CS3237
Introduction to Internet of Things
AY 24/25 Semester 1

Group Project Report

Group 09

Tan Kai Xin, Nicole	A0237969E
Wang-Li Zehan	A0234099X
Nguyen Hong Ngoc	A0245629W
Dinh Quang Huy	A0239201R
Koh Hui Yi, Phylicia	A0238871U

Table of Contents

1. Introduction

In an era where technological integration into everyday life is paramount, the development of automated systems for service industries has become increasingly essential. The objective of this project is to innovate how services, particularly in fast-paced environments, can be optimised using robotic solutions. This report outlines the design and implementation of a robotic system designed to improve customer service efficiency through automation and real-time data processing.

1.1. Project Motivation

The motivation behind this project stems from the necessity to enhance customer service interactions and reduce the human error factor in order processing environments. Traditionally, taking orders in busy settings, such as restaurants or retail stores, is prone to inaccuracies and inefficiencies. By integrating voice recognition and real-time data handling into a robotic platform, we aim to streamline these interactions, making them more efficient, accurate, and scalable.

1.2. Problem Statement

The project seeks to develop a robot capable of navigating to a customer to take orders autonomously. Utilising advanced sensory technologies, including camera vision and microphone input, coupled with robust communication protocols via MQTT, the robot aims to interpret customer gestures and vocal commands accurately to process orders effectively.

1.3. Proposed Solution

The proposed solution is structured around a three-part ESP32 module system, designed to optimise the robot's interactive and operational capabilities. ESP32-1 module serves as the primary communication hub, interfacing with an edge computer that processes visual data from a connected camera. This camera system is programmed to recognize customer gestures indicating service requests, which are then communicated to ESP32-1 via MQTT for appropriate action, such as triggering movement or initiating interaction protocols.The ESP32-2 module is dedicated to mobility and navigation, controlling the motors for the robot's wheels and managing inputs from line detectors and ultrasonic sensors to navigate the service environment effectively.

The ESP32-3 module houses a TensorFlow Lite-based voice recognition model, enabling the robot to process and understand spoken orders. The results from this module are sent back to ESP32-1, which logs the information into an inventory management system through HTTP communication. This setup ensures real-time inventory tracking and maintains order accuracy, integrating complex data processing seamlessly across the modules to enhance service efficiency and customer interaction.

2. IoT System Vision

We will be designing an IoT system consisting of a robot that is able to on command receive a trigger signal from a restaurant user using an external monitoring system, such as . The trigger signal could be a waving of a hand by a restaurant customer, requiring manual intaking of food orders. This trigger signal once detected by the external monitoring system will trigger a mobile robot to move its starting position A to the customer’s location B using a wifi-based communication protocol, and upon reaching its designated location B, it will activate its voice recognition program on board the robot to take in food orders. Food orders declared by the customer will be recognised and detected using voice recognition on the robot, and communicated back over a wifi based communication protocol to an inventory dashboard of the restaurant, where the order will be directly sent to the kitchen for preparation and also assist in stock inventory prediction for the restaurant. After sending the order, the robot will move from the customer’s location B back to its original starting point A, awaiting new command signals from the customer.

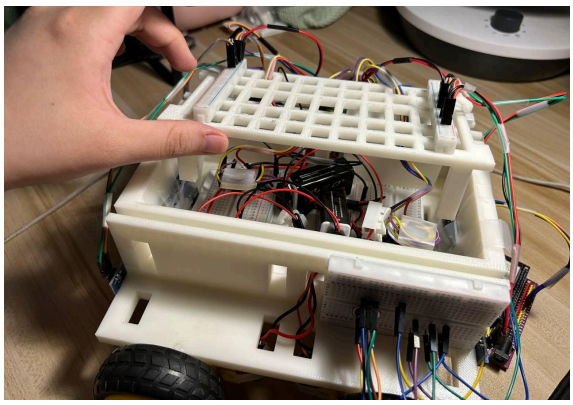
3. Robot Development

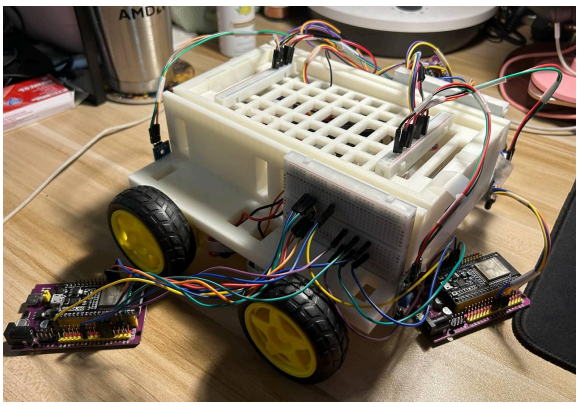
3.1. Design Considerations

Designing a robot for a restaurant environment requires addressing functionality, safety, and scalability. A compact design and bidirectional wheels enable precise navigation through tight spaces, while ultrasonic and infrared sensors ensure seamless movement in crowded areas. Safety is critical, with obstacle detection, avoidance systems, and automatic braking minimising collision risks. Modularity allows the robot to adapt to future use cases, such as food delivery, with interchangeable components and standardised interfaces for easy upgrades. Sufficient payload capacity and scalable software further enhance versatility. Constructed with lightweight, durable materials, the robot must handle various flooring types and withstand frequent use. These considerations ensure the robot remains reliable, safe, and adaptable for long-term needs.

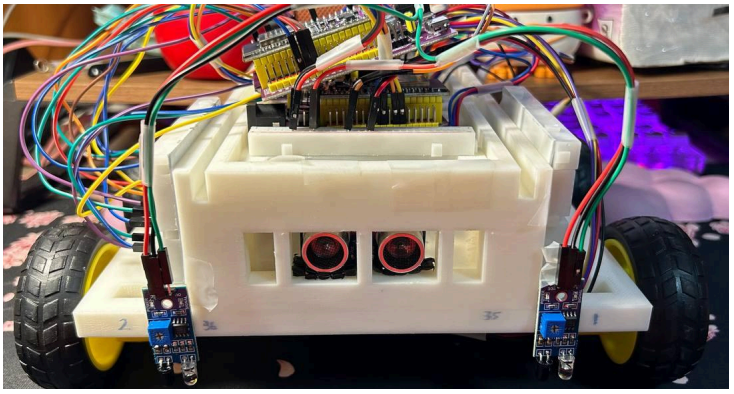
3.2. Robot Design

From our design considerations, we have decided to 3D print the robot's physical structure using PLA material. PLA is lightweight yet sturdy, making it ideal for withstanding the weight of batteries, microcontrollers, and sensors onboard the robot. Its durability ensures the structure remains reliable during extended use in a restaurant environment while maintaining the flexibility needed for future modifications or upgrades. Additionally, PLA's lightweight nature reduces overall strain on the robot's mobility components, enhancing efficiency and longevity.



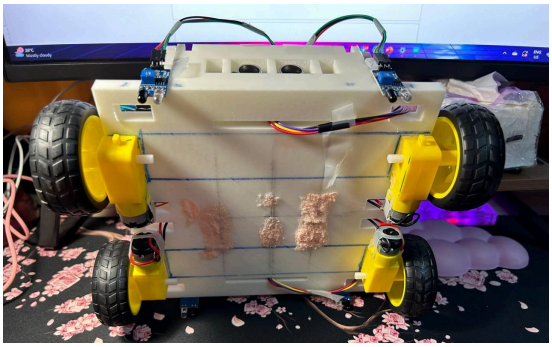
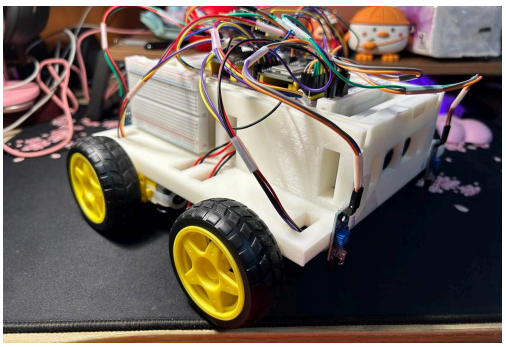


The robot's design incorporates various openings and multi-level housings strategically positioned to facilitate easy wiring and component connections throughout the structure. These openings ensure accessibility for maintenance and upgrades while reducing the overall weight of the robot.



