ASSISTIVE VOICE ASSISTANT

Abishek Ajai Satnur
Charles Bruner

Client: Levern Currie (Drivingo)
Instructor: Dr. Edward A. Fox

VIRGINIA TECH, Blacksburg, VA 24061
May 9, 2024
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>5</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>6</td>
</tr>
<tr>
<td>REQUIREMENTS</td>
<td>7</td>
</tr>
<tr>
<td>Assumptions</td>
<td>9</td>
</tr>
<tr>
<td>DESIGN AND IMPLEMENTATION</td>
<td>10</td>
</tr>
<tr>
<td>TESTING</td>
<td>19</td>
</tr>
<tr>
<td>TESTING TIMELINE</td>
<td>25</td>
</tr>
<tr>
<td>USER MANUAL</td>
<td>26</td>
</tr>
<tr>
<td>DEVELOPER MANUAL</td>
<td>31</td>
</tr>
<tr>
<td>LESSONS LEARNED</td>
<td>36</td>
</tr>
<tr>
<td>Timeline</td>
<td>37</td>
</tr>
<tr>
<td>Problems and Solutions</td>
<td>41</td>
</tr>
<tr>
<td>FUTURE WORK</td>
<td>43</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>45</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>46</td>
</tr>
</tbody>
</table>
**TABLE OF FIGURES**

FIGURE 1 - ARCHITECTURE DIAGRAM .................................................................10

FIGURE 2 - CONVERSATION IN GERMAN ............................................................19

FIGURE 3 - CONTINUED CONVERSATION IN GERMAN ......................................20

FIGURE 4 - TESTING Backend ACCURATELY STORES ORDER ...........................22

FIGURE 5 - USER INTERFACE OF THE APPLICATION SHOWING ORDER ..........23

FIGURE 6 - ORDERING 6 MEDIUM BLUEBERRY MUFFINS ................................23

FIGURE 7 - TESTING RANDOM QUESTIONS WITH ASSISTANT ..........................24

FIGURE 8 - USER INTERFACE EXAMPLE FOR GESTURE ORDERING ................26

FIGURE 9 - USER INTERFACE AFTER SPEAKING YOUR ORDER ........................27

FIGURE 10 - DEFAULT ORDER SCREEN .............................................................28

FIGURE 11 - DEFAULT ORDER SCREEN IN SPANISH .......................................29

FIGURE 12 - SUBMISSION SCREEN ONCE USER PLACES THEIR ORDER ............30

FIGURE 13 - FLASK ENDPOINT THAT HANDLES AUDIO ORDERING ................32

FIGURE 14 - CREATES/UPDATES THREAD RESPONSIBLE FOR PROCESSING ORDER .........................................................................................................................33
FIGURE 15 - RUNNING THREAD WHICH GENERATES AI RESPONSE ..................34

FIGURE 16 - HOW MULTIPLE ITEM ORDERING IS DEALT WITH ......................35
TABLE OF TABLES

TABLE 1 - PYTHON FILE PURPOSES .................................................................12

TABLE 2 - GENERAL TIMELINE .................................................................37
ABSTRACT

The traditional drive-through ordering process can often fall short in accessibility, leading to challenging experiences for customers facing language barriers or those with hearing and speech impairments. To address this issue, Drivingo has developed an innovative all-in-one ordering kiosk that enhances accessibility through touch, hand gestures, and audio-based interfaces. This kiosk caters to a diverse customer base, improving service inclusivity.

In collaboration with Drivingo, our project focused on enhancing the audio-based ordering component by integrating a large language model API. This API employs conversational AI and natural language processing to accurately understand user intent, allowing the system to interact with customers as would a human drive-through attendant.

During the development of this software, we encountered the challenge of creating a model that was sufficiently general to operate across various fast-food chains and menus, yet detailed enough to capture the nuances of customer orders. This required extensive research and fine-tuning of the AI to work seamlessly with Drivingo’s existing systems.

We believe that our approach not only broadens access to drive-through services but also introduces a scalable model adaptable to various fast-food chains. Looking forward, we envision enhancements such as automatic menu adaptation, which will reduce the need for manual updates, and simplify integration for new establishments.

This project has demonstrated how AI can be leveraged to create more inclusive and efficient customer service solutions in the drive-through industry, pointing the way toward a future where technology continually enhances accessibility in everyday experiences.
INTRODUCTION

Problem:
Traditional drive-through ordering systems often lack accessibility, creating challenges for customers with language barriers or those with hearing or speech impairments [1]. This leads to a frustrating experience due to misunderstandings and communication gaps.

Client:
Our client is Levern Currie (VT ISE [2] doctoral student and CEO of Dringo [3]). Dringo aims to create an innovative all-in-one ordering kiosk to enhance accessibility by incorporating touch, hand gestures, and audio-based ordering, catering to a wide range of customers. To participate in this project, it was necessary to sign a Non-Disclosure Agreement (NDA). Therefore, this report will not disclose any specific details about the implementation or any other information considered confidential by Levern Currie.

