line. This line converts all characters in the text to lowercase. It helps standardize the text, ensuring that comparisons are not case-sensitive. The next step is to tokenize the text. This line breaks the text into individual words, known as tokens. For example, the sentence "Hello, how are you?" would be split into ['hello', ',', 'how', 'are', 'you', '?']. After tokenize, the program will create a set of stop words, This line creates a set of common English stop words. The stopwords.words('english') function from the nltk library provides a list of common words that are typically not useful for analysis. At last is to filter out stop words, this line creates a new list, filtered_tokens, which includes only the words that are not in the set of stop words. It uses a list comprehension to check each word. For finishing this part by returning the filtered tokens.

```
def reaction(text):
    """Generate a response based on the input text."""
    news_api_key = "78f541f419d84eb889a5b7489a1c0047"

    text = text.lower()

    if 'hello' in text or 'hi' in text:
        return "Hello! How can I assist you today?"
    elif 'assistant' in text:
        return "What can I help you with?"
    elif 'your name' in text:
        return "My name is Assistant."
    elif 'bye' in text or 'goodbye' in text:
        return "Goodbye! Have a great day!"
    elif 'news' in text:
        return get_news(news_api_key)
    elif 'joke' in text:
        return "What do you call a Star Wars droid that takes the long way
around? R2 detour."
```

Purpose : The main purpose of this part is to respond to most of the possible input that the user will input.

Implementation : The mainly way of implementing this part is by if ”the text” they enter by what they enter then return which respond.

```
def weather_reaction(api_key, text, city_name):
    """Respond with weather information if the input text asks for it."""
    if 'weather' in text:
        return get_weather(api_key, city_name)
```

Purpose :

The weather_reaction() function

- Looks for the word "weather" in the user's input text.
- If found, it fetches and returns the weather information for the specified city.

This function helps the virtual assistant respond appropriately when the user asks about the weather.

Implementation : I set up the input parameters at first, then it will check for weather in the text, then will call the get_weather function.

```
def change_voice():
    """Change the voice setting."""
    global settings
    voices = tts_engine.getProperty('voices')
    print("Available voices:")
    for i, voice in enumerate(voices):
        print(f"{i}: {voice.name}")
    choice = input("Enter the number of the voice you want to use: ")
    try:
        settings['voice'] = int(choice)
        update_tts_engine()
        return f"Voice changed to {voices[settings['voice']].name}."
    except (ValueError, IndexError):
        return "Invalid choice. Please select a valid voice number."
```

Purpose :

- **Display Available Voices**: It shows the user a list of all available voices that can be used by the TTS engine.
- **Allow User to Choose a Voice**: It prompts the user to select one of these voices by entering a corresponding number.
- **Update the Voice Setting**: It updates the voice setting in the settings dictionary and applies the change to the TTS engine.

Implementation : First **Access the Global settings Dictionary**, this allows the function to modify the global ‘setting’ dictionary. Then get list of available voices, after this display the available voices, and this will goes through each voice in the list and

prints out the index and name of the voice. Then update the voice setting and return the confirmation message. Last handle the invalid input.

For the other 3 changing setting function is mostly similar to this changing voice setting function.

```
def text():
    """Handle text-based interaction with the assistant."""
    api_key = "83c9ab29ae1b3e085a11b443028b6c7d"
    while True:
        user_input = input("Please enter your text (or type 'back' to return
to the main menu): ").lower()
        if user_input == 'back':
            break
        if user_input:
            tokens = process_text(user_input)
            if 'weather' in tokens:
                city_name = input("What city's weather do you want to check?
")
                response = weather_reaction(api_key, user_input, city_name)
            else:
                response = reaction(user_input)
            print(f"Assistant: {response}")
            text_to_speech(response)
            if 'bye' in user_input or 'goodbye' in user_input:
                break
```

Purpose :

The text() function:

1. **Engages in a Text-Based Conversation**: It provides a way for users to interact with the virtual assistant using text.
2. **Processes Commands**: It handles various commands, such as checking the

weather, greeting, telling jokes, or getting the current time.

3. **Provides Responses**: It prints responses and uses text-to-speech to read them aloud.
4. **Exits on Command**: It can exit and return to the main menu when the user types "back" or says "bye".

Implementation :

- **Start a Conversation**: The function starts an ongoing conversation where it keeps asking the user to type something.
- **Get User Input**: It asks the user to type a command or a question. For example, the user might type "What's the weather?" or "Tell me a joke."
- **Check for Exit**: If the user types "back," the function stops and goes back to the main menu. If the user types "bye" or "goodbye," the conversation ends.
- **Process the Input**:
  - If the user asks about the weather, the function will ask for the city name and then check the weather for that city.
  - If the input is about something else (like a greeting, joke, or time), it will respond accordingly.
  - **Respond**: The assistant displays the response on the screen and also speaks the response aloud using text-to-speech.
  - **Repeat**: The function continues to ask for user input until the user types "back" or says "bye" to end the interaction.