The top of the robot base features a U-shaped support structure, designed to provide seamless integration for future enhancements. This modular approach allows additional housing or components to be easily stacked on top of the robot without requiring significant structural changes. This forward-thinking design ensures that the

robot can accommodate future functionalities, such as trays for food delivery or additional sensors, making it versatile for evolving use cases.

The bottom of the robot is built on a square-base design with dimensions of 20cm by 20cm. This design allows for optimal positioning of the wheels, enabling the robot to move bidirectionally and make sharp turns with precision. Such manoeuvrability is critical for navigating tight and compact spaces often found in restaurant environments. The square base ensures stability, while the precise wheel alignment allows the robot to operate smoothly even in complex layouts.

3.3. Hardware & Sensors

- ➔ 1 x ESP32 (ESP32-2)
- ➔ 4 x TT Motors (Bidirectional)
- ➔ 2 x L293D Motor Drivers
- ➔ 2 x 9V Rechargeable Batteries
- ➔ 1 x 5000mAh Power Bank
- ➔ 2 x Ultrasonic Sensors
- ➔ 4 x Infrared Line Tracking Module
- ➔ 4 x TT motor wheels

The ESP32-2 here is the ESP32 responsible only for sensor integration and movement. ESP32-2 will be powered by a 5000mAh power bank provided. Both left and right side movement of the robot will be independently controlled by 2 TT DC motors attached with wheels to 1 L293D motor driver which is externally powered by a 9V rechargeable battery. As the TT motors are bidirectional, directionality of the movement will be controlled by ESP32-2, which will receive inputs from both ultrasonic sensors and all IR sensor modules.

3.4. Robot Movement

The movement code controls the robot's mobility by coordinating motor control and sensor feedback, enabling it to perform actions such as moving forward, backward, turning, and stopping. Ultrasonic sensors are used to measure the distance to obstacles, ensuring the robot stops when an object is detected within a predefined threshold. Simultaneously, IR sensors provide precise feedback for line tracking and navigation, enabling the robot to adjust its course dynamically if it detects deviations. The IR sensors also pave the path for a straightforward movement of the robot to its designated location, eliminating the need for complex navigation algorithms to transit between locations.

The robot's **forward movement** is executed by powering both left and right motors to move in unison. If the robot encounters obstacles during this phase, it halts immediately, as determined by ultrasonic sensor feedback, and waits until the path clears before resuming. The robot also uses its IR sensors for alignment; for example, if the left sensor detects a deviation while the right does not, the robot turns slightly right to realign itself, and vice versa. Similarly, **backward movement** is managed by reversing the motors, using the same sensors for obstacle detection and path adjustment to return to the starting position safely, this eliminates uncertainties in executing U-turns, ensuring a more reliable trackback path. When required to make precise turns, the robot adjusts its motors asymmetrically—one side moves forward while the other moves backward—allowing the robot to pivot sharply. This functionality is particularly useful for navigating tight or narrow spaces. At any time, the robot can **stop** by deactivating all motor controls, ensuring a controlled halt to avoid collisions or when waiting for obstacles to clear.

The robot's movement is guided by state-driven phases, which structure its behaviour. In the **Forward State**, the robot moves toward a destination, constantly monitoring its path with sensors and adjusting its direction or stopping as needed. The transition to the **Recording State** occurs when the robot reaches a predefined stopping condition, such as detecting both IR sensors simultaneously crossing a threshold and also when ultrasonic sensors detect an obstacle below its threshold distance. During this state, the robot waits for an external signal before proceeding to the next phase. In the **Backward State**, the robot moves back to its starting position, applying the same sensor-driven navigation and obstacle handling logic as in the forward movement. To ensure smooth and precise movements, the code uses Pulse Width Modulation (PWM) to regulate motor speed. This allows the robot to move at a controlled pace, enabling it to make fine adjustments in its trajectory.

3.5. Challenges

Our robot movement system uses two TT motors per side, controlled by one L293D motor driver each. While the L293D provides flexibility, it requires substantial GPIO usage—six pins per side, totaling 12 GPIO pins for motor control alone. Adding four pins for two ultrasonic sensors and another four for IR sensors brings the total pin count to over 20, exceeding the ESP32's 19 usable GPIO pins. This creates a shortfall of at least one pin for the movement system. Additionally, incorporating the deep sleep protocol further complicates the design, as it requires external GPIO pins for ext0 interrupt wakeup, making the GPIO pin shortage even more challenging to address.

3.6. Solutions

To address the challenge of limited GPIO pins, the number of pins utilised for DC motor movement was reduced through a series of optimizations. First, instead of assigning individual pins to each motor, the motors on each side of the robot were grouped and controlled collectively. This means that the two motors on the left side share the same control pins for forward and backward movement, as do the two motors on the right side. By combining the control of paired motors, the total number of input pins required per side was minimised.

Additionally, the enable pins were consolidated. Instead of using separate enable pins for each motor, a single enable pin is shared across both motors on each side of the robot. This approach reduced the number of enable pins from four to two, significantly cutting down GPIO usage. The direction control was also streamlined by assigning just two GPIO pins per side for controlling forward and backward movements. This allowed precise motor control while minimising the number of required connections. Furthermore, efficient pin assignment was prioritised, ensuring each pin was used to its full potential. By utilising one enable pin and two direction control pins per side, the total GPIO usage for motor movement was reduced to just six pins, compared to the initial twelve pins. This optimization freed up additional GPIO pins for other critical components such as ultrasonic sensors, IR sensors, and wake up protocols.

4. Computer Vision

Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from visual data, such as images and videos, and take actions or make recommendations based on that information. It involves various techniques like image processing, object detection, and recognition, often using machine learning models.

In our project, we aim to integrate computer vision to enhance automation in a restaurant setting. Specifically, we use computer vision to detect customers and direct a robot to interact with or serve them. This creates a seamless and efficient customer experience by leveraging visual data to trigger intelligent robot responses.

4.1. Hardware & Implementation:

ESP32-CAM and Development Board

The ESP32-CAM is a low-cost development board that integrates a powerful ESP32 microcontroller with a camera module. Key specifications include:

- ➔ Dual-core Tensilica LX6 microprocessor
- ➔ 520 KB SRAM and 4 MB external PSRAM
- ➔ Built-in Wi-Fi and Bluetooth capabilities
- ➔ OV2640 camera module with resolutions up to 1600 x 1200
- ➔ Support for SD cards for storage

This compact and versatile board is ideal for basic computer vision tasks, making it a suitable choice for prototyping in our project.

Initial Implementation: Object Detection with YOLOv3

Our first attempt at implementing computer vision used the ESP32-CAM with object detection powered by YOLOv3 (You Only Look Once, Version 3). YOLOv3 is a state-of-the-art, real-time object detection algorithm known for its speed and accuracy. Following some tutorials, we configured the ESP32-CAM to detect and identify objects using Python and OpenCV. [1]

This initial implementation helped us familiarise ourselves with computer vision pipelines but had limitations in real-world applications, such as detecting specific human activities or gestures.