Motivation:
The motivation behind this project is to refine the audio-based ordering component, making it more intuitive and efficient. Specifically, it focuses on enhancing the interaction experience for deaf/hard of hearing (HOH), English as a Second Language (ESL), mute, and nonverbal customers. By leveraging our conversational AI model and natural language processing through GPT-4 Turbo [4], the system we develop aims to mimic human interaction, providing a seamless ordering experience for all customers, regardless of their language proficiency or physical abilities.

General Approach:
The approach involves integrating OpenAI's large language model [4] into our backend system to accurately understand and process customer orders across multiple languages (French, English, German, Spanish, and Chinese). This involves extensive research, development, and testing to ensure the system is adaptable and efficient for any fast-food chain and menu, ultimately aiming to improve inclusivity and efficiency in drive-through services.
**REQUIREMENTS**

**Features and Specifications:**

1. **Conversational Ordering System:**
   The system must accurately interpret customer orders, addressing complex requests, modifications, and preferences without the need for repetitive confirmations. The dynamic voice assistant will be capable of conducting natural conversations like those between a drive-through employee and a customer.

2. **Multilingual Interface:**
   The assistant must automatically detect and adapt to the customer's spoken language, offering support for key languages such as English, Chinese, German, French, and Spanish. Our AI model should then respond to the customer in their language. Additionally, the user interface, including menu displays, should dynamically update based on the detected language, ensuring a seamless experience for non-English speakers.

3. **Adaptive Communication Modes:**
   Beyond voice, the kiosk should support touch and hand gesture inputs, adapting to the user's preferred communication method. The system must switch seamlessly between modes, maintaining a consistent and accessible experience for deaf/HOH, ESL, mute, and nonverbal customers (already implemented prior to our project).

4. **Integration and Scalability:**
   The voice assistant must integrate smoothly with Drivingo's existing code infrastructure while being scalable to accommodate different fast-food chains and their specific menus. It also needs to be compatible with existing hardware and software frameworks to facilitate easy updates and maintenance.
5. **Natural Language Processing and Machine Learning:**

The system must utilize natural language processing and machine learning techniques to understand and process customer input accurately. This includes recognizing slang and menu-specific items. This requirement involves developing a model capable of continuous learning from customer interactions to improve accuracy and user experience over time. It should also be able to handle ambiguous or incomplete orders by asking clarifying questions, similar to a human employee. This feature is critical for ensuring the system can adapt to new menu items, customer preferences, and emerging language trends, maintaining a high level of service quality and customer satisfaction.

6. **Testing and Model Training:**

We will conduct thorough testing to confirm that the model comprehensively understands the various ways users might communicate their orders, covering a broad spectrum of languages and slang. The model needs to undergo continuous refinement and training based on real customer interactions to enhance its accuracy and adaptability.

**Expected Impact:**

The implementation of this project is anticipated to enhance customer service interactions in drive-throughs and beyond, making ordering processes more accessible and efficient for everyone. This system aims to bridge communication gaps and cater to a diverse range of customer needs and preferences.
ASSUMPTIONS

Voice Capture Quality:
Our client has instructed us to proceed with the assumption that the kiosk utilizing our application will perfectly capture user inputs and effectively eliminate background noise. This expectation sets the standard for the voice recognition system to operate flawlessly in diverse environments, ensuring clear and uninterrupted user interactions.
**DESIGN AND IMPLEMENTATION**

*Architecture Diagram*

(Figure 1 - Architecture Diagram)

**Frontend**

The frontend of the application is developed entirely in React.js [5]. It records user audio and converts it into text. After the user stops speaking for 1.5 seconds, the text is transmitted to the Flask [6] backend, where language detection and natural language processing take place. Upon receiving the response from the backend, the frontend can respond in any combination of three ways: updating the menu language to match the language spoken by the user, displaying the user's updated order on the left side of the screen, or speaking to the user in the same way a drive-through employee would. A graphical flow of data throughout the app can be seen in Figure 1.
**Backend**
The backend of our application, developed in Python, is crucial for initiating interactions with customers using the voice assistant. It acts as the central processing unit, handling multiple functions such as order management, maintaining chat history and context, interpreting results from OpenAI's NLP technology [4], and generating responses to user messages. The backend is also responsible for identifying what language is being used and syncing the frontend to display the menu in the detected language.

**Flask Server**
Our Flask [6] server integrates the entire application, enabling:
- Requests to a pre-trained machine learning model for gesture-based ordering.
- Interactions with OpenAI's APIs for natural language understanding (NLU) and natural language processing (NLP).
- Management of order processes through the Python backend.

