CG4002 Computer Engineering Capstone Project

"Furniture Layout Designer"

Design Report

Group B13	Name	Student #	Primary Component	Secondary Component
Member #1	Low Tjun Lym	[redacted]	Hw Sensors	Sw/Hw AI
Member #2	Gandhi Parth Sanjay		Sw Visualiser	Comms
Member #3	Davian Kho Yong Quan		Comms	Sw Visualiser
Member #4	Ong Wei Xiang		Sw/Hw AI	Hw Sensors

Table of Contents

Section 1 System Functionalities

Problem statement: Current Augmented Reality (AR) powered furniture viewers are limited in scale and functionality, as it only allows for basic features, which essentially overlays images of the furniture onto a background of the image. Our product aims to address this shortcoming by using an AR-powered mobile app and wearable.

Feature list:

- Voice commands to place, select and delete objects inferred by AI
- AR generated furniture displayed in the camera's view
- Hardware gesture controls to move and rotate the objects
- 12 degrees of freedom for movement and rotation
- Object Detection within the AR Scene
- Screenshot and save current AR Scene within application
- Synchronisation of shared gallery across devices with the same username

User stories:

- As a user, I want to be able to preview how a certain furniture looks like in my environment, so that I can better plan room layouts.
- As a user, I want to be able to change the orientation of the virtual furniture to be better able to position them with respect to the real environment.
- As a user, I want to control the AR application using a remote control, so that I can interact with the environment.
- As a user, I want the app to receive the gestures in real-time, so that the AR app feels smooth and natural.
- As a user, I want to be able to save the layout I have designed, so that I can refer to it later.
- As a user, I want my stored image layouts to be saved across devices.
- As a user, I want to use my voice to control what I select, so that remote controlling is easier.

Section 2 Overall System Architecture

2.1 Initial Proposed System Architecture

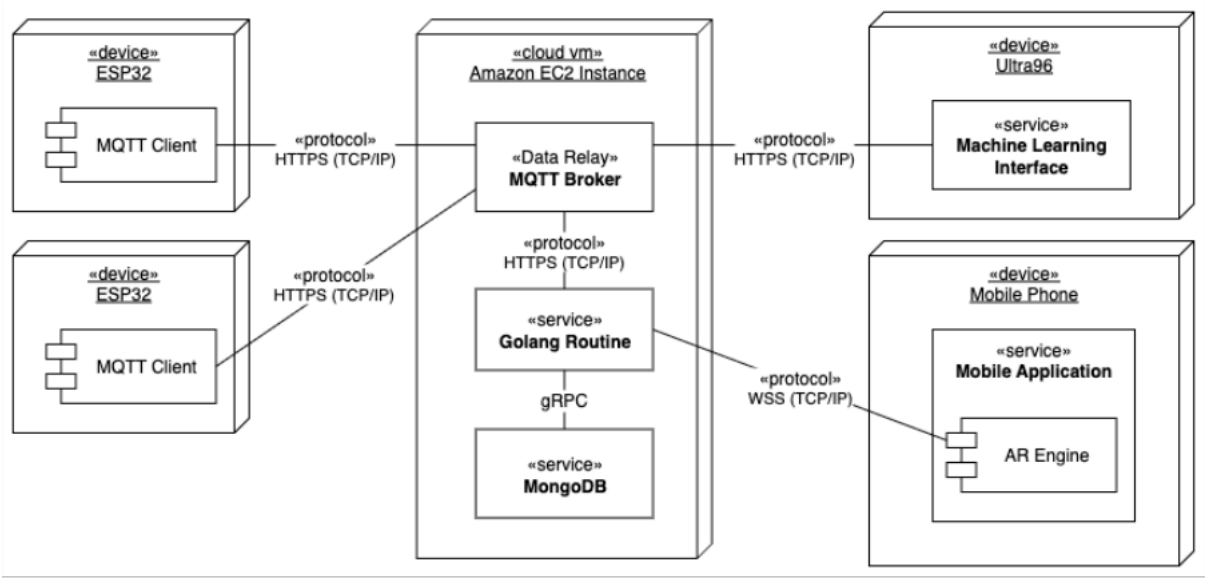


Fig 1. UML deployment diagram for initial proposed architecture

2.2 Final System Architecture

2.2.1 High Level System Architecture

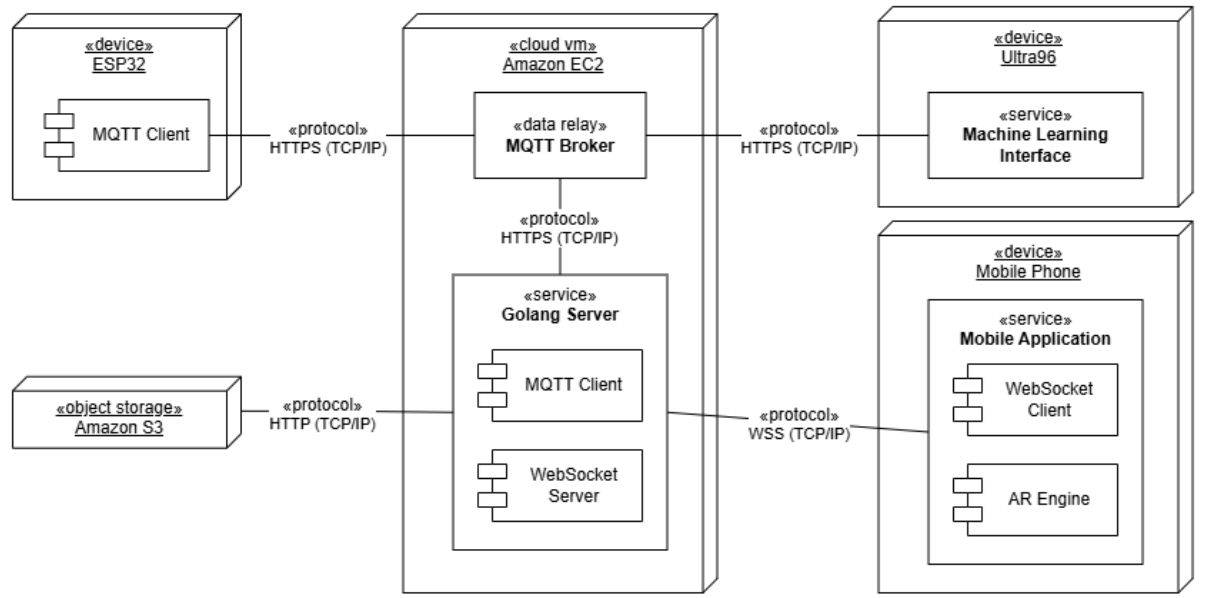


Fig 2. UML deployment diagram for final architecture

2.2.2 Changes Made

The initial proposed architecture involves the usage of dual ESP-32 for a dual user SharedAR experience within the same session, along with the use of MongoDB for user authentication. Due to issues during prototyping, as well as complexity and time constraints, SharedAR and MongoDB was not implemented in the final design. AWS S3 was used instead for object storage. Refer to Section 7 for more details on issues.

2.2.3 FireBeetle (ESP32)

The ESP32 interfaces with the MQTT Broker on a cloud EC2 instance, allowing direct connection regardless of which network the device is connected to. The dataflow from the ESP32 to the Ultra96 is unidirectional, and further communications breakdown will be discussed in Section 4.

2.2.4 Ultra96

The Ultra96 will read from the MQTT broker incoming gesture data (if any exists), and passes it through the ML model for inference, before publishing its gesture results (if valid) into MQTT for consumers.

2.2.5 Backend on AWS EC2

The backend is responsible for orchestrating all communications across all devices, as well as maintain state of the visualiser. The MQTT Broker, database and Golang Server will be running in separate containers managed by docker compose.

2.2.6 Object Storage on AWS S3

The S3 is used to allow persistent image storage for any users across devices. It communicates mainly with the Golang Server, which then relays user image information to the visualiser.