Transition to Human Pose Detection

Recognizing the need for a more application-specific approach in a restaurant setting, we decided to shift our focus to human pose detection. Pose detection identifies key points on the human body, such as joints and limbs, allowing for recognition of gestures, movements, or postures. This method is more applicable for detecting customer presence and intent, enabling the robot to respond appropriately, such as approaching a customer who raises their hand to signal for assistance.

4.2. Computer Vision Algorithm

Our computer vision algorithm consists of two components: an Arduino sketch used to initialize the ESP32-CAM, and a Python script that enables pose detection using MediaPipe [2] and OpenCV [3].

Initialisation of ESP32-CAM

The Arduino sketch is created using the Esp32cam.h library [4], which was used in our initial implementation. The code begins by configuring the Wi-Fi settings, enabling the ESP32-CAM to connect to the network using the specified SSID and password. It then sets up a web server on port 80 and defines three camera resolutions: low (320x240), medium (350x530), and high (800x600). The **serveJpg()** function captures an image from the camera and, if the capture is successful, sends the image in JPEG format over HTTP. The functions

handleJpgLo(), **handleJpgHi()**, and **handleJpgMid()** are responsible for handling image resolution requests. They adjust the camera's resolution accordingly before calling serveJpg(), which was defined earlier, to send the image to the client.

In the **setup()** function, the ESP32-CAM will be configured and connected to the specified Wi-Fi network and will wait for a successful connection before proceeding. Following on, the server.on() statements associate URLs with the corresponding handler functions (handleJpgLo, handleJpgHi, handleJpgMid), while server.begin() starts the web server. Finally, the loop() function continuously processes incoming HTTP requests using server.handleClient(). Based on the request, the appropriate handler function is triggered to serve the image to the client.

As such, Arduino sketch initialises the ESP32-CAM by connecting it to a Wi-Fi network, starts the web server, and continuously captures frames from the camera. These frames are then sent to the client via the web server as a video stream. The stream can be accessed by navigating to the ESP32-CAM's IP address in a web browser on any device connected to the same network. Additionally, the server provides three resolution options—low (320x240), medium (350x530), and high (800x600)—to customise the image quality.

Pose Detection

Inspired by a video which uses MediaPipe to develop a push-up counter [5], the python script begins by first importing the necessary packages: cv2, mediapipe, urllib.request, paho.mqtt.client etc.

The **pose_detection()** function starts by initialising the Pose model from MediaPipe with confidence thresholds for detection and tracking. Thereafter, it will enter an infinite loop where it will continuously download the image stream from ESP32-CAM, decode the image into a format OpenCV can handle, convert the image from BGR (OpenCV default) to RGB (MediaPipe required format) and process the image with MediaPipe to detect human pose landmarks (e.g. shoulder, wrist, elbow etc.).

Following on, it will check if either the right or left wrist is above the corresponding shoulder. In the event if either wrist is above the shoulder, the MQTT message "START" will be published to topic "start_A/now" to signal that a customer has raised their hand so as to inform the robot to start on Action A (Forward State). In addition, the hand_raised flag will be changed to True. The loop will continue processing until either the key "q" is pressed or when a customer has raised his/her hand (hand_raised flag == True) where the video window will close and the loop breaks.

The **on_connect()** callback is triggered when the MQTT client has successfully connected to the MQTT broker and it will subscribe to the "parked/#" topic. On the other hand, the **on_message()** callback is triggered when a message is received on the subscribed MQTT topic ("parked/#"). If the received message is "Parked" (from ESP32-1), it would mean that the robot has returned to its original position after completing all its actions (A,B and C). As such, the hand_raised flag is reset to False and the function pose_prediction is called to restart the pose detection

Lastly, the MQTT client is configured such that the on_connect and on_message callbacks are assigned to handle MQTT events. The client connects to the MQTT broker at localhost, the function pose_prediction() is called to start the pose detection loop and client.loop_forever() keeps the MQTT client running which handles any incoming messages and maintains the connection to the broker.

Therefore, the Python script continuously monitors the human poses in the ESP32-CAM video feed. When the system detects that a customer has raised their hand (based on the relative position of the wrist and shoulder), it sends an MQTT message with the payload "START" to trigger the robot to move towards the customer. Once the robot starts its movement, the video window closes, and the loop breaks. The MQTT client then waits for a message from the subscribed topic "parked/#". When the robot returns to its original position after completing its actions (A, B, and C), it sends the message "Parked". Upon receiving this message, pose detection restarts, and the sequence of actions is initiated again.

4.3. Challenges

Transitioning from object detection to pose detection introduced several challenges:

Implementing Pose Detection
Pose detection algorithms, such as OpenPose or MediaPipe, require more computational resources than the ESP32-CAM can natively provide. This necessitated offloading processing to an external computer. Real-time pose detection demands low-latency data transfer and efficient coordination between the ESP32-CAM and the processing unit.

Environmental Factors
Variations in lighting and restaurant layouts made consistent detection difficult. Ensuring robustness in a dynamic environment with multiple people and overlapping objects was also a key difficulty.

Integration with the Robot
Aligning pose detection outputs with robot commands required fine-tuning communication protocols and minimising delays to maintain smooth interactions.

By addressing these challenges, we aim to create an efficient system that enhances customer experience through intelligent automation.

4.4. Solutions

To resolve the challenge of implementing pose detection, we have implemented another python script, as mentioned previously, to run the pose detection algorithm instead of running the entire algorithm in the arduino sketch. The arduino sketch is mainly used to initialise the ESP32-CAM where the video stream can be accessed by navigating to the ESP32-CAM's IP address on any device connected to the same network, where in our case, it refers an external laptop which will be running the python script. Therefore, this offloads the processing from the ESP32-CAM to the laptop, allowing for a more efficient pose detection.

For the challenge on environment factors, we had to ensure that the area that we would be conducting the pose detection has sufficient light so that the algorithm is able to capture the human poses through the video stream of ESP32-CAM.

Lastly, to integrate our system with the robot, we have decided to use MQTT communication protocol in our python script. To inform the robot to start on Action A, the MQTT message “START” will be published to topic “start_A/now” (A topic which ESP32-1 is subscribed to) to signal that a customer has raised their hand. On the other hand, the MQTT client is subscribed to the topic "parked/#" such that it will receive the message “Parked” when the robot has returned to its original position after completing its actions (A, B, and C). As such, this ensures a smooth interaction between the robot and ESP32-CAM, allowing for a more efficient customer service.

5. Voice Recognition

5.1. Hardware & Sensors

- ➔ 1 x ESP32 (ESP32-3)
- ➔ 1 x 5000mAh Power Bank
- ➔ 1 x INMP441 Microphone

5.2. Voice Recognition Algorithm

The voice recognition is carried out using a model trained with TensorFlow and runs on the ESP32 using TensorFlow Lite. The source code referenced is from github and is titled “Voice-Controlled Robot With the ESP32”. [6]

Data collection

We collected recordings for the commands both from our own voice recordings and voice recordings from the Internet. In addition to commands for our menu items—“chicken rice,” “salad,” and “fish soup”—we included other unrelated "nonsense words" to help the model distinguish important commands from irrelevant phrases. All recordings were sampled at 16 kHz, approximately 1 second in duration, recorded in mono channel, and saved in WAV format.

Preprocessing and Input

To ensure consistency, each audio input is standardised to a length of 16,000 samples, equivalent to exactly 1 second at a sampling rate of 16 kHz. We verify each file’s length and, if necessary, either trim a few hundred milliseconds from the end or pad it with additional milliseconds. Since all our recordings are already approximately 1 second long, these adjustments do not significantly impact the core data. For recordings in formats like .m4a, we use the Python library *soundfile* to convert and save them in WAV format.