**Python gesture recognition ML model**
Developed previously by Drivingo, this model translates hand gestures into commands, facilitating order placements on our frontend without verbal communication. This model enhances user interaction by enabling gesture-based commands.

**OpenAI API**
We utilize the assistant features of OpenAI's API, leveraging its large language model (LLM) for its robust language understanding capabilities. The API serves two main purposes:

1) Understanding user commands through natural language understanding: helps decipher user intentions such as adding, removing, or altering items in their order.

2) After processing the user's intent and values via the backend, it communicates the outcome back to the LLM. Depending on whether the action was successful, the LLM crafts the assistant’s response. This response is then conveyed to the frontend, reflecting updates in the backend and informing the user of any changes to their order.
Our project is structured around a microservice architecture, ensuring that each component—frontend, backend, and APIs—operates independently yet cohesively. We had to alter the existing frontend codebase and calls to the gesture recognition ML model (trained on hand images collected from the clients) to enforce this decoupled architecture. This separation of concerns allows each part of the application to focus on specific functionalities without overlap, reducing complexity and enhancing the maintainability of the system. The frontend is designed to be lightweight and pleasing to the eye. The backend handles data processing and service integration. We have created each file in the backend (see Table 1) to handle different functions. We have a file dealing solely with order management, another dealing with conversation management and context history, and one dealing with handling returned responses from API calls.

<table>
<thead>
<tr>
<th></th>
<th>Python File Purposes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>assistant_creation.py</td>
</tr>
<tr>
<td>2)</td>
<td>message_generate.py</td>
</tr>
<tr>
<td>3)</td>
<td>order_management.py</td>
</tr>
</tbody>
</table>

(Table 1 - Python File Purposes)

**Assistant_creation.py**
The primary goal of this script is to manage the creation and configuration of an OpenAI assistant tailored for a drive-through ordering system. The assistant is programmed to validate customer orders, ensuring all necessary details (food type, quantity, size) are included, and to interact with customers by guiding them through the ordering process. This script automates the setup process, making it easier to deploy and manage the AI assistant.
How an assistant is created in the file

Get_or_create_assistant function

This function handles the retrieval or creation of the OpenAI assistant. It checks if an existing assistant ID is available; if not, it creates a new assistant with predefined instructions.

Explanation:

- Global Variable Check: The function first checks if ASSISTANT_ID is set. If it is, it tries to retrieve the existing assistant using this ID.
- Exception Handling: If the assistant retrieval fails (e.g., if the ID is invalid), it proceeds to create a new assistant.
- Assistant Creation: If no valid ASSISTANT_ID is found, it creates a new assistant with specific instructions, ensuring it follows the guidelines for validating orders and interacting with users. The new assistant ID is then printed and should be updated in the environment file.

How a thread is created

Get_or_create_thread function

This function is responsible for managing the communication thread used by the assistant. It ensures that there is an active thread for interactions, either by retrieving an existing one or creating a new one.

Explanation:

- Global Variable Check: Similar to get_or_create_assistant, this function checks if THREAD_ID is set.
- Thread Retrieval: If THREAD_ID is valid, it tries to retrieve the corresponding thread.
- Exception Handling: If retrieval fails, it creates a new thread.
- Thread Creation: If no valid THREAD_ID is found, it creates a new thread and updates the THREAD_ID with the new value.
Additional Important Details About this File

Environment Variables:
The script relies heavily on environment variables (OPEN_AI_API_KEY, ASSISTANT_ID, THREAD_ID, and MODEL) for configuration. These variables need to be set correctly in the .env file for the script to function properly.

File Upload:
The script includes a utility function to upload files, which is essential for providing the assistant with necessary resources like the menu file. This is where the drive-through menu would be uploaded. Look at the function “upload_file”.

message_generate.py

The primary goal of this script is to manage the creation and configuration of an OpenAI assistant tailored for a drive-through ordering system. The assistant is programmed to validate customer orders, ensuring all necessary details (food type, quantity, size) are included, and to interact with customers by guiding them through the ordering process. This script automates the setup process, making it easier to deploy and manage the AI assistant.

How a response is generated in the file

generate_response function:
This function handles the retrieval or creation of the thread and generates a message for the assistant to use in its interactions.

Explanation:

- Global Variable Check: The function first checks if THREAD_ID is set. If it is not, it calls the get_or_create_thread function to create a new thread and updates THREAD_ID with the new thread's ID.
- Message Creation: It creates a new message in the thread with the user's input (message_body).
- Assistant Run: The function then calls run_assistant to process the message and generate a response.
Return Response: Finally, it returns the response generated by the assistant.

How a thread is created

get_or_create_thread function:
This function is responsible for managing the communication thread used by the assistant. It ensures that there is an active thread for interactions, either by retrieving an existing one or creating a new one.