2.2.7 Mobile Phone / Software Visualiser

The software visualiser utilises mobile phones running the Unity-based AR application that renders the virtual furniture models with the physical environment. The software visualiser receives information from the Golang Server via WebSocket, which it then processes to change the display accordingly.

2.2 Main Algorithm

The user flow is as follows:

1. The user makes a voice command through the remote, voice data is recorded and published to MQTT via EC2.
2. The Ultra96 listens for incoming voice data via MQTT. The data is sent into the ML accelerator for inference.
3. With a result, the Ultra96 publishes the voice result to MQTT via EC2.
4. The Golang Service listens for incoming voice results via MQTT and forwards the data to an existing WebSocket connection between the Service and the Visualiser.
   a) If there is no such connection, the data is simply dropped.
5. Once the Visualiser receives a voice command result, it performs the respective function.
6. After an object has been selected, the user can then send movement data via the remote, which also publishes to MQTT via EC2.
7. The Golang Server listens for gesture commands via MQTT and forwards the data to an existing WebSocket connection between the Service and the Visualiser.
8. When the Visualiser receives a movement command, it performs the movement.
   a) If no object is selected, the data is simply ignored.
9. Once the user is done with placing and moving all the objects they want, they can press the screenshot button to capture an image of the Visualiser scene.
   a) If the user is not connected to the WebSocket, the image is stored locally in cache memory.
   b) Users can use the sync button to synchronize the image gallery between the Visualizer and the S3 service, allowing them to download and upload images.
10. Users can then save their images to their phone's gallery for further use.

2.3 System Description

The system comprises of a remote control used to transmit voice data to the Ultra96 for commands, or IMU input to the AR Visualizer through MQTT and a Golang server. The remote control contains the ESP32 as our MCU, and various sensors to allow these inputs to be detected and sent.

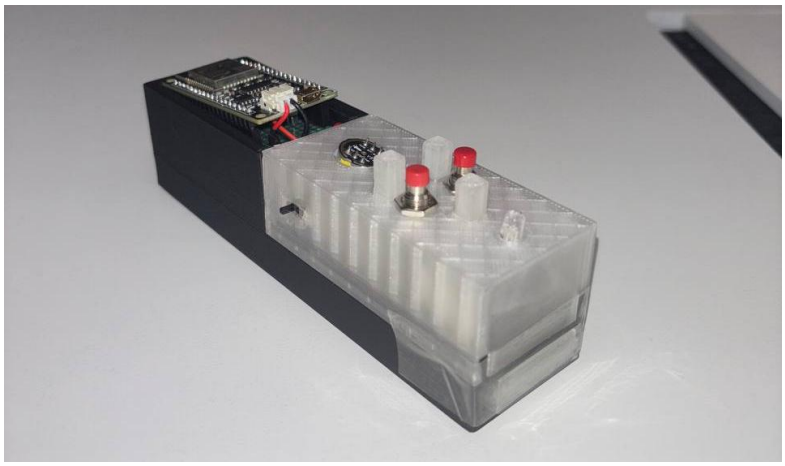


Fig 3: Final System

Section 3 Hardware Sensors

3.1 Introduction

The Micro Controller Unit (MCU) that we are using is the DFR0478 FireBeetle, which is a MCU powered by an ESP32. In the remainder of this report, we will refer to this MCU as ESP32. The main benefit it has is its ease of use, as well as a built-in Wi-Fi module. This allows for users to easily connect to other devices, such as a relay laptop or the Ultra96 FPGA directly.

The main sensors used are a MPU6050 IMU, a INMP441 Microphone and MAX17043 Fuel Gauge. The main actuators for user feedback are a RGB LED and an active buzzer.

The 5 main features of the app are used through buttons, to select, delete, move and rotate objects, as well as to screenshot the current view. For the rest of the section, the buttons that trigger the actions will be named *screenshot*, *select*, *delete*, *move* and *rotate* respectively.

The fuel gauge allows users to better estimate the remaining lifespan of the battery before it has to be recharged, so that they are aware of when they need to charge the battery.

Lastly, to power the entire system, we have chosen a Lithium-Polymer (LiPo) battery. This is due to its lightweight form factor, as well as its steady voltage throughout its discharge cycle.

In the next sections, we will discuss in greater detail the reasoning behind each component's choices, as well as the complete schematics and explaining how the components connect to one another. Next, we will then discuss the battery size before concluding with a pseudocode section of the main programme running within the ESP32.

The respective datasheets for the components are included in the appendix at the end of this report.

3.2 Design Choices

The first component to discuss is the ESP32. It contains 30 GPIO pins, which are multiplexed to perform other functions, such as I2C Communications, UART and so on. Wi-Fi is the main mode of communication between the ESP32 and other devices, which the communication will be outlined in future sections.

The MPU6050 IMU is chosen for its wide popularity and availability, which allows for us to source for the part, as well as read up on the documentation of this component. It comes with a 3-axis gyroscope and 3-axis accelerometer, which allows us to easily control the movement of objects in the AR world. The MPU6050 is also energy-efficient, drawing roughly 4mAh at 3.3V. We are also able to sample frequently within the maximum Serial Clock frequency. This frequency allows us to track the user's movements closely, without overwhelming the ESP32, as the I2C rail is shared with the battery manager.

5 pushbuttons are used in the ESP32, tied to each of the 5 functions above. The *screenshot*, *select* and *delete* are debounced to 4 seconds, to prevent flooding of the Visualiser and Ultra96 with input. The *move* and *rotate* are debounced at 100ms, so that the IMU data can be streamed to allow for smooth movement.

The battery source we have chosen is a LiPo battery, used for its lightweight form factor and high voltage discharge relative to its size. The LiPo battery also allows for convenient charging via the USB-C port on the ESP32. The JST PH2.0 output also allows us to use the DFR0563, a battery fuel gauge that is a low-power, compact IC that calculates battery percentage based on the current and voltage passing through the circuit and can relay the data to the ESP32 through I2C. The battery percentage is then fed into a function to blink the RGB LED at certain thresholds.

However, the main issue with using the DFR0563 is that the battery input has to be passed into both the ESP32 and the DFR0563 (which can be seen in the schematics in the next section). This requires a splitter, which I made for the project, as commercially available ones do not fit into the small form factor of our device.

3.3 Schematics and Pinouts

3.3.1 Schematics of the complete circuit

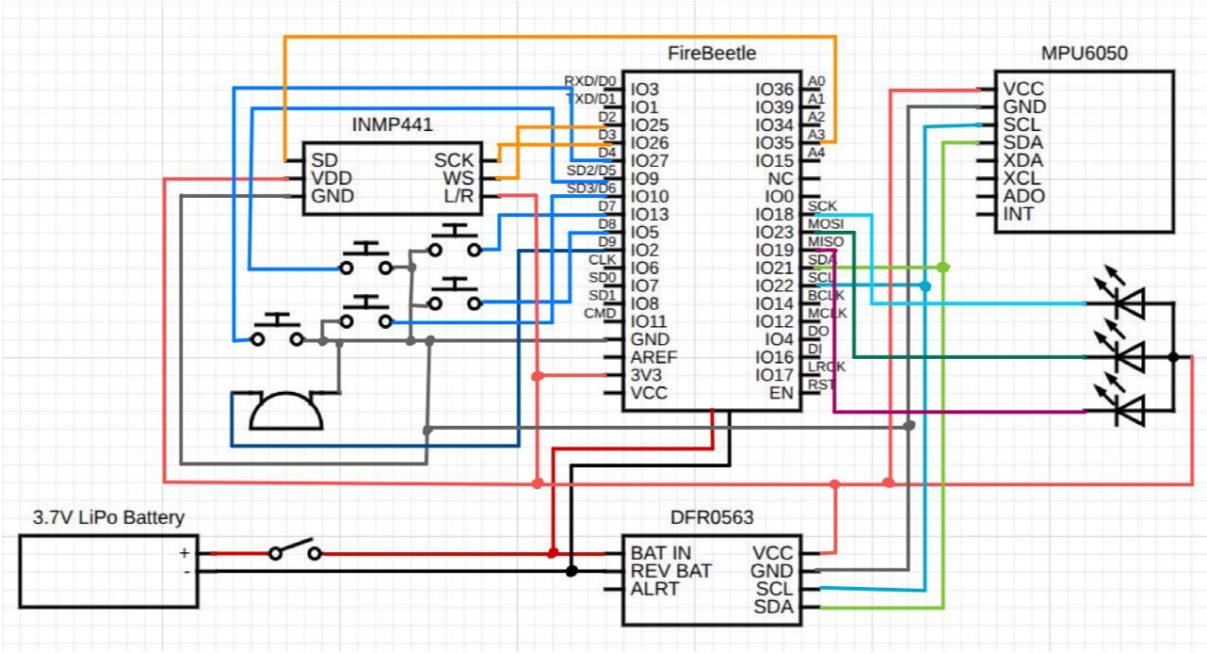


Fig 4: Final schematics of the complete circuit

The ESP32 will power most of the circuit via the 3.3V output, which is within the recommended operating voltage range for most sensors used, while the ESP32 itself is powered by the 3.7V LiPo battery, through its JST 2.0 port.