And for the speech side is mostly also same will the text side, but use change the input method form text to voice input.

```
def main():
    """Main function to start the virtual assistant."""
    while True:
        print("Virtual Assistant is running...")
        choice = input("Hello, would you rather use text or speech to interact
with me? ").lower()
        if choice == "text":
            text()
        elif choice == "speech":
            speech()
        elif choice == 'exit':
            print("Goodbye!")
            break
        else:
            print("Invalid choice. Please type 'text', 'speech', or 'exit'.")

if __name__ == "__main__":
    main()
```

Purpose :

- **Control Center**: The main() function acts as a control center for the virtual assistant, determining the interaction method.
- **User Choice Management**: It manages how users choose to interact (text, speech, or exit) and directs them to the appropriate function.
- **Program Flow**: It provides a structured way to start, interact, and exit the assistant program.

Testing

Basic Functional Testing

Test Case	Description	Expected Output	Result
TC1: Text-based Interaction	User inputs text: "hello"	Assistant replies "Hello! How can I assist you today?"	Pass/Fail
TC2: Speech-based Interaction	User says "hello"	Assistant replies "Hello! How can I assist you today?"	Pass/Fail
TC3: Weather Request (Text)	User inputs text: "What's the weather like?"	Assistant asks for the city name and returns weather info	Pass/Fail
TC4: Weather Request (Speech)	User says "What's the weather like?"	Assistant asks for the city name and returns weather info	Pass/Fail
TC5: News Request	User inputs text or says "What's the news?"	Assistant returns the top 5 news headlines	Pass/Fail
TC6: Time Request	User inputs text or says "What's the time?"	Assistant returns the current time	Pass/Fail
TC7: Date Request	User inputs text or says "What's the date?"	Assistant returns the current date	Pass/Fail
TC8: Joke Request	User inputs text or says "Tell me a joke"	Assistant responds with a joke	Pass/Fail
TC9: Change Voice	User inputs text or says "Change voice"	Assistant prompts to change the voice and applies the change	Pass/Fail

Edge Case Testing

Test Case	Description	Expected Output	Result
TC10: Empty Input (Text)	User inputs an empty string	Assistant responds with "Sorry, I didn't understand that. Can you please rephrase?"	Pass/Fail
TC11: Empty Input (Speech)	User remains silent after prompt	Assistant responds with "Sorry, I didn't catch that. Could you repeat?"	Pass/Fail
TC12: Invalid Command (Text)	User inputs text: "xyz123"	Assistant responds with "Sorry, I didn't understand that."	Pass/Fail
TC13: Invalid Command (Speech)	User says an unknown phrase	Assistant responds with "Sorry, I didn't understand that."	Pass/Fail
TC14: Invalid Weather City	User inputs an invalid city name	Assistant responds with "City not found or invalid API key."	Pass/Fail
TC15: Invalid Voice Selection	User inputs an out-of-range value when changing voice	Assistant responds with "Invalid choice. Please select a valid voice number."	Pass/Fail
TC16: API Timeout (Weather)	Simulate a timeout from the weather API	Assistant responds with "The weather service is taking too long to respond. Please try again later."	Pass/Fail
TC17: API Timeout (News)	Simulate a timeout from the news API	Assistant responds with "The news service is taking too long to respond. Please try again later."	Pass/Fail

Settings Adjustment Testing

Test Case	Description	Expected Output	Result
TC18: Change Speech Rate	User inputs text or says "Change speed" and selects a valid rate	Assistant adjusts the speech rate to the selected value	Pass/Fail
TC19: Change Volume	User inputs text or says "Change volume" and selects a valid volume level	Assistant adjusts the volume to the selected level	Pass/Fail
TC20: Change Language	User inputs text or says "Change language" and selects a valid language code	Assistant adjusts the language to the selected code	Pass/Fail
TC21: Invalid Speech Rate	User inputs an invalid value for speech rate	Assistant responds with "Invalid rate. Please enter a numeric value."	Pass/Fail
TC22: Invalid Volume Level	User inputs an invalid value for volume	Assistant responds with "Volume must be between 0.0 and 1.0."	Pass/Fail
TC23: Invalid Language Code	User inputs an invalid language code	Assistant responds with "Invalid language code."	Pass/Fail