Explanation:

- Global Variable Check: Similar to get_or_create_assistant, this function checks if THREAD_ID is set.
- Thread Retrieval: If THREAD_ID is valid, it tries to retrieve the corresponding thread.
- Exception Handling: If retrieval fails, it creates a new thread.
- Thread Creation: If no valid THREAD_ID is found, it creates a new thread and updates the THREAD_ID with the new value.

Additional Important Details About this File

Environment Variables:
The script relies heavily on environment variables (OPEN_AI_API_KEY, ASSISTANT_ID, THREAD_ID, and MODEL) for configuration. These variables need to be set correctly in the .env file for the script to function properly.

Assistant Request Processing:
The script includes the run_assistant function, which processes the messages in the current thread. It handles the assistant's function calls, waits for completion, and manages tool calls and outputs.

run_assistant function:

- Assistant Run Creation: It creates a run for the assistant using the specified thread_id and assistant_id.
- **Status Check:** The function waits for the run to complete by checking the status periodically.
- **Tool Call Handling:** If the run requires action (tool calls), it processes the required actions by building arguments for the tool calls and executing the necessary functions (e.g., `add_to_orders_list`, `list_user_current_order`).
- **Tool Call Map:** It constructs a `tool_call_map` to keep track of tool call outputs and submits these outputs back to the assistant.
- **Message Retrieval:** Once the run is completed, it retrieves the assistant's response from the thread messages and returns it.

**order_management.py**

The primary goal of this script is to manage the current orders in a drive-through ordering system. It handles adding, removing, and retrieving orders from a global list. Additionally, it provides functions to return the current order list and the menu.

**How an order is managed in the file**

`addToOrder2` function:

This function adds an order to the global order list. If an order item of the same type and size already exists, it increments the quantity instead of creating a new item.

**Explanation:**

- **Arguments Handling:** The function extracts `food_type`, `food_size`, and `food_quantity` from the input JSON. It sets default values if size or quantity is not specified.
- **Menu Item Validation:** It validates if the `food_type` exists in the menu by checking against `validMenuItems`.
- **Finding Matches:** It calls `findInOrderList` to find existing items of the same type in the order list.
- **Updating Quantity:** If a matching item with the same size is found, it increments the quantity. Otherwise, it creates a new order item.
- **Return Message:** It returns a message indicating the success or failure of the operation.
removeFromOrder function:

This function removes an order from the global order list. It handles various scenarios such as mismatched sizes, multiple matches without specified size, and over-removal quantities.

Explanation:

- Arguments Handling: The function extracts food_type, food_size, and food_quantity from the input JSON.
- Finding Matches: It calls findInOrderList to find existing items of the same type in the order list.
- Single Match Handling: If only one matching item is found, it checks the size and quantity before removing or decrementing the quantity.
- Multiple Matches Handling: If multiple matches are found, it ensures the size is specified and handles accordingly. If sizes do not match, it prompts for clarification.
- Return Message: It returns a message indicating the success or failure of the operation.

Additional Important Details About this File

Environment Variables:
The script does not directly use environment variables, but it is expected to be part of a larger system where such variables are used for configuration.

Order Management:
The script maintains a global order list and a menu description, which are used for adding, removing, and retrieving orders.

returnCurrentOrder function:

This function returns the current global order list. It provides a snapshot of all items currently in the order.

findInOrderList function:
This function searches for items in the global order list that match a specified food type. It is used internally by the addToOrder2 and removeFromOrder functions to find existing orders.
TESTING

We have run a suite of tests, which we expand on below, to ensure that our model works appropriately in various scenarios. We broke down our testing into multiple parts.

1) Testing the flow of the conversation and ability to recall
2) Testing the AI’s multilingual capabilities

We ran a suite of tests in English, French, Chinese, Spanish, and German to make sure that the AI model could effectively understand and respond in the language, seen in Figures 2 and 3.

(Figure 2 - Conversation in German)
The translation from Figures 2 and 3 is:

T - How are you doing today? My name is Abishek and I like the color red.
A - Hello Abishek! I'm doing well, thank you. Red is a really beautiful color! How can I help you today? Would you like to order something or receive more information about our menu?

T - Can you list the specials for today?
A - Assistant: Today we have some delicious specials on the menu! Here are some highlights:

1. All-Day Breakfast - A savory combination of eggs and bacon, perfect for any time of day.
2. Artery Clogger - A giant burger with double beef patties, bacon, cheese and a special sauce.
3. Heart Stopper - A lavish treat with triple beef patties, layers of bacon, cheese, and topped with onion rings.

. Ube Milkshake - A creamy milkshake with a unique taste and purple color, made from ube (purple yam).
4. Avocado Smoothie - A nutritious smoothie of avocado, milk, and honey, with a hint of vanilla.