After processing the audio files, we extract spectrograms to train our model. First, the audio is converted into an array of floats, scaled between -1 and 1. We then reposition the audio to ensure the spectrogram captures the most relevant data, avoiding silent parts, and augment the data by adding scaled random background noise. To create the spectrograms, we normalise the audio, generate the spectrograms, and reduce the number of frequency bins to a manageable level. This dimensionality reduction simplifies model processing, while dynamic range compression ensures smaller values are not overshadowed by larger ones, which is advantageous for neural network training.

Finally, we split the preprocessed data into three sets—for training, validation, and testing. Each data is shaped as (99, 43, 1), making it ready for input into our machine learning model.

Convolutional Neural Network (CNN)

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv_layer1 (Conv2D)	(None, 99, 43, 4)	40
max_pooling1 (MaxPooling2D)	(None, 49, 21, 4)	0
conv_layer2 (Conv2D)	(None, 49, 21, 4)	148
max_pooling3 (MaxPooling2D)	(None, 24, 10, 4)	0
flatten_3 (Flatten)	(None, 960)	0
dropout_6 (Dropout)	(None, 960)	0
hidden_layer1 (Dense)	(None, 80)	76,880
dropout_7 (Dropout)	(None, 80)	0
output (Dense)	(None, 4)	324

Total params: 77,392 (302.31 KB)
Trainable params: 77,392 (302.31 KB)
Non-trainable params: 0 (0.00 B)

We implemented a convolutional neural network (CNN) model using TensorFlow/Keras, specifically designed for audio classification tasks. Given our 2D spectrograms—visual representations of the frequency content of audio signals over time—and the proven effectiveness of CNNs in processing 2D structured data like images or spectrograms, this model was well-suited for our application.

The model architecture consists of two convolutional layers, two pooling layers, two dense layers, two dropout layers, and one flatten layer. The convolutional layers detect essential features such as edges, frequency patterns, and transitions in the spectrogram. The pooling layers reduce the spatial dimensions of the spectrogram, retaining critical features while lowering computational complexity. The flatten layer transforms the 2D feature maps from the convolutional layers into a 1D array to prepare them for the fully connected layers.

To prevent overfitting, the dropout layers randomly deactivate 10% of the neurons during training. The dense layers are used for classification: the first dense layer (hidden layer) captures high-level combinations of features extracted from the spectrogram, while the second dense layer (output layer) maps these features to class probabilities using a softmax activation function.

The model is trained with a batch size of 32 for 10 epochs, with each epoch comprising 564 steps. This setup balances computational efficiency and learning capacity for effective training.

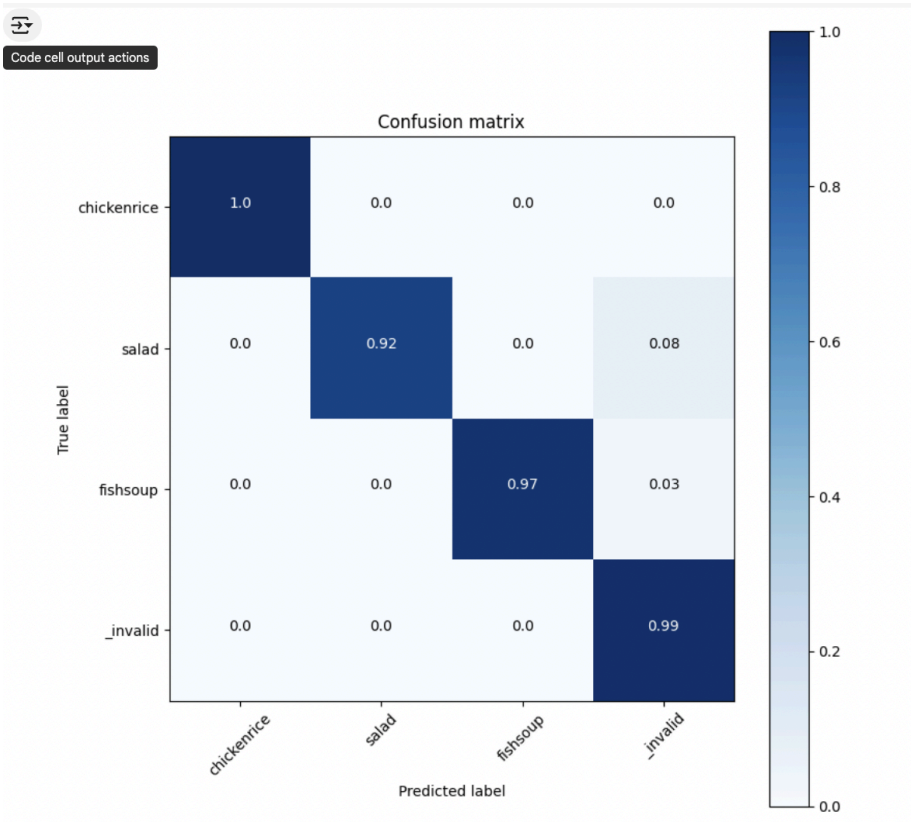
The final metrics achieved is:

Training Accuracy: 0.9977

- The model achieves an accuracy of 99.77% on the training set.

Validation Accuracy: 1.0000

- The model achieves 100% accuracy on the validation set.



Convert to C++

After obtaining the model, we convert it into tflite model first then convert out the tflite model to C code to run on our ESP32.

5.3. Challenges

There are two main challenges with our voice recognition service: data collection and microphone integration with the model.

Finding existing datasets for our specific command words/menu items—*"chicken rice," "fish soup,"* and *"salad"*—proved to be difficult.

Integrating the microphone into the system was challenging as it was hard to determine whether recognition issues stemmed from the hardware (microphone) or the software (machine learning model).

5.4. Solutions

We created our own dataset by recording voice samples ourselves. Additionally, we sourced recordings from the Internet. During preprocessing, we enhanced our dataset by duplicating the primary command recordings and introducing small variations to simulate a larger dataset from limited recordings (data augmentation).

To identify the root cause of the issues, we tested the microphone and machine learning model separately. Using an external app to simulate sound waves from captured voice, we discovered that our original microphone had hardware issues and could not capture voice correctly. This was resolved by purchasing a new microphone.

However, further debugging was required to configure the I2S microphone properly. Unlike most I2S microphones that read from the left channel, the INMP441 microphone reads from the right channel. This discrepancy required additional time to pinpoint and resolve, ensuring the microphone functioned as expected.

6. Inventory Prediction Dashboard

6.1. Inventory Prediction Algorithm

Database

We opted for a local PostgreSQL database to store all inventory information because of its simplicity and seamless integration with our system. The database includes a table named *inventory*, where each record represents the inventory for a specific week, identified by the *startOfWeek* field.

	id [PK] integer	startofweek date	chickenorder integer	fishorder integer	saladorder integer
1	1	2024-01-01	44	66	5
2	2	2024-01-08	44	68	6
3	3	2024-01-15	35	77	6
4	4	2024-01-22	33	78	7
5	5	2024-01-29	33	79	8
6	6	2024-02-05	27	80	9
7	7	2024-02-12	23	85	11
8	8	2024-02-19	13	100	10
9	9	2024-02-26	0	120	9
10	10	2024-03-04	1	110	10
11	11	2024-03-11	9	99	12
12	12	2024-03-18	12	95	13
13	13	2024-03-25	20	85	15
14	14	2024-04-01	21	82	20
15	15	2024-04-08	22	77	29

Data collection

We will gather data in real-time, when an order is placed after the voice recognition service identifies a menu item, combined with our prepared dataset. Specifically, once the voice recognition system detects the menu items in an order, an HTTP request is sent to the API endpoint `/api/addorder` with the quantities of chicken rice, fish soup, and salad in the order. This endpoint updates the inventory by incrementing the respective items for the current week by the specified amounts. If no entry exists for the current week, a new entry is created with the corresponding quantities for each item.