Error Handling

Test Case	Description	Expected Output	Result
TC24: Google Speech Recognition	Simulate a failure in Google's speech recognition service	Assistant responds with "Could not request results from Google Speech Recognition service"	Pass/Fail
TC25: Internet Connection Down	Simulate loss of internet connection	Assistant responds with "Unable to connect to the weather/news service"	Pass/Fail
TC26: Invalid API Key (Weather)	Use an invalid API key for weather	Assistant responds with "City not found or invalid API key."	Pass/Fail
TC27: Invalid API Key (News)	Use an invalid API key for news	Assistant responds with "Unable to fetch news or invalid API key."	Pass/Fail

Performance Testing

Test Case	Description	Expected Output	Result
TC28: Speed of Response (Text)	Measure the response time for text-based interactions	Response time should be less than 1 second	Pass/Fail
TC29: Speed of Response (Speech)	Measure the response time for speech-based interactions	Response time should be less than 2 seconds	Pass/Fail
TC30: Multiple Commands (Text)	User enters multiple commands in quick succession	Assistant should respond appropriately to each command	Pass/Fail

Evaluation

1. Functionality

Core Features

- **Speech Recognition**: Evaluate how well the speech recognition (via Google's API) performs. Does it accurately transcribe spoken commands? Are the common commands like “hello,” “news,” or “weather” recognized properly?
- **Text-to-Speech**: Assess how well the text-to-speech (TTS) engine works. Is the voice clear and understandable? Do voice settings (speed, volume, language) change as expected?
- **Command Handling**: Does the assistant respond appropriately to the range of commands it supports (e.g., weather, time, date, jokes, quotes)?
- **API Integration**:
    - **Weather**: Check if the assistant can successfully retrieve weather information using the OpenWeather API.
    - **News**: Evaluate the performance of the news API integration. Does it fetch and present news headlines correctly?

2. Usability

User Experience

- **Text Interaction**: Is the text-based interaction smooth? Are the prompts clear, and do the responses make sense?
- **Speech Interaction**:
    - How well does the program handle ambient noise?
    - Is it easy for the user to interact with the program using speech?

    - Does the assistant understand a variety of accents and speech speeds?

- **User Feedback**: The program provides feedback for unrecognized inputs or errors. Does the feedback help the user understand what went wrong and how to correct it?

Settings Customization

- **Voice Changes**: Is the process of changing the voice intuitive and functional? Can the user switch between available voices easily?
- **Speech Rate & Volume**: Are the speech rate and volume settings easy to adjust, and does the change reflect immediately?
- **Language Settings**: Is changing the language straightforward, and does the program actually start recognizing and responding in the selected language?

3. Performance

Speed

- **Response Time**: Measure how quickly the assistant responds to text or speech commands. Ideally, text responses should take less than 1 second, and speech responses should take less than 2 seconds.
- **API Call Time**: Evaluate how long it takes to fetch weather or news information. Ensure API calls are optimized and don't take too long (within 2-3 seconds).

Stability

- **Handling Multiple Requests**: Can the program handle multiple consecutive requests without crashing? For example, what happens when the user quickly asks for the weather, then a joke, then the time?
- **Error Handling**: Does the program handle errors (e.g., API timeouts, speech recognition failures) gracefully without crashing?

4. Code Quality

Code Structure

- **Modularity**: Are the program's functions modular and well-separated? For example, functions like get_weather(), reaction(), and text_to_speech() should handle only their respective tasks.
- **Code Readability**: Is the code easy to read and maintain? This includes following proper naming conventions, avoiding hardcoding values where possible, and adding comments where necessary.
- **Error Handling**: Does the program handle exceptions correctly? The try-except blocks should catch potential errors in API requests, speech recognition, and TTS operations without halting the program.

Optimization

- **API Calls**: Are unnecessary API calls avoided? For example, if a request fails, is there an appropriate retry mechanism or error message rather than continuously calling the API?
- **Resource Management**: Does the program manage resources efficiently? For instance, is the microphone correctly released after speech input, and is the TTS engine handled properly without memory leaks?

5. Scalability

Future Features

- **Expandable Design**: Is the code structure flexible enough to allow for future feature additions, such as adding more commands (e.g., weather forecast, reminders, or timers)? For instance, the reaction() function could be extended without cluttering the code.

7. Final Evaluation Summary

Based on the above criteria, I summarize the evaluation in a report format:

- **Functionality**: Does the assistant correctly handle basic and complex commands? (Rating: Poor, Fair, Good, Excellent)
- **Usability**: How user-friendly is the interface, and does the assistant give helpful responses? (Rating: Poor, Fair, Good, Excellent)

- **Performance**: Is the response time fast, and can it handle multiple tasks smoothly? (Rating: Poor, Fair, Good, Excellent)
- **Code Quality**: Is the code well-structured, modular, and easy to extend or debug? (Rating: Poor, Fair, Good, Excellent)
- **Areas for Improvement**: What areas need more work (e.g., handling of unrecognized speech, optimization of API calls)?