T - I want a large Ube milkshake.
A - I have added a large Ube milkshake to your order. The quantity is set to 1 by default. Can I help you with anything else? Maybe something tasty with it?
T - What do you suggest?
A - I would definitely recommend eating something with a delicious Ube milkshake! Here are a few suggestions:
1. Bacon Burger - A classic burger with a beef patty, delicious crispy bacon and cheddar cheese.
2. Fried Chicken Meal - A complete meal with crispy fried chicken, mashed potatoes and coleslaw.
3. Chocolate Cake - A rich and moist chocolate cake with a decadent chocolate frosting.
4. Onion Rings - Crispy, golden brown fried onion rings, coated with beer batter.

T - I will take one large onion rings.
A - I have added large onion rings to your order, again in a quantity of 1. You now have a large Ube milkshake and large onion rings. Is there anything else I can do for you? Maybe another drink or an extra snack?

Note: Each “Waiting….” console output is equivalent to 1 passing second.
3) Testing whether the backend accurately stores the user’s order.

We tested our backend to ensure that the orders were being stored correctly. It successfully updates our “global_order_manager” list, which temporarily stores the current customer orders until the customer has completed their order (at which point it resets). This was successfully done with the correct food_item, food_size, and quantity being added to the list, as you can see in Figure 4. Here we see that the fries food_item was captured accurately in the list and the brownies were promptly not stored as they are not available in the drive-through.

(Figure 4 - Testing Backend Accurately Stores Order)

4) Testing whether the frontend reflects changes made to the backend after user commands

We ordered after turning the microphone option on. A live transcript was displayed while the user was speaking (this was only seen for testing purposes). Then, after the transcript was
processed by the backend, the order was updated as shown. In the example in Figure 5, the user said: “Hi can I please get three medium blueberry muffins”. The backend then responded, as shown in Figure 6. Our global order manager employs a hashmap that is used for quick lookup of the user’s order for addition, deletion, or modification of orders.

(Figure 5 - User Interface of the Application Showing Order)

The frontend then receives this response and is updated accordingly as shown in Figure 5.

(Figure 6 - Ordering 6 Medium Blueberry Muffins)
5) Random question testing

We tested our backend with some random questions that have no relation to drive-through ordering. After showing our client we were able to come to the conclusion that our replies were on par with a human drive-through employee manning the drive-through. Some responses can be seen in Figure 7.

(Figure 7 - Testing Random Questions with Assistant)
TESTING TIMELINE

Week 1: Setup and Initial Testing
We set up testing environments and decided on our roles and responsibilities. Abishek managed manual testing of the voice assistant with basic scenarios in English, focusing on standard orders, modifications, and cancellations.

Week 2: Advanced Scenario Testing
Abishek continued with complex scenarios in English, including simultaneous order modifications and handling unexpected user inputs. We asked the assistant to add items that were both on our menu and not on our menu. It was able to identify which one to add correctly, and relayed to the user that the other item was not available. Abishek started testing multilingual functionality, ensuring the voice assistant accurately recognizes and responds in different languages. Testing was also done on obscure questions, one of which is shown in Figure 8.

Week 3: Multilingual and Frontend Integration
Abishek tests edge cases in multilingual scenarios, such as accents and dialect variations. Charles ensures that the frontend correctly updates for all occurrences, such as language changes and real-time order updates.
**USER MANUAL**

This manual includes explicit instructions on how to use all features on the website interface.

*Initiation Process (How to Order on the Website Interface):*

Upon approaching the kiosk, you have three options to place your order:

1. *Hand Gestures:*

To order using hand gestures, face the screen, and use specific gestures that correspond to different menu items. Pop-ups on the screen will guide you to understand which gestures to use. Figure 8 shows the kiosk understanding a user signaling for “burgers”.

(Figure 8 - User Interface Example for Gesture Ordering)
2. Voice Recognition:

If you prefer to use voice recognition, press the microphone button located on the kiosk. Once the kiosk is activated, you can start speaking to the voice assistant to place your order. In Figure 9, the user asked: “I want 4 large deluxe burgers, 2 large french fries, and 6 medium iced teas”, and the kiosk was updated accordingly as shown.

(Figure 9 - User Interface After Speaking you Order)
3. **Touch Screen:**
Similar to using a smartphone, you can also order by directly touching the screen options on the kiosk. Simply tap your selections from the menu shown in Figure 10 to build your order.

(Figure 10 - Default Order Screen)

**Language Selection:**
When using the kiosk, you can select your preferred language in two ways. If you prefer manual selection, choose your language directly from the touchscreen options, which will update the menu screen accordingly. You can also just start speaking in your desired language; the voice assistant is equipped to automatically detect and respond in the language you are using, ensuring a seamless interaction. For example, if you started speaking in Spanish, the menu would be updated accordingly as shown in Figure 11.
Confirmation and Payment:

When you're happy with your order as it is displayed, click the “finish order” button that is located under where the current order is displayed. After clicking, the screen will appear as it does in Figure 12.
Additional Clarifications:

Users have the flexibility to return to any previous step in the ordering process before final confirmation to make changes.
**DEVELOPER MANUAL**

**Important Information**

There are two main components that make the whole application work: drivingo_frontend and drivingo_backend. These are our two codebases for the frontend and backend. Both need to be running for the application to work properly. The frontend can run without the backend but this will cause errors if trying to order via microphone.