Preprocessing and Input

For each week's entry, we calculate the inventories for the previous week and the week before that for each item. Any null values are replaced with the mean value of the corresponding column. The resulting data frame is then split into training and testing sets for their respective purposes. Each record in the feature set (X) has a shape of (6, 1), consisting of the previous week's and the week-before-last's inventories for fish, chicken, and salad. Each target record (Y) has a shape of (3, 1), representing the predicted inventories for fish, chicken, and salad.

Random Forest Regression Model

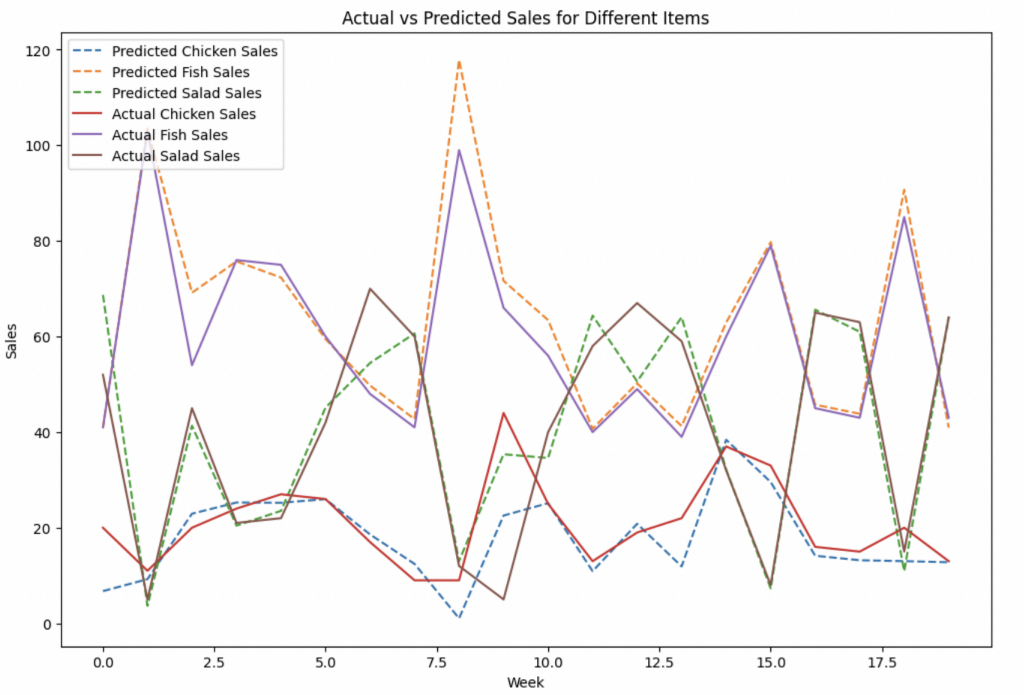
We utilise the RandomForestRegressor with MultiOutputRegressor to address multi-target regression tasks. RandomForestRegressor is chosen for its robustness against overfitting and noise, making it suitable for handling diverse and potentially noisy datasets. Additionally, it incorporates feature selection, contributing to better model interpretability. MultiOutputRegressor enables the handling of multiple target variables by fitting a separate model for each output, allowing us to perform multi-output regression effectively.

```
from sklearn.ensemble import RandomForestRegressor
rf_regressor = RandomForestRegressor()
model = MultiOutputRegressor(rf_regressor)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)
```

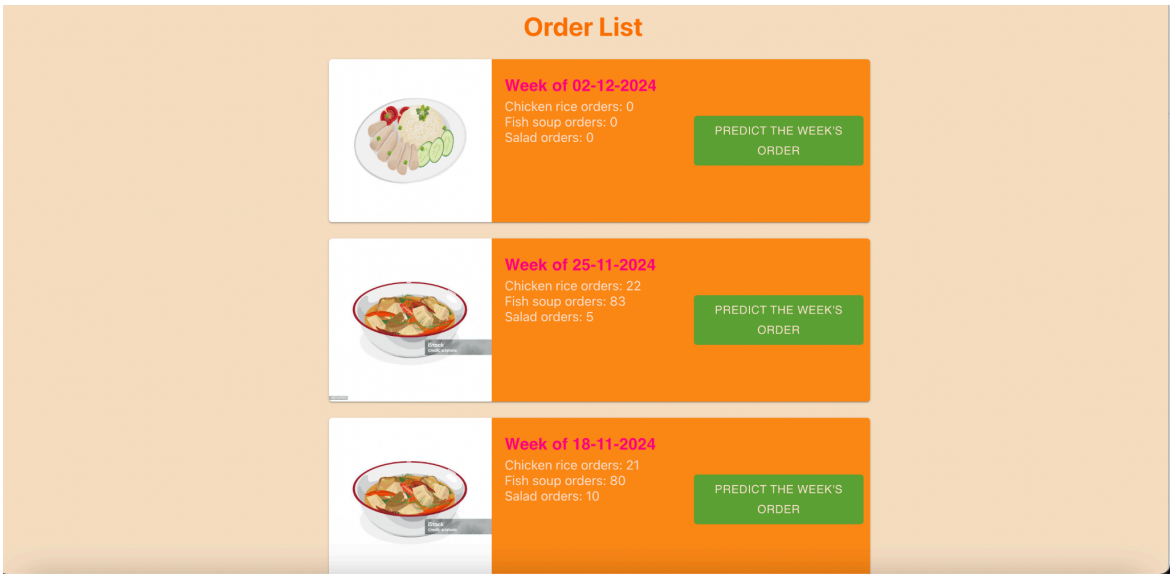
The final mean square score for Random Forest Regression is 0.719.



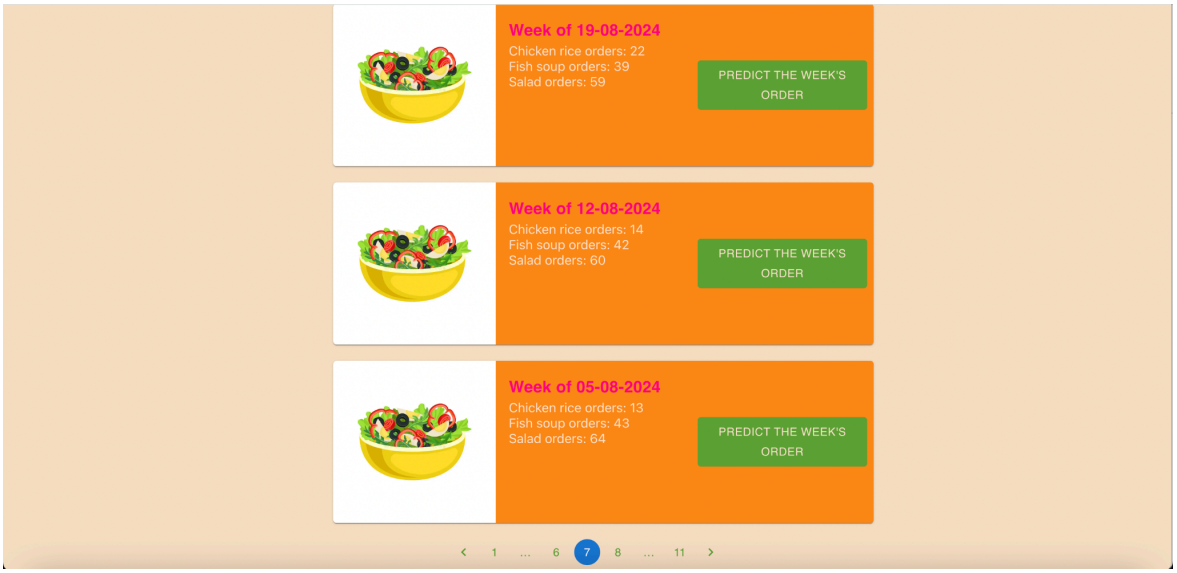
6.2. User Interface

The user interface of the application was developed using React.js, a JavaScript library for building user interfaces and MUI, a React Component library. Various components were created and libraries were added to increase the functionalities of the app.