3.3.2 Pinout diagrams of devices used

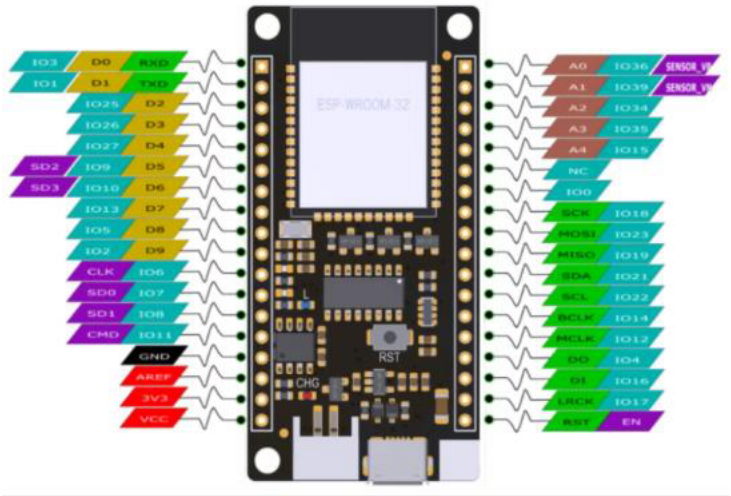


Fig 5. Pinout of DFR0478 ESP32

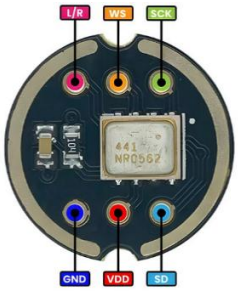


Fig 6. Pinout of INMP441

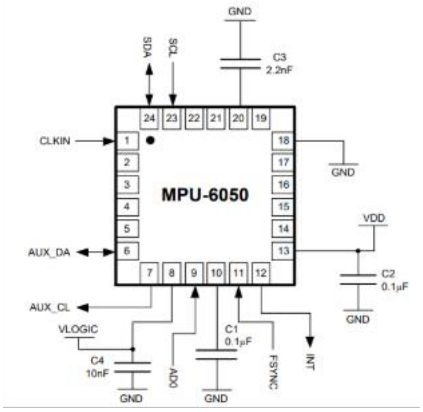


Fig 7. Pinout of MPU6050

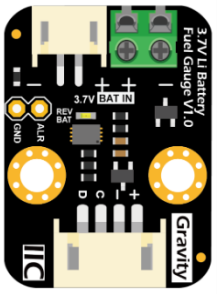


Fig 8. Pinout of DFR0563 Fuel Gauge

3.3.3 Pin connections of devices used

Component	Component Pin	ESP32 Pin	Remarks
MPU6050	VCC	3V3	
	GND	GND	
	SCL	SCL/IO22	
	SDA	SDA/IO21	I2C Address: 0x70
DFR0563	BAT IN	NIL	Connect to 3.7V LiPo Battery via JST2.0
	REV BAT	NIL	
	+	3V3	
	-	GND	
	SCL	SCL/IO22	
	SDA	SDA/IO21	I2C Address: 0x36
INMP441	VDD	3V3	
	GND	GND	
	L/R	3V3	
	WS	IO25	
	SCK	IO26	
	SD	IO35	Read-only pin
Pushbuttons		IO27	Screenshot
		IO9	Select
		IO10	Delete
		IO13	Move
		IO5	Rotate
Buzzer		IO2	
RGB LED	Red Pin	IO19	
	Green Pin	IO23	
	Blue Pin	IO18	
	VDD	3V3	

3.4 Power Management

The typical use case of the product will be roughly 2h, with the scenario of an Interior Designer showing their potential client a simulation of the room with various furniture.

Below is the operating current of the various components:

Component	Hourly Power Draw (mAh)
DFR0478	80mAh typical running power 240mAh for Wi-Fi Tx
MPU6050	3.9mAh
DFR0563	50-75μAh (round up to 0.1mAh)
INMP441	3mAh
RGB LED	20mAh per colour

As the RGB LED is usually minimally powered, we can estimate the power draw at 3mAh, as the battery checker code only flashes the LED on for 50ms every 30 seconds, and the LED is also only fully powered for 2 seconds when audio data is being recorded. However, since the RGB LED used is common anode, there is some leakage current, even when the LED is supposed to be turned off. This can be seen by a dim blue light flickering when the remote is idle.

For 2h of continuous use, total power required is

$$I_{total} = 2h(80 + 240 + 4 + 3 + 3)mAh$$

$$I_{total} = 660mA$$

As stated in the Initial Design Report, the initial power requirement of the proposed design was much higher, and thus a 1200mAh power source had been purchased. In the revised design, a battery of around 1000mAh would be sufficient, using the same 1.5x factor. However, the batteries were procured after the Initial Design Report, and thus we maintain the same battery.

3.5 Physical Prototyping

3.5.1 Main design choices

Unlike the initial design, the final design is made using a 3D Printer. The design shifted from a glove to a remote control in order to better protect the internal electronics from physical strain such as pulling on the wires and external damage. The design also had to be compact, so as to not inconvenience the user. As such, the two main considerations in this remote were size and user experience.

The first key design choice was to place the buttons relatively in the centre, so that the remote is more ambidextrous. The button layout is also shaped differently so that users can intuitively figure out the functions of the buttons.

In the button layout, the *screenshot* button is right at the top, as it is a standalone button with no complementary button. The *select* and *delete* buttons are the next two, using round buttons to differentiate them from the *move* and *rotate* buttons. The layout is done such that *select* is on the left and *delete* is on the right, aiming to replicate the familiar motions of a mouse using the left mouse button to select and right mouse button to delete. Similarly, the *move* and *rotate* buttons are the last row, in the square buttons. In this case, the same button shape can be used as *screenshot* as it is physically far apart enough to tell the different functions apart. *Move* is also on the left and *rotate* is on the right, as intuitively a mouse would use the left button for the primary action (i.e. moving) and the right button for a secondary action.

The RGB LED is also placed on the far end of the remote, both to indicate the top side of the remote and to give users visual feedback of certain actions being performed. For example, the LED lights up in a bright blue when voice data is being recorded, and also has a faint blue due to the common anode of the LED. This also helps to indicate that the remote is switched on and idle.

The INMP441 Microphone is placed as close as possible to the bottom of the remote, so that users need not move the remote as much in order for the microphone to pick up their voice data for *select* and *delete*.

Lastly, the power button is placed on the right-hand side of the remote. This is the one feature that caters to the right-handed majority, as it is more convenient to switch the remote on and off using the right thumb.

The biggest challenge of the design was fitting the many components into one singular device, especially on that was meant to be light, handheld and portable. In order to meet these requirements, we had various solutions, one of which was a use of a proto-breadboard, which we had used a handsaw to cut into smaller pieces to fit into the remote. Fig 9 shows a segment of the breadboard that was cut, with the metal side facing up to show the shorted holes. This was crucial in creating a common VCC and Ground rail, as many devices had to share the same Ground and VCC.