The .env file has a key called **OPENAI_API_KEY**. This must be properly set before running the backend. This is the key to use OpenAI API calls [8].

**Cloning and running frontend from github repository -**

Run: npm install --legacy-peer-deps
Then run: npm run build:client
Lastly run: npm run start:client
After running this, the frontend for the website’s frontend should be running on your localhost port 3017.

**Cloning and running backend from github repository -**

Make sure to have installed any version of Python over 3.0.
Run the script setup.bat - “./setup.bat”
Activate the newly made virtual environment - “./venv/Scripts/activate”
Finally, run the main file python.exe “\main.py”
**Backend Implementation -**

Our backend is exposed as a Flask API. The frontend can make calls to the backend using the endpoint /orderFoodThruAudio. Figure 13 has the code that exposes our backend as an API call.

```python
@app.route('/orderFoodThruAudio', methods=['GET'])
def process_user_input():
    print('HIT')
    print(f"REQUEST: {request.args.get('transcript')}"
    user_input = request.args.get("transcript")
    print(f"Current user input is: {user_input}"
    if user_input is None:
        return {"error": "No transcript provided"}, 400
    bot_response = message_generate.generate_response(user_input)
    #This returns the bot response <string>
    current_order = order_management.returnCurrentOrder()
    #This returns the order in a list of jsons. [{food_type:fries, food_size:small, quantity:1}, ...]
    #Vocal response: <response>, order: <current_order>
    response = {"vocal_response":bot_response, "order":current_order}
    print(f"Driving: {response}"
    return response, 200
```

(Figure 13 - Flask Endpoint that Handles Audio Ordering)
To create our assistant’s response we first call the `generate_response` function. The call is made using OpenAI’s API, as seen in Figure 14.

```
def generate_response(message_body):
    global THREAD_ID
    if THREAD_ID == "None":
        # TO DO: UPDATE THREADS FOR NEXT ORDER ON .env
        thread = assistant_creation.get_or_create_thread()
        THREAD_ID = thread.id
        message = client.beta.threads.messages.create(
            thread_id=THREAD_ID,
            role="user",
            content=message_body,
        )
    new_message = run_assistant(THREAD_ID)
    # print("Drivingbot:", new_message)
    return new_message
```

(Figure 14 - Creates/Updates Thread Responsible for Processing Order)

This is where we create a conversation thread for every customer.

Once a thread is made, the user input gets passed to the openAI assistant via the `run_assistant` function, which can be viewed in Figure 15.
Inside this function, the thread is run and we wait for GPT 4-turbo to generate a response. If the thread’s run status is not completed and is instead equal to “requires_action”, that means that action cycles must be triggered on our backend to either add or modify order items. This is the action cycle for adding to the order.
If there are multiple entities in a single user input (2 or more food_types), we have an iterative approach to deal with the tool_calls returned to us. This can be seen in Figure 16. This way of dealing with required openAI tool_calls (tool_calls are run states from openAI’s API telling us that we need to handle something before they can continue with the generation of the output on their end) makes it easy for us to manage each required tool_call one at a time through a for loop.
LESSONS LEARNED

One of the primary challenges was dealing with high variance in training the Keras neural network base of Rasa AI [7]. The lack of sufficient training examples—about 250 for each intent—resulted in an accuracy of only 75% in detecting intents. This was inadequate for our system's needs, as accurately detecting intents was crucial for backend functionality. Moreover, the model struggled with extracting the correct features from complex user prompts. For instance, an input like, "Can I get 3 cheeseburgers. And 2 fries. Actually, make it 5 small burgers," should ideally result in a detailed breakdown of items and quantities, but the model frequently failed to achieve this.

To address these issues, we considered two potential solutions: increasing the number of training examples, and leveraging OpenAI's GPT-4 LLM API for intent and feature extraction. While we explored both approaches, the second solution proved to be far more efficient and reliable. Implementing the GPT-4 LLM API significantly improved our system's performance, exceeding our expectations in terms of accuracy and speed.

Through this process, we acquired substantial new knowledge and skills. We learned how to address overfitting in neural networks, particularly for complex, real-world applications. We also succeeded in separating the processes of intent detection, feature extraction, backend order management, and user response generation using the LLM. Additionally, we gained experience in managing threads and maintaining conversational context while making API calls to the GPT-4 LLM.