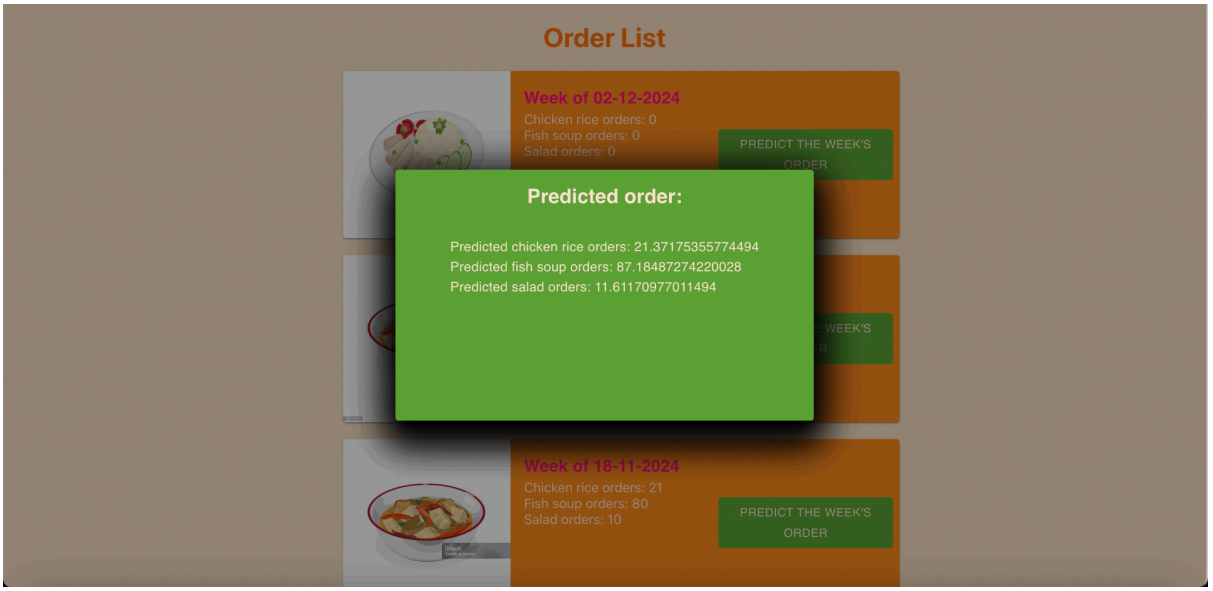
When a user opens the web application, a list of every week's inventory recorded will be shown. Each card will represent a week's inventory (denoted by the week of DD-mm-YYYY where DD-mm-YYYY is the start day of that week): how many chicken rice, fish soup and salad were ordered. On the leftmost of each card will be an image of the dish that is ordered the most for that week.



There is also pagination at the end of each page so that restaurant owners can easily navigate through inventories of many weeks.



On the rightmost of the card is a button, when it is pressed, a pop up will appear showing the predicted inventory for that week. This prediction is only meaningful for weeks in the future.



Shown in the image is the predicted order for week of 02-12-2024.

6.3. Backend

The backend is built using Flask, a lightweight web framework. Its responsibilities include connecting to the database, providing endpoints for the frontend to access data, and hosting the machine learning model for inventory prediction. Key endpoints include:
- /api/addorder: Updates the inventory.
- /api/order: Returns a list of inventories for all weeks.
- /api/predict: Provides predicted inventory for a given week based on a specified day.

6.4. Challenges

The main challenge is to test the entire pipeline thoroughly to identify and fix bugs across all components: frontend, backend, database and machine learning model.

6.5. Solutions

To address this, we applied the Separation of Concerns principle, ensuring each component focuses solely on its specific responsibilities. The frontend is designed to display data retrieved from backend endpoints, while the backend manages data retrieval from the database and hosts the machine learning model, which operates only with predefined inputs. The backend also needs to handle edge cases and returns appropriate HTTP status codes for undefined behaviours or errors.

7. Full IoT System Integration & Architecture

After breaking down the IoT system and building the separate sections individually, the separate sections have to be successfully integrated for the overall project to operate efficiently.

7.1. Power Consumption, Deep Sleep Protocol & Communication Challenges

7.1.1. Power Consumption and Power Management

In our IoT system design, ESP32-2, responsible for robot movement, faces potential power supply challenges due to the large number of GPIO pins in use. To evaluate the power demands, we conducted a stress test by setting all available GPIO pins to HIGH continuously and timing the runtime of the provided 5000mAh power bank. This test yielded an approximate runtime of **19.5 hours**, corresponding to a maximum current draw of **256mA**. Accounting for real-world power loss and current leakage (~30%), the realistic runtime reduces to **13.65 hours**, which is risky for long operational periods, such as a restaurant operating 12+ hours (e.g., 7am to 7pm). This necessitates power management strategies like Deep Sleep Protocols when the robot is idle or not triggered.

7.1.2. MQTT Communication and Deep Sleep Power Management Protocol

Deep sleep protocols are critical for ESP32-2 to conserve power, but integrating them with MQTT communication presents challenges. The ESP32-2 must wirelessly communicate with a mainframe via MQTT to receive start commands, transmit end commands, and send HTTP data to the inventory dashboard. However, when ESP32-2 enters deep sleep, its Wi-Fi module is disabled, making it incapable of receiving MQTT commands during sleep. MQTT requires an active connection for command reception, so ESP32-2 cannot wake up using MQTT messages. To address this, alternative wakeup mechanisms like **ext0 external GPIO interrupt** or **timer wakeup** must be implemented.

While this approach allows ESP32-2 to conserve power by waking only at specified intervals, it also introduces the risk of **missing crucial MQTT commands**. For instance, if ESP32-2 subscribes to an MQTT topic after a command has already been sent during its sleep interval, the message will not be retained, and the robot will fail to execute the intended task. MQTT communication is not inherently designed to store and forward messages for devices in deep sleep unless the broker is explicitly configured with retained messages, which may not always align with system requirements. To address this limitation, a second ESP32 (ESP32-1) dedicated solely to wireless communication will be incorporated.

7.1.3. Solution

ESP32-1 will be another ESP32 introduced to the system that operates in **Active Mode** and remains continuously connected to the network, listening for incoming MQTT commands and triggering ESP32-2 when signal is received. In this architecture, ESP32-1 acts as a communication gateway, handling all wireless communication and buffering commands for ESP32-2. By offloading the communication responsibilities to ESP32-1, the system achieves a balance between reliable command execution and efficient power management for ESP32-2, which remains in deep sleep until triggered. This dual-ESP32 configuration provides a simple and scalable solution to the challenges of integrating deep sleep with MQTT communication.

When deciding the type of wake up to be implemented on ESP32-2, **ext0 external interrupt** mechanism was first explored and implemented. Upon receiving a relevant command, ESP32-1 can use an external GPIO signal to wake ESP32-2. This setup ensures that ESP32-2 is woken up only when necessary, eliminating the risk of missed commands while optimising power consumption for the robot's movement and task execution. However, during the testing of **ext0 wake up** we encountered a new issue where, after approximately 20 seconds of ESP32-2 being in deep sleep awaiting for a trigger wakeup signal from ESP32-1, the power bank supplying power to ESP32-2 in deep sleep automatically powers off, possibly due to deep sleep drawing a current that is

too small for the power bank to register ESP32-2 as being a load, hence turning off. Once the power bank is turned off, **ext0** signals from ESP32-1 will be unable to wake ESP32-2 from deep sleep and turn the power bank on. To resolve this issue, we have decided to adopt the timer wakeup on ESP32-2 where upon every wakeup cycle ESP32-2 reads the GPIO pin connecting to ESP32-1, and if the signal is HIGH, it will proceed with the desired actions, else ESP32 will fall back into deep sleep.