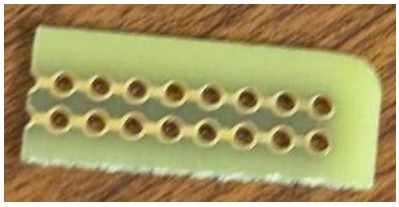


Fig 9. Part of a protoboard used for common rails

The ESP32 has header pins soldered onto a protoboard for easy mounting, as well as removal for software updates. The central area of the protoboard is free, and thus we have also soldered the IMU directly under the ESP32 to make it more compact. Additionally, a small gap was also cut into the protoboard to insert the buzzer, enabling the prototype to be smaller. This is illustrated in Fig. 10 below.

Fig. 10: Protoboard with the IMU and Buzzer soldered on

3.5.2 Initial CAD Design

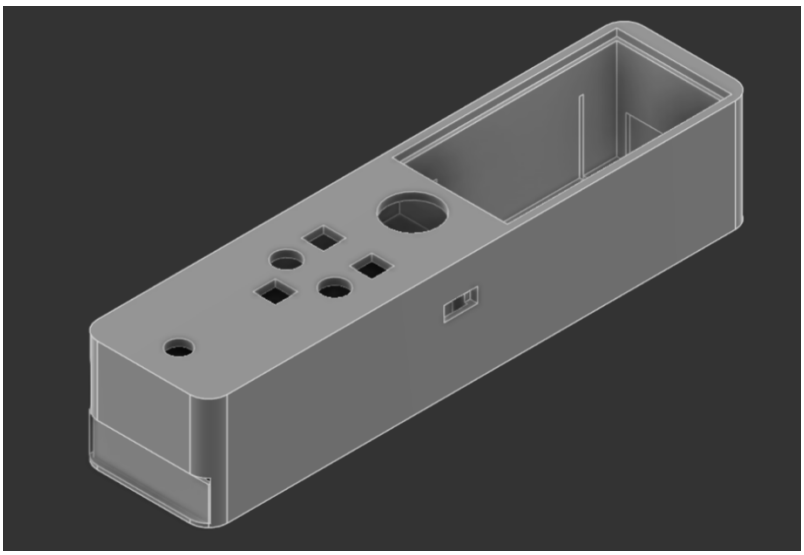


Fig 11. Initial CAD of remote

The first prototype of the remote was made with the above considerations, with the button holes at the top made to sit the buttons, measured using a vernier calliper for maximum accuracy.

The main internals were also segmented into three main chambers. one to contain the battery, one for the ESP32 and the last one for the peripherals Fig 12 below demarcates the 3 chambers. In the image, the peripherals chamber is highlighted in red, the ESP32 chamber in green and the battery chamber in blue. The ESP32 and Battery chambers have arrows indicating the location that they are supposed to be placed in.

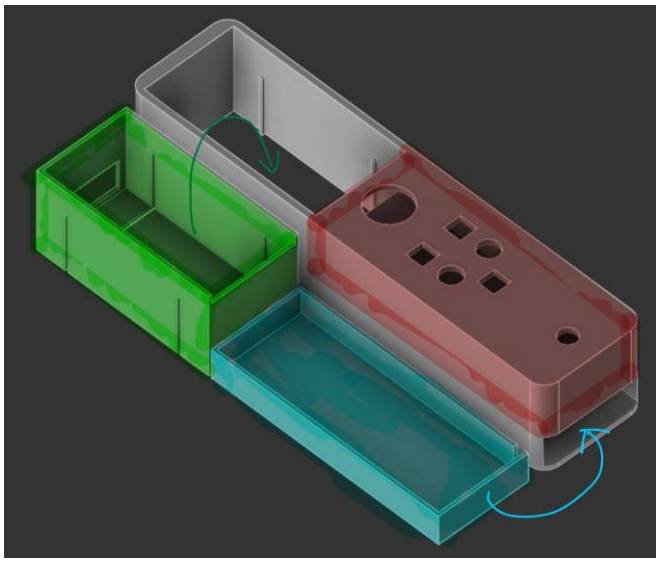


Fig 12: The three main chambers of the remote

The peripherals chamber was also designed to have a large hole to run wires through to the ESP32. In the ESP32 chamber, the space was designed with a holder for the ESP32 with a large hole at the top to run the wires out as well. However, this design was flawed, as we ran into several issues: Firstly, the peripherals were taller than their chamber and thus could not fit into the chamber comfortably. Additionally, the space for the wires were too small to fit the peripherals in. We had only realised these flaws after the initial print and worked to quickly design a second prototype. This first prototype confirmed that the holes for the peripherals were aptly designed and could be maintained for the next iteration.

Fig 13. The wire hole that could not fit the peripherals.

3.5.3 Second Design

Learning from the mistakes of the first design, we added one main feature: a rail to slide the top of the peripherals chamber out. Fig 14 in the next page shows the CAD design of the new rail system.

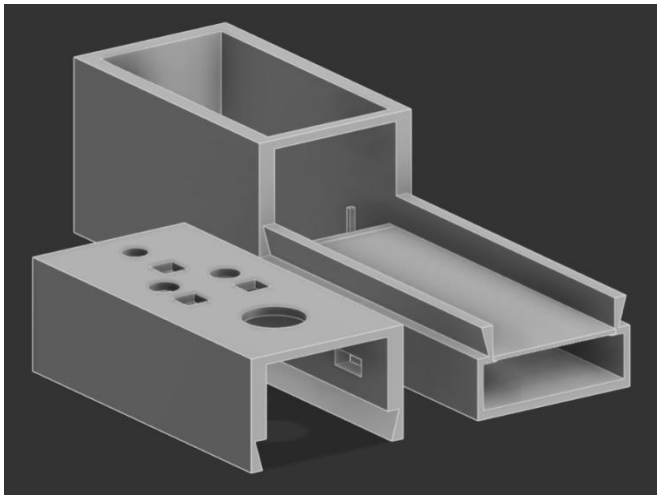


Fig 14. CAD of second iteration with rails

One adjustment made was to increase the height of the Peripheral Chamber, in order to allow for more room for the wires, as well as to account for the height of the round buttons. The larger chamber allows for more space to fit all the devices. This increased the overall size of the remote, but it would not have been possible to fit all the peripherals and wiring in the previous iteration, thus it was a necessary change.

The second dimensional adjustment was to make the holder for the ESP32 much lower in the front (facing the peripherals chamber) and also widening the space for the wires to be passed through. While the previous iteration was technically large enough to fit all the cables, having the extra space allowed me to manoeuvre the wires comfortably within the remote without having to bend the wires awkwardly.

The rail used was a triangle, which was modelled after the dovetail join used in woodworking. The main goal was to allow the remote to open and close easily, in order to access the peripherals chamber. One small oversight was the lack of a proper locking mechanism, which meant that over time, the rail would become smooth, and the top part could slide in and out easily. A temporary measure that we had performed was to use cellophane tape to cleanly seal both the battery chamber and the peripherals chamber, so that the components are securely in place.

3.5.3.1 Design Improvements

One small tweak made in the second design much later was to make the top of the battery chamber removable, in order to wire the battery much more conveniently than having to thread the wire through a small hole at the top of the chamber.

3.6 Libraries used

The ESP32 is programmed using PlatformIO, as it contains the relevant plugins to access the pins and flash the board efficiently.

The MPU6050 has multiple libraries developed for its use, but the one we have decided to use is developed by Electronic Cats. The main advantage of this is to allow us to read the I2C values from a different address than the hardcoded address of 0x68 for the other libraries.

The MAX17043 can be programmed using the SparkFun MAX1704X Fuel Gauge Arduino library. This library offers plug-and-play solutions that made the battery gauge code extremely simple.

The INMP441 reads in data via I2S, which does not require any external library but is included within ESP-IDF.