We underscored the importance of agile problem-solving, effective communication, and continuous learning. Regular meetings with our client and among team members were crucial for promptly addressing challenges and adapting our strategies.
## TIMELINE

<table>
<thead>
<tr>
<th>Week 1</th>
<th>• Plan out milestones and timelines for the rest of the project.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 2</td>
<td>• Research NLP libraries [9] and technologies that we can use to comprehend user input.</td>
</tr>
</tbody>
</table>
| Weeks 3-4       | • Basic implementation of the RasaAI dialogue management system.  
                  • The first version of the NLP model will be trained on the hardcoded dataset that was used for the touch and gesture ordering menu that we were given. |
| Weeks 5-6       | • User testing plan (test extensively within ourselves).          |
| Weeks 7-8       | • Conduct integration testing to ensure all components work together seamlessly.  
                  • Make necessary adjustments to the system based on test results. |
| Weeks 9-10      | • Research and implement libraries and technologies that can detect the language the user is speaking, and update the website accordingly [10].  
                  • Conduct testing with Google Translate to mimic people speaking other languages to make sure the system works just as well as it did at the end of milestone 1 with the new languages. |
| Weeks 11-12     | • Use OpenAI’s API to generate a response that confirms their order. If the user says something it cannot recognize, notify the user. |
| Weeks 13-14     | • Conduct testing to make sure the LLM generated response is helpful for the user. It should confirm the order with |
the user and ask for confirmation. Refine the model to account for this if necessary.

| Week 15 | ● Client demo |

(Table 2 - General Timeline)

Week 1: Planning Milestones and Timelines
In the first week of our project, we focused on planning to set the foundation for a successful development process. We outlined the key milestones and established timelines for each phase of the project. This involved breaking down the project into manageable tasks, estimating the time required for each task, and assigning responsibilities to team members. By defining clear goals and deadlines we aimed to complete this goal by our given date. Table 2 summarizes how we approached this planning process.

Week 2: Researching NLP Libraries and Technologies
The second week was dedicated to researching various natural language processing (NLP) libraries [9] and technologies that could be utilized to comprehend user input effectively. Our goal was to find the most suitable tools that could handle multilingual input and provide accurate responses. This research phase also involved experimenting with different libraries to assess their performance and ease of integration with our existing systems. By the end of the week, we had a comprehensive understanding of the capabilities of the available technologies and selected the ones that best fit our project requirements.

Weeks 3-4: Implementing the Dialogue Management System
During weeks three and four, we focused on the basic implementation of the dialogue management system. We trained the first version of our NLP model using a hardcoded dataset provided for the touch and gesture ordering menu. This initial model served as a proof of concept, allowing us to test the basic functionalities of the system. This model was built using RasaAI (see the prior section of this report). This phase also involved refining the model to handle different types of user queries and ensuring it could maintain context throughout the
conversation. The primary challenge was to develop a system that could interpret user intent accurately and provide relevant responses.

Weeks 5-6: User Testing Plan
In weeks five and six, we developed a user testing plan to evaluate the system's performance. We conducted tests within our team and with our roommates to identify any issues or areas for improvement. This testing phase involved simulating various ordering scenarios and manually testing the backend to ensure the system could handle different types of user interactions. We gathered feedback on the system's accuracy, usability, and overall performance. This was not up to our expectations at all, so we switched from RasaAI as our dialogue management system and instead used OpenAI’s thread management features.

Weeks 7-8: Integration Testing and User Feedback
Weeks seven and eight were dedicated to conducting integration testing to ensure all components of the system worked together. We tested the interaction between the frontend, backend, and the NLP model to verify that data flow and functionality were consistent and accurate.

Weeks 9-10: Language Detection and Testing
In weeks nine and ten, we researched and implemented libraries and technologies that could detect the language the user is speaking. This involved integrating language detection APIs and updating the website accordingly. We conducted testing using Google Translate to mimic users speaking other languages. Our goal was to ensure that the system could provide the same level of accuracy and responsiveness in multiple languages. This phase also involved refining the language detection algorithms and ensuring the frontend could switch languages seamlessly depending on the user’s voice input.

Weeks 11-12: Generating Multilingual Responses
During weeks eleven and twelve, we focused on generating responses that confirmed users' orders and narrated them back in the language they spoke, while minimizing the costs that would be incurred via calls to OpenAI’s assistant APIs. We implemented a feature that notified users if the system could not recognize their input.
Weeks 13-14: AI Response Testing and Refinement
Weeks thirteen and fourteen were dedicated to testing the AI's responses to ensure they were helpful and accurate. We focused on confirming orders with users and asking for confirmation. This phase involved refining the NLP model to improve its ability to understand and respond to user inputs. We conducted tests to ensure the system could handle various scenarios, including ambiguous or incomplete orders. Based on the test results, we made necessary adjustments to enhance the model's performance and ensure a seamless user experience.