7.2. Voice Recognition Model Integration Challenges

7.2.1. Lack of Memory Capacity for Voice Recognition Model

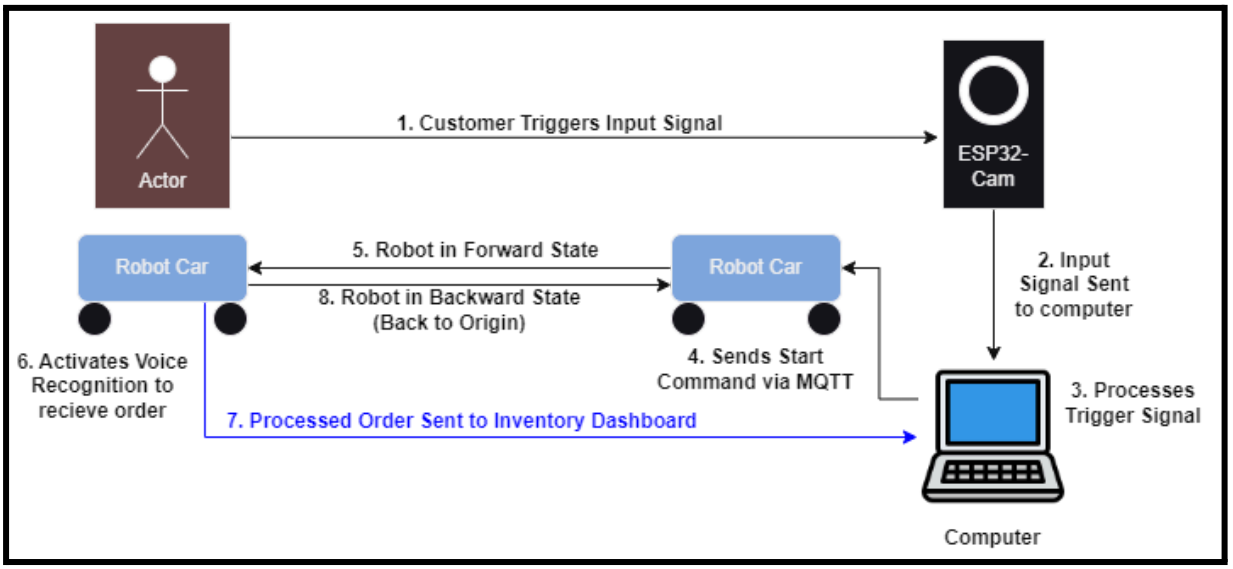
The primary challenge in integrating a comprehensive voice recognition model into the ESP32 revolves around its limited memory capacity. Even when using TensorFlow Lite, optimised for low-power devices, the memory required for running complex voice recognition algorithms exceeds the available resources on the ESP32. This limitation hindered our ability to integrate the microphone code with the communication code originally loaded on ESP32-1, as it exceeded the maximum memory by 3kB after including necessary libraries like the wifi library, particularly close to the demonstration deadline.

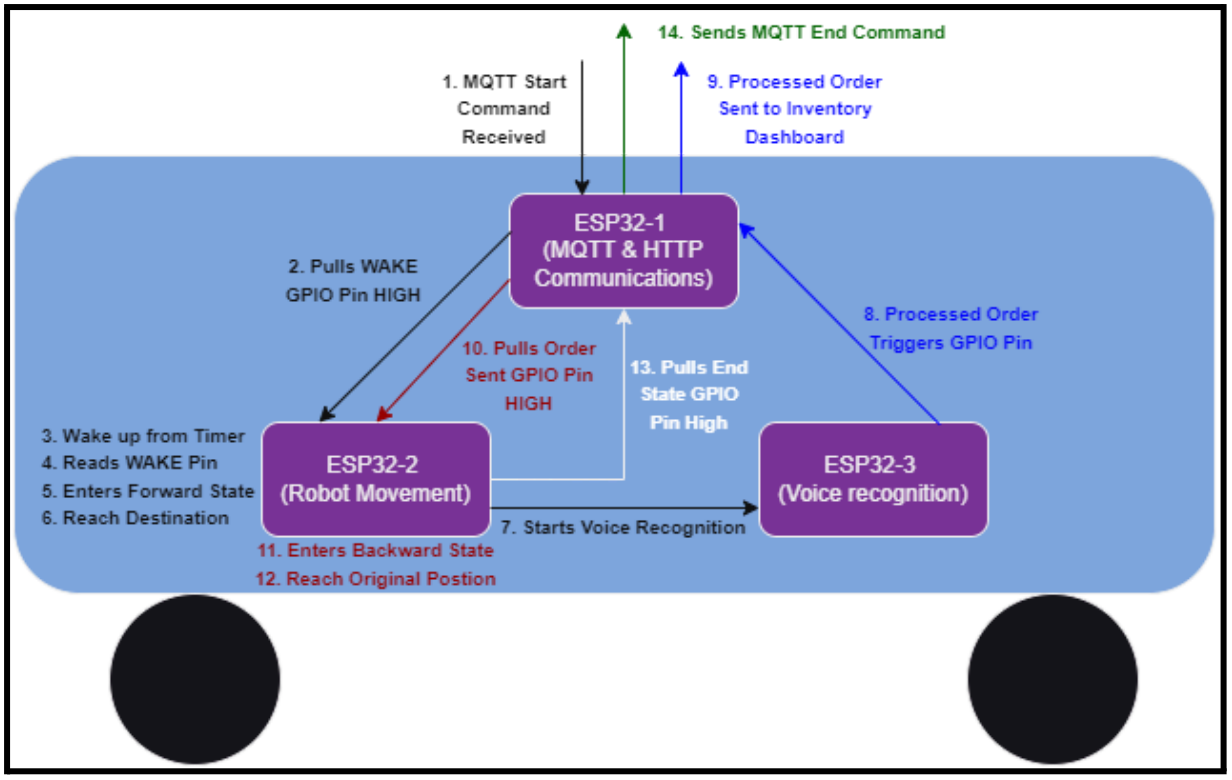
7.2.2. Solution

To address these challenges and avoid the risks associated with modifying and removing libraries under tight deadlines, a strategic decision was made to divide the tasks between two ESP32 boards. ESP32-1 was designated to handle main MQTT and HTTP communication, while ESP32-3 was dedicated to processing voice recognition with the TensorFlow Lite model onboard. This division allowed for the effective use of ESP32-3's capabilities in processing complex computations without memory constraints, outputting results through GPIO pins to trigger specific actions, such as updating the inventory system based on recognized voice commands.

This split-system approach ensures that the primary ESP32 board manages communication efficiently without the overhead of heavy data processing, which is offloaded to a secondary board. This solution not only resolves the memory overflow issue but also optimises the system's performance, making it feasible to handle real-time voice order processing effectively.

7.3. Full IoT System Architecture





7.4. Final Project Outcomes

Chuckie Bytes Video Demonstration: https://youtu.be/EtTRmmGdmg8

Chuckie Bytes Slides: https://docs.google.com/presentation/d/1lQQrmZFc0Ay9toCRatclPUknVhQEKp4Z2JOiT8LBVUE/edit#slide=id.ga4b3b7ab8f_0_15

Chuckie Byters GitHub Repo Link: https://github.com/RubyNguyen07/CS3237-project

8. Limitations

The present iteration of our autonomous order-taking robot is hindered by several hardware and software limitations that restrict its efficiency and scalability. Below, these limitations are elaborated upon alongside potential improvements that could be incorporated in future developments.