In order to send data via MQTT, we use PubSubClient by Knolleary. This allows us to send data to the MQTT broker, in both plaintext and binary. Plaintext is used for general commands, and binary is used for voice data.

Sources of the libraries can be found in the References section at the back of this report.

3.7 Sensor algorithms

The main loop of the ESP32 checks if any of the 5 buttons are depressed and sends the corresponding commands and data. Fig 15 below shows the data flows, based on the buttons.

Fig 15. Flowchart of function to data flows

When the button is pressed, the respective packet is sent to the MQTT Broker. For *select* or *delete*, 2 seconds of data is recorded and transmitted in 3 packets to the MQTT Broker. In the ESP32, voice data is recorded as 32-bit integers but is transmitted as 16-bit integers in binary.

For *screenshot*, the command is directly sent to the MQTT Broker with no further data.

For *move* and *rotate*, accelerometer data is recorded and sent to the MQTT broker. However, several checks are made within the ESP32 before the data gets sent. The first packet is "discarded" and used as a calibration packet, and every subsequent packet subtracts the values of this initial packet to cancel out any initial movement before pressing the button. Afterwards, the ESP32 samples at 30ms intervals to figure out which axis is the most likely to be the direction of movement, by taking the direction with the largest acceleration. The most likely direction is defined here as the first direction with 5 largest accelerations. As a result, there will be a delay of at least 180ms, but has been found to be quite negligible in testing.

Debug is activated by pressing *move* and *rotate* at the same time, and it will send the current epoch timestamp to the millisecond to the esp32/command topic, and the last recorded voice data to esp32/voice_data. This is for diagnostics and timing information and should not be used as a command during general use.

Section 4 Communications

4.1 Message Queue Telemetry Transport (MQTT)

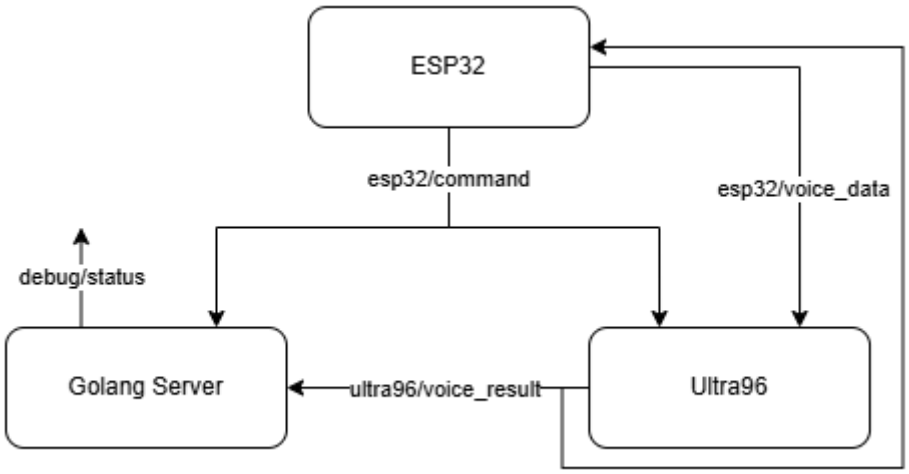


Fig 16. Topics used across nodes

MQTT is a lightweight messaging protocol that is designed for resource-constrained devices and networks and devices with low bandwidth or high latency. Its event-driven architecture is also suitable for our project, which relies heavily on workflows such as a voice data packet triggering a certain action.

The usage of a centralised message broker helps facilitate message passing across services such as the Wi-Fi enabled ESP32 and the Ultra96, as well as the AR device. The use of a publication / subscriber model means that the Ultra96 can publish results, and any subscribers such as the Golang Server and ESP32 are able to receive that message for processing.

The MQTT client is implemented on the ESP32 using the PubSubClient library, while the Ultra96 and Golang server both utilize the Paho MQTT client for communication. The broker is self-hosted using a Mosquitto MQTT broker docker container located on the EC2. All data from client to broker is sent via HTTPS, whereby the packets are distributed to subscribers using the same protocol. The MQTT topics are structured with the following convention *<origin>/<datatype>*, while debug/status is a special topic used for performance testing. The respective data formats sent across the clients can be seen in Section 4.5.

4.1.1 esp32/command Topic

This topic is used for the publishing of commands from the ESP32 for action. The Golang server processes all commands, while the Ultra96 listens only for debug commands. The list of commands includes SELECT, DELETE, MOVE, ROTATE, SCREENSHOT and DEBUG.

4.1.2 esp32/voice_data Topic

This topic is used for the publishing of raw voice data recorded with the INMP441 microphone. The Ultra96 listens on this topic for incoming voice data, and receipt of data is counted as an event.

4.1.3 ultra96/voice_result Topic

This topic is used for sending and receiving response after running inference with the data sent by the microphone. The Golang Server listens on this topic for relaying, while the ESP32 listens to this topic as a form of confirmation that the voice data sent was successfully processed.

4.1.4 debug/status Topic

This topic is mainly used for debugging, as well as latency testing across the entire system. Summary results of the system's latency are published here.

4.2 WebSocket

To enable image storage as a feature, we needed bi-directional communication between the visualiser and the Golang server to allow request-response messaging formats. Using MQTT would require a dedicated topic for the visualiser, while using traditional REST APIs would incur additional overhead costs from opening and closing a HTTP connection. Implementing WebSocket allows us to reap the benefits of both, by allowing request response packets to be sent from either side, while allow constant streaming of the gesture data. Using WebSocket also helps enforce separation of concerns, where the visualiser is only concerned with maintaining a singular connection to a server, while the server oversees the consolidation of information from all topics into a standardised messaging format.

4.2 Amazon Elastic Compute Cloud (EC2)

Amazon EC2 instance is a Virtual Machine service provided by Amazon under Amazon Web Service (AWS). It allows us to provision a lightweight machine on the cloud that we are able to access from a publicly exposed IP address given by AWS Elastic IP. The VM offered is also hosted at a nearby location in Singapore (ap-southeast-1), providing fast and low latency access.

With that public IP address, it allows the ESP32 to connect to the MQTT broker under any network, if the IP address is known. Hosting the MQTT broker there also allows us to bypass the Sunfire SoC VPN, allowing the Ultra96 to send and receive to the broker directly and for us to work with both hardware and inference without local host discovery issues. Certain ports are exposed by editing the instance's inbound and outbound rules to allow MQTT and WebSocket connections.

Within the EC2 instance, there are two services orchestrated by Docker Compose. The services consist of the MQTT Broker and the Golang server.

4.2.1 Golang Server

The Golang server's primary function is to be the middleware between the visualiser and all other devices. The Golang server subscribes to 2 topics, the `esp32/command` and the `ultra96/voice_result` topics, restructuring the simple message formats from the topics into standardised JSON fields that the visualiser can more easily interpret. Because of this malformed packet are ignored, and not passed to the visualiser, acting as a filter as well.

The server's secondary function is to perform synchronisation with the visualiser for any images that need to be in S3. Using AWS SDK, to perform I/O operations with the buckets in

S3, we can send GET, PUT and DELETE requests to S3 using temporarily authorised pre-signed URLs. Due to its nature of being hosted on an AWS resource it is also able to access S3 without requiring extra authentication, hence the server acts as a middleman for S3-related tasks, rather than having the visualiser authenticate with S3 for any actions.

For both uploading and deleting images, the URL is sent to the visualiser in the data field of the packet. For syncing across devices, the server receives a list of all images on the user's device and in the S3 bucket belonging to the user. A comparison is made, where missing images on either side are identified. They required image GETs and PUTs are then sent to the visualiser for synchronisation.

Finally, the server acts as a consolidation point for latency metrics. When triggered by the ESP32, latency between all devices can be calculated, and are published to the `debug/status topic`.