Week 15: Final Adjustments and Client Demo
In the final week, we made any last-minute changes to ensure that we met client demands on the project with whatever little time we had. We demonstrated the assistant capabilities to the client and her team. By the end of this week, our goal was to have a system that our client would be happy with.


**PROBLEMS AND SOLUTIONS**

(1) Selection of Rasa AI [7]

We initially chose RasaAI to develop our assistant. This decision presented significant challenges. First, RasaAI required extensive training data to accurately determine user intent. We lacked the resources and time to compile a suitable dataset for effectively training Rasa's neural network. Additionally, managing conversation context with RasaAI proved challenging. Another issue was that the bot's responses were overly rigid and seemed hardcoded. We aimed for more human-like interactions from our AI model. After two weeks of training and testing the RasaAI model with a 250 example datasets for 3 different intents and exploring alternatives, we opted to develop our own AI model instead of using RasaAI’s boilerplate. We attempted to develop our Rasa model completely in English first. This was to make sure it was possible to achieve a smooth contextual conversational flow in a language we were comfortable with before delving into our multi-language requirements. We trained “intents” on Rasa using 150 examples for each type of intent. For example, for “adding menu items” we trained the model with simple examples like “Please add 2 burgers to my order” and more ambiguous ones like “Ummmm lemme get a burger”. After attempting to train this model and check its accuracy we realized that the intents were not being guessed correctly at least 40% of the time. Hence we switched to OpenAI’s function calling API instead. Our new model integrated OpenAI’s API SDKs, utilizing pre-trained language models for user intent classification and generating human-like responses. We designed a backend system to handle orders, order history, and conversational context. Our backend system is a simple in-memory storage that only tracks a single user’s current order. Once the user is finished, the order is reset and the system awaits the next user’s order.

(2) Unrelated user inputs and fabricated data

When we tested our AI model with unrelated queries such as the weather, it sometimes led to inappropriate or fabricated responses from the bot. For instance, the bot would invent weather conditions or menu items. These inaccuracies could cause confusion during production as non-existent menu items were being added to the backend order
management system. We implemented additional functions to address unrelated queries more effectively. We also solidified the AI’s understanding of what menu items were available. We implemented this by creating an intent called “all_other_intents”. When the user query was classified with this intent we make our assistant answer their query (if possible, otherwise inform that it does not have that capability) and subtly prompt it to do something else.

(3) Latency with data retrieval (Solution: RAG)

Attaching a file with menu information for the OpenAI assistant resulted in slow response times due to lengthy data retrieval processes. We resolved this issue by creating a pseudo retrieval augmented generation (RAG) model and enhancing our model via in-memory calls for menu item checks for validity checks and menu-related inquiries. Instead of including the menu in every OpenAI call, we instead passed it flags on whether an item existed or not by cross referencing it with an in-memory menu database stored as a list.
**FUTURE WORK**

*Enhancing the Assistant with Voice Response Capabilities:*

A next step could be to add voice functionality to our assistant. When someone speaks to it, it must do two things - modify the menu and also communicate a response back to the user.

Introduction of Voice Output: Building on our successful integration of voice recognition, future developers could implement voice response capabilities for the assistant. This advancement could enable the assistant to communicate with users audibly, completing the interactive voice experience.

Development Objectives: The main objective would be to allow the assistant to not only understand spoken commands but also to respond verbally. This feature could make interactions more dynamic and engaging, mimicking human-to-human conversation. Users would receive spoken confirmations and answers, which enhances accessibility and user-friendliness.

Technical Implementation: The implementation could involve selecting and integrating a text-to-speech (TTS) engine that can convert the assistant's textual responses into natural-sounding speech. The chosen TTS technology must support various languages and dialects, ensuring broad usability. Additionally, it must deliver clear and understandable speech that reflects the assistant's intended tone and context. To this end, research is needed on what TTS engine meets all of the needs. Our prior study led us to learn about Python libraries to use for this like Mistral and Fast Whisper.

Challenges and Solutions: One of the challenges in implementing voice output is ensuring the naturalness and appropriateness of the voice tone, especially in different interaction contexts. To address this, developers could experiment with different TTS engines, potentially customizing voice modulations to better suit the assistant's personality and the nature of the tasks it performs. Developers also could conduct user testing to gather feedback on the voice quality and make necessary adjustments.
Future Enhancements: Once voice output is successfully implemented, future enhancements may include emotional intelligence to adjust the tone based on the user's spoken inputs.
ACKNOWLEDGEMENTS

We would like to thank our client and Drivingo’s CEO, Levern Currie (lcurrie@vt.edu), for this amazing opportunity to help make ordering food at restaurants more accessible to all people. We are also grateful that the existing codebase we expanded upon was written well, and was clear and simple to build upon. Working with Drivingo throughout the Spring 2024 semester was a great pleasure, and we would not have been able to learn as much as we did throughout this project without the continued feedback and support from Levern Currie.
REFERENCES