8.1. Memory Constraints

The ESP32 microcontroller is utilised for its compatibility with general IoT applications. However, it exhibits constraints in RAM, which becomes particularly problematic when tasked with running complex TensorFlow Lite models for voice recognition alongside computer vision processes.

While the current split-system approach addresses the immediate memory constraints and operational needs, it is recognized that utilising multiple ESP32 boards for different segments of processing may not be the most resource-effective solution in the long term. For future iterations and scalability, the adoption of a dedicated edge computing device is advisable. This device would handle intensive computations, leaving the ESP32 to manage real-time interactions and simpler processing tasks. This would streamline data processing flows, achieve faster response times, reduce power consumption, and maintain performance stability, especially under high customer interaction volumes. This shift towards a more centralised processing unit within an edge computing framework would allow for greater scalability and ease of integration with other systems, such as inventory management and additional IoT devices, providing a robust foundation for expanding the capabilities of our autonomous service robots.

Beyond the standardised development kit that we received for this project, we can transit to more capable microcontrollers with higher RAM capacities, such as the NVIDIA Jetson series or the Raspberry Pi 4, could drastically improve computational abilities and enable more robust multitasking capabilities without compromising system stability.

8.2. Sensor Integration

The integration of I2S microphones has been challenging, with recurring issues such as data loss or corruption. This impacts the accuracy of voice recognition, a critical feature for order-taking functionality.

Upgrading to more advanced digital microphone arrays with built-in error correction and enhanced data transmission stability could resolve these issues. An example includes microphones compatible with the XMOS VocalFusion platform, which are specifically designed for high-precision voice interactions.

8.3. Power Management

The system’s reliance on battery power limits continuous operation capabilities. Inefficiencies in power management protocols have been observed, particularly when the ESP32 enters deep sleep mode and fails to wake in time to respond to MQTT commands.

Implementing a more sophisticated power management IC, such as the Texas Instruments BQ25895, which supports higher efficiency and better power allocation, could enhance battery life and performance. Additionally, integrating solar panels for continuous charging could be explored to extend operational durations.

8.4. Communication Delays

Inherent delays in Wi-Fi communication are observed, especially during peak network traffic, leading to delayed responses from the robot. This not only affects operational efficiency but also impacts user experience.

Migrating from traditional Wi-Fi and MQTT protocols to more robust industrial communication standards such as OPC UA over TSN (Time-Sensitive Networking) could greatly enhance communication reliability and speed. This protocol is designed for high-speed industrial automation and could significantly reduce latency.

8.5. Hardware and Software Synchronisation

Synchronisation between hardware components and software commands sometimes fails, causing operational delays and errors in task execution.

Implementing a real-time operating system (RTOS) such as FreeRTOS, specifically configured for the ESP32, could enhance synchronisation through better task management and scheduling, ensuring that hardware and software components operate in harmony.

9. Future Development

This section delineates a structured development roadmap for the autonomous order-taking robot, aimed at systematically enhancing its capabilities and operational efficacy. The roadmap is segmented into distinct phases, each targeting specific technological advancements and functional enhancements to meet commercial deployment criteria.

Phase 1: Communication and Control Optimization

- Objective: Refine the foundational communication systems of the robot to ensure robustness and scalability suitable for commercial application.
- Initiatives:
    - Replace MQTT and pulling GPIO Pins with the Robot Operating System (ROS) to facilitate more sophisticated inter-device communication capabilities. ROS offers comprehensive tools for managing complex messaging and networking requirements essential for scalable robotic operations.
    - Integrate 5G communication technology to enhance data transmission speeds and minimise latency. This upgrade is critical for supporting a larger network of interconnected devices and robots, ensuring seamless operational synchronisation.

Phase 2: Advanced Sensory and Interaction Capabilities

- Objective: Augment the robot's sensory modules and interactive interfaces to support advanced customer interactions and navigate complex environments.
- Initiatives:
    - Implement facial and gesture recognition technologies using convolutional neural networks to enable personalised interactions and recognize human gestures, thereby enhancing user engagement and service personalization.
    - Develop a real-time collision avoidance system using LiDAR or advanced stereo vision technologies, which will allow the robot to navigate through dynamic environments safely and efficiently.

Phase 3: Voice Recognition Enhancement

- Objective: Elevate the precision and contextual understanding of the voice recognition system to reduce misunderstandings and improve interaction quality.
- Initiatives:
    - Utilise Generative Adversarial Networks (GANs) to artificially expand the training data set for voice recognition, thereby enriching the model's exposure to diverse linguistic inputs without the need for extensive data collection.

    - Enhance the system's Natural Language Processing (NLP) capabilities to process and respond to complex verbal orders accurately, facilitating more nuanced and effective communication.

Phase 4: Autonomous Service Functionality Expansion

- Objective: Broaden the robot's functional range to include comprehensive service tasks, transitioning from a simple order-taking unit to a full-service robotic waiter.
- Initiatives:
    - Integrate modular service accessories, such as trays and beverage holders, enabling the physical delivery of food and drinks and thus expanding the robot's role within service environments.
    - Implement algorithms for multi-robot coordination to optimise the collaborative operation of multiple robots in the same environment, enhancing efficiency and spatial utilisation.

Phase 5: Aesthetic and Brand Integration

- Objective: Tailor the robot's external design to align with corporate branding and enhance aesthetic appeal for seamless integration into service environments.
- Initiatives:
    - Design custom robot casings with digital displays for interactive communication and promotional uses, reflecting the brand's visual identity and enhancing customer engagement.
    - Establish interactive user feedback mechanisms within the robot to collect immediate customer reactions, providing valuable insights for continuous service improvement.

Phase 6: Full-Scale Commercial Deployment

- Objective: Prepare for the widespread deployment of robots in operational settings, ensuring robust performance during peak service periods and seamless integration with existing business processes.
- Initiatives:
    - Conduct pilot tests in operational environments to finalise the robot's design and functionality based on real-world performance data.
    - Develop staff training programs and maintenance protocols to ensure that the deployment of robots enhances service delivery and integrates efficiently with human staff.

Each phase of this roadmap is designed to build upon the previous, ensuring technological and operational maturation that culminates in a robot capable not only of fulfilling its primary role but also of significantly augmenting service capacity and customer satisfaction in the hospitality industry.

10. Bibliography

[1] E. Fahad, “ESP32 CAM with Python OpenCV Yolo V3 for object detection and Identification,” *Electronic Clinic*, May 31, 2023. https://www.electroniclinic.com/esp32-cam-with-python-opencv-yolo-v3-for-object-detection-and-identification/

[2] “Pose landmark detection guide | Google AI Edge,” *Google for Developers*. https://ai.google.dev/edge/mediapipe/solutions/vision/pose_landmarker

[3] “OpenCV Python Tutorial,” *GeeksforGeeks*, Jan. 30, 2020. https://www.geeksforgeeks.org/opencv-python-tutorial/

[4] J. Shi, “esp32cam: OV2640 camera on ESP32-CAM,” *GitHub*, May 10, 2023. https://github.com/yoursunny/esp32cam

[5] I know python, “Push-up counter using Mediapipe python,” *YouTube*, Mar. 29, 2022. https://www.youtube.com/watch?v=ZI2-Xl0J8S4

[6] “Voice-Controlled Robot With the ESP32,” *GitHub*, Sep. 22, 2023. https://github.com/atomic14/voice-controlled-robot