4.3 Amazon Simple Storage Service (S3)

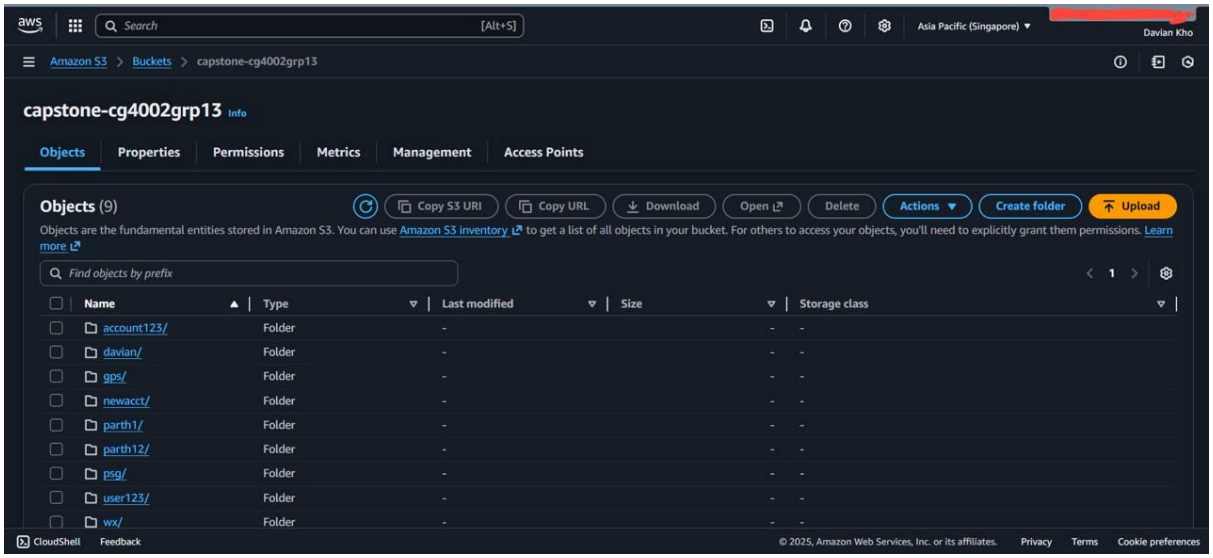


Fig 17. S3 dashboard

The AWS S3 is an online object storage service that was chosen due to its ability to handle and store large number of images. Traditional databases were not chosen as their performance tends to suffer with increasing sizes of data. Images are stored in the format `<username>/<timestamp>.jpg`, allowing the data to be grouped into different sub-buckets where each bucket only contains the images of a specific user. Using the pre-signed endpoints provided by the Golang server, the visualizer can interact with S3 directly without credentials for a limited time. Additionally, it was easier to use and maintain alongside the EC2 instance.

The S3 also doubles as a place of storage for temporary build artifacts generated during deployment (See section 4.4.4). These files are configured to self-destruct after 1 day.

4.4 System Quality Attributes

4.4.1 Encryption

MQTT supports the usage of authentication, which requires the client to enter a username and password before being able to connect to a topic and publish to it. On top of that, MQTT also

supports HTTPS. The WebSocket Server also initiates an upgrade to Secure WebSocket when any client connects. In both situations, mutual TLS (mTLS) is used, and anyone without the correct key-cert pair and root certificate will not be able to connect.

For mTLS to work, a root certificate, server key-cert pair and client key-cert pair is required, where the server and client key-certs are self-signed using the root certificate. Using the OpenSSL command on Linux, a script was created that can take in parameters that allow custom naming of the certificates and specify fields such as Common Name (CN) and Subject Alternative Name (SAN). The SAN and CN only allow connection to localhost and the EC2 instance, ensuring secure communication and identity verification.

4.4.2 State Management

Across the entire system, it is important that the actions on the ESP32 is accurately reflected elsewhere. To ensure atomicity of the actions, we opted for a stateless system, where state only exists on the visualiser. The ESP32 acts as a I/O conduit for the visualiser and Ultra96, while ensuring that actions are singular and locked such that there are no simultaneous actions being sent across. With a stateless system, it becomes more robust and resilient to external factors like intermittent connections and disconnections.

4.4.3 Availability and Consistency

A stateless system also means that data may not always be consistent, as data recorded on the ESP32 may not be immediately reflected on the visualiser. Data communication is asynchronous, leading to eventual consistency instead of immediate consistency. The data is also highly available, as the visualiser stores the last recorded action of the ESP32.

The images stored in S3 can be seen as a replication problem with local devices, where the desired outcome is to have both S3 and local devices with the same images. Captured and deleted images are immediately sent to S3, while changes made outside of the session need to be pulled manually. Since images are created with timestamps, there will not be an issue with conflicting changes between both sides.

4.4.4 Concurrency

When considering concurrency, there are 2 ways to go about doing it. One is through multi-threading; one is through multi-processing (parallelism). There was consideration of using the Ultra96 for multi-processing, since it offers multi-core environment for better parallel execution of threads. Due to the intermittent nature of messages being passed by the MQ, having a service run on the Ultra96 might lead to resource contention, especially since there needs to be a machine learning model running on the Ultra96. Additionally, the broker is hosted elsewhere, and it would make it more convenient if all connections were centrally managed by the EC2 instance.

With the Golang service having to manage data flows from different areas, there is a need for concurrency management. We can leverage Go's in-built goroutines, which allow for concurrency through the Go Runtime. Each routine is managed by the Go Runtime, where they are scheduled to available OS threads for processing. However, the creation of a single Go routine is a lot cheaper than creating an OS thread, allowing for better and more performant management of routines.

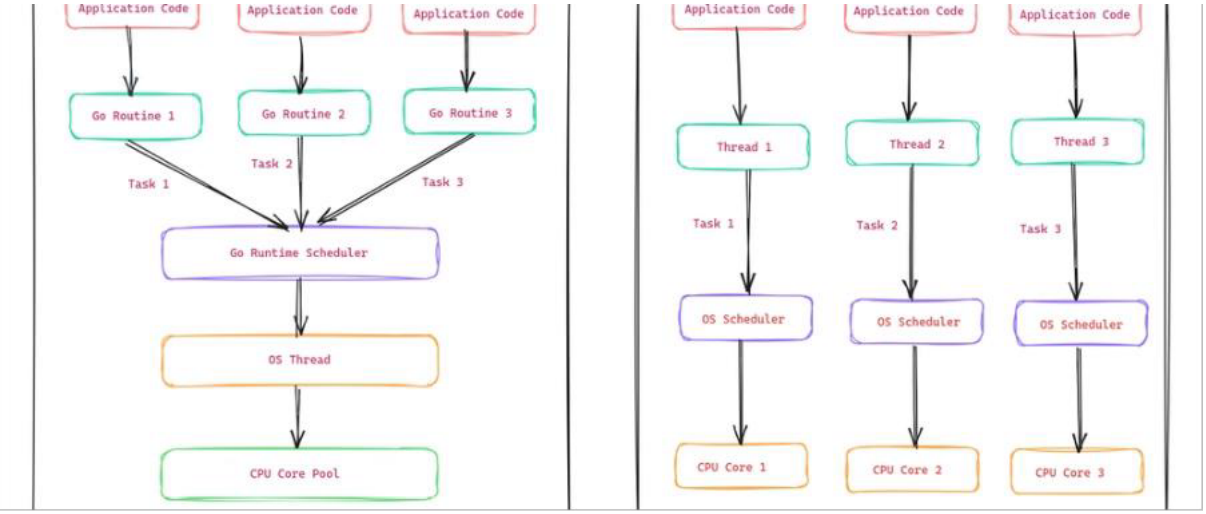


Fig 18. GoRoutines vs traditional threads

4.4.4 Continuous Delivery (CD)

Due to the nature of the server, there is an element of deployment required to the EC2 whenever changes are being made. Two separate versions of the system exist, one for local development and one for deployment. To ensure changes on development are reflected quickly in the EC2, a CD pipeline was created using AWS CodePipeline.

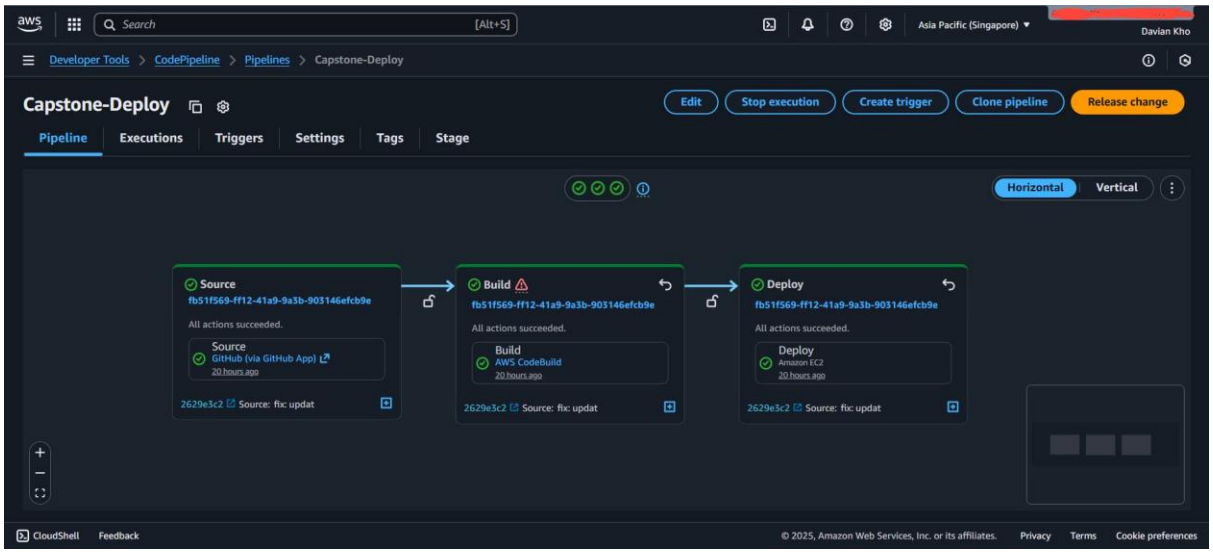


Fig 19. AWS CodePipeline Workflow

The pipeline detects pull requests into our main git branch, which then pulls the code, builds the docker images and stores it in AWS Elastic Container Registry (ECR). Two scripts in EC2 are then executed, one to stop the current running docker compose, the other to pull the new image and restart the services. The code pulled from GitHub is stored temporarily on the S3, to allow different resources from AWS to access the codebase.

4.5 Message Formats

For readability and ease of parsing, all data are sent in raw JSON format. (See appendix for examples) This format is also supported by MQTT, allowing us to send the data more easily.

To support JSON, the data needs to be encoded in string data. Voice data is the exception to this, due to packet size limitations and to make data transfer more efficient. (see section 7)

4.6 Performance

A key aspect with all systems, especially with one that does data streaming across the internet, is how fast the data can travel from point A to point B. However, it is difficult to enable real-time metrics with MQTT on ESP32, so we opted to use timestamps to help track time of packet sending and receiving. With 2 possible actions – gesture and voice, we have 2 paths of data flow we need to test.

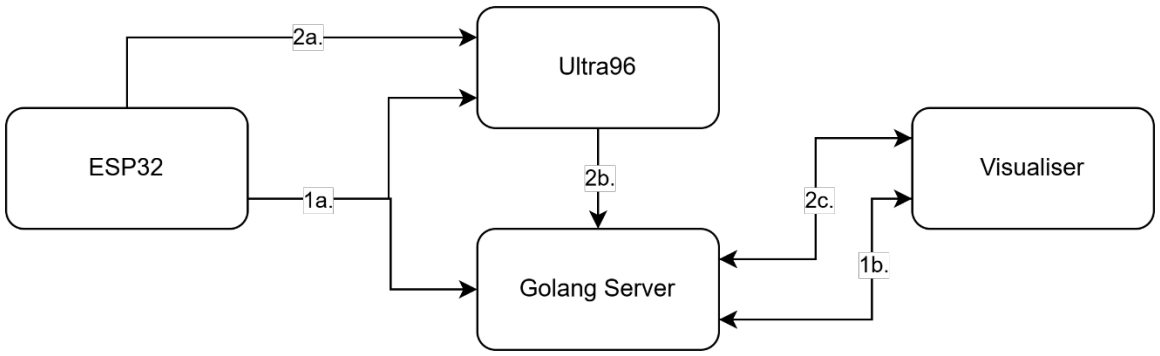


Fig 20. Latency testing workflow

The steps to test latency are as follows:

1. This data flow simulates the entire path for gesture data.
   a. Within the ESP32, upon WiFi connection, a time sync is done with the Network Time Protocol (NTP) server to make sure the local time is updated. The latency measurement is started with a special input from the ESP32. A packet of debug type is sent from the ESP32 to the `esp32/command` topic.
   b. The server sends a gesture ping and receives a gesture pong. The timestamp of the pong is used to calculate the following latency information:
      i) End to End (E2E) latency from ESP32 to visualiser for gesture data
      ii) Server to Visualiser latency
2. This data flow simulates the entire path for voice data.
   a. After the debug command is sent, the ESP32 immediately sends the previous voice_data recorded to the Ultra96. We do not re-record data as we are only concerned here about the time taken to send the voice data packets.
   b. The Ultra96 sends the time it receives the voice data packet, the inference time of the model and the send time of the results to the `Ultra96/voice_result`. The following latency information are calculated or retrieved:
      i) ESP32-Ultra96
      ii) Ultra96-Server
      iii) Inference Time
   c. The server sends a voice ping and receives a voice pong. The timestamp of the pong is used to calculate the following latency information:
      i) End to End (E2E) latency from ESP32 to visualiser for voice data
      ii) Server to Visualiser (if not already initialised)

Both parts *1b.* and *2c.* are executed concurrently.

Step 1a. is primarily responsible for triggering a Goroutine on the server that starts a latency testing session lasting 5 seconds. Latency information is published on the `debug/status` either when all data has been collected or a 5 second timeout has been triggered. Refer to the data packet schema for the format of the published results.

End to End	Time in ms (NUS_STU)	Time in ms (Personal Hotspot)
Voice Recognition	960	568
Gesture Data	117	69

Fig 21. End to End latency with different networks

Node to Node	Time in ms (NUS_STU)	Time in ms (Personal Hotspot)
ESP32 to Ultra96 (Voice Data)	335	177
ESP32 to Golang Server (Gesture Data)	63	60
Ultra96 to Golang Server (INF Result)	7	14
Golang Server to Visualiser	53	8

Fig 22. Node to Node with different networks

The above tables show the latency results of the system. Three samples with the two networks are taken and averaged at different physical locations within NUS SoC MPH.

Section 5 Software/Hardware AI

5.1 Input data and selection of AI model

Before selecting an appropriate AI model for the project, it is necessary to break down and understand the input data that our project is working with. For our project, we are performing audio recognition with our AI, specifically command recognition where the spoken audio will only consist of one word.

However, keeping in mind the limitations of the provided FPGA, we cannot simply just pass the entire waveform into the model as input. The input size will be far too big for the FPGA to handle. Through extensive research, I found that an appropriate method of representing the waveform with a smaller-size array is the Mel Spectrogram transformation. A Mel Spectrogram is similar to the Fourier Transform of an audio signal, except that instead of the frequency scale, we use the Mel scale which more accurately represents how human ears perceive pitch. As can be seen in the graph below, human ears can more accurately perceive differences in lower frequencies than at higher frequencies, and the Mel scale uses this concept to match the human ear's non-linear perception of frequency.

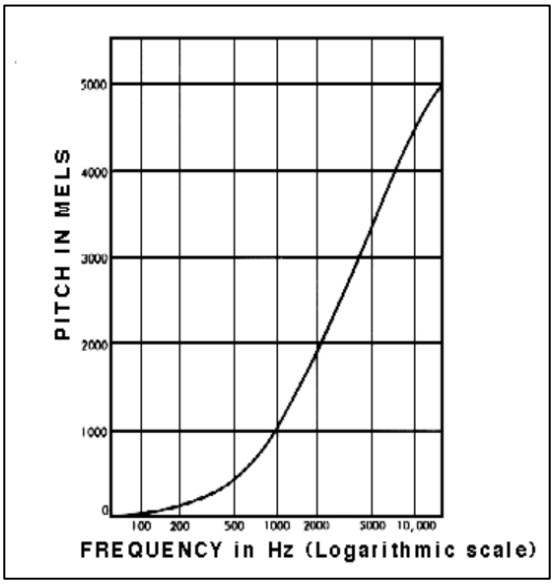


Fig 23. Non-linear perception of frequency

For this project I also further applied decibel scaling to match the loudness perception of human ears. With such a transformation, we not only achieve a better representation of the audio for classification but also manage to reduce the size of the input to 1/3 of the original waveform.

Due to the limited data that we would be able to gather through manually recording audio samples, I concluded that we would not be able to train an entire model from scratch, regardless of what model architecture I decided to use. Hence, I looked towards taking advantage of a popular Machine Learning (ML) technique, fine-tuning instead. I first had to scour the web to look for an appropriate dataset that I could pre-train my model on before performing fine-tuning with our recorded audio data, and fortunately I manged to find an appropriate dataset that closely matched our use-case. This was the Google Speech Commands dataset, which was a massive dataset consisting of over 100k audio samples across 35 different commands. The samples were 1 second audio clips sampled at 16kHz, which would be converted to a Mel Spectrogram before being fed as input into the model for training.

Next came the model selection. This was truly a difficult choice as there were already many pre-trained classification models available open source, but I quickly discovered that most of them would still be way too large to implement in the FPGA. Even the "mini" versions of huge ML models would still take hours to synthesize into Verilog on the Vitis IDE, which was just unrealistic for implementation. Hence, I decided to just come up with my own architecture, although with reference to other popular architectures that I knew already worked well. My model architecture took heavy inspiration from ResNet-18, which although is traditionally an image classification CNN model, I believed that it would work well as a Mel Spectrogram is similar to an image where analysing a group of neighbours rather than an individual datapoint is more useful in identifying the changes in pitch and loudness that is present in audio.

The final model architecture is as such:

```
self.network = nn.Sequential(
        nn.Conv2d(1, 16, kernel_size=(3,3), padding=1, stride=2),
        nn.ReLU(),
        nn.Conv2d(16, 16, kernel_size=(3,3), padding=1, stride=1),
        nn.ReLU(),
        nn.Dropout(0.2),

        nn.Conv2d(16, 32, kernel_size=(3,3), padding=1, stride=2),
        nn.ReLU(),
        nn.Conv2d(32, 32, kernel_size=(3,3), padding=1, stride=1),
        nn.ReLU(),
        nn.Dropout(0.2),

        nn.Conv2d(32, 64, kernel_size=(3,3), padding=1, stride=2),
        nn.ReLU(),
        nn.Conv2d(64, 64, kernel_size=(3,3), padding=1, stride=1),
        nn.ReLU(),
        nn.Dropout(0.2),
        nn.AdaptiveAvgPool2d((1,1))
)

self.classifier = nn.Sequential(
        nn.Linear(64, n_classes)
)
```

Fig 24. CNN architecture

The network itself consists of three separate "blocks", with each "block" having similar architecture to the original ResNet-18. Dropout was implemented between each block to prevent the network from overfitting, which is particularly important here as the network weights are only going to be adjusted during pre-training. The last step in the network is an adaptive average pooling that outputs a single value which is then fed into the classifier for classification.

The commands that we used for the audio recognition can be found here.

5.2 Training and testing of AI model

There are multiple phases associated with the training and testing of the model which will be further expanded on below:

5.2.1 Model pre-training

I am using Google Speech Command's dataset to perform pre-training on the model. Google had already split the dataset into training and testing portions to make it simple for the user, so it was not difficult to train and evaluate the model. The decrease in test error started plateauing around the 30-epoch range, which was quite normal for the size of the dataset used, so I decided to stick to 30 epochs for the model's pre-training. For the training itself, the Adam optimizer was used with argmax classification. The results here were rather unexpected (in a good way), as my model performed better than the original ResNet-18 in classification accuracy on the test dataset. It boasted around 94.5% accuracy on average across multiple training sessions, which was 0.5% better than ResNet-18 on average.

However, the classification accuracy here is admittedly not that important as this was just the pre-training phase, so I was only targeting around 93-94% classification accuracy to avoid overfitting the model to Google's dataset. For sanity check purposes, I also tried other architectures such as different CNNs, and even Multi-Layer Perceptrons, with poor results. Nevertheless, with my target accuracy reached, I could then move on to the next phase – fine-tuning.

5.2.2 Gathering of audio samples for fine-tuning

Before fine-tuning, we had to record audio samples for our actual vocabulary. All in all, there are about 20 or so unique people that we recorded. However, some of the audio was of rather poor quality (bad environmental noise, speaker too soft etc.), so I had to drop some of them from the dataset. As such, I also decided to add some AI-generated audio samples to enhance the diversity of the dataset. About 5 or so unique AI speakers were used in our dataset. While recording, we also took note of the speaker's gender, accents, and intonation in speech to ensure diversity in our audio dataset. However, we felt that recording with just a 1 second window was too small as a lot of the times the audio just gets cut halfway through before the person could even finish speaking, so we opted for a 2 second window instead. The microphone was recording at 8kHz, which was half of what was used in the Google Speech Commands dataset.

The recorded audio was also split into training and testing datasets. This was done by randomly selecting a few unique voices' samples for the testing dataset.

5.2.3 Model fine-tuning

For the fine-tuning phase, I first replaced the final linear classifier layer to match the number of commands that we were using. This new linear classifier's weights were initialized with Xavier Initialization to stabilize gradients during the fine-tuning. Afterwards, every layer except the last layer was frozen to ensure that only the last layer's weights were updated during fine-tuning.

As mentioned, our recorded samples were 8kHz which was half of the 16kHz that was used in pre-training, but since we used a 2 second window instead of 1 second, the total number of samples in the input remained the same (8000*2 = 16000*1), so I could just feed our recorded audio as input into the model directly. Initially I attempted to up sample the recorded samples to 16kHz and extract a 1 second clip from it to maintain the 16kHz from pre-training, but I

found that resampling the audio introduces artificial datapoints into the sample which is not good for classification purposes.

At this stage I also had to perform some hyperparameter tuning to ensure the final layer does not get underfitted/overfitted. I found that I had to increase the learning rate by quite a bit to ensure that the last linear layer could learn meaningful weights. I also had to increase the number of epochs as the fine-tuning dataset was much smaller (only ~300 samples in total) in comparison to the Google Speech Commands dataset.

I also tried other techniques which garnered little improvement, if any, to the final model. One of them was fine-tuning the entire model instead of just the final layer, which seemed to be too much for the model to learn. I also tried learning rate scheduling which is a technique to adjust the learning rate dynamically over the training process instead of keeping it fixed, but I could not utilize it properly due to my inexperience with it, hence I just dropped it and kept to a fixed learning rate.

5.2.4 Model evaluation

For model evaluation, I simply compared the average test accuracy across multiple fine-tuning attempts. As mentioned above, I employed different techniques to improve model performance and chose the best performing model for deployment. I purposefully selected some lower quality audio for the test dataset to enhance the model generalization, so I was satisfied with >80% test accuracy for actual deployment.

5.3 Realising the AI model on the Ultra96 and FPGA

The model training was all done on Python for ease of use, with the help of libraries like PyTorch and TorchAudio. After the training is complete, the weights are then extracted to be used in a Vitis project for Register Transfer Level (RTL) synthesis.

In the Vitis project, we have 3 files to take note of: the weights header file that we extracted from Python, the reimplementation of the CNN model in C++, and a testbench file to crosscheck the correctness of implementation. By running the “C Simulation” function in the Vitis IDE, we can check the correctness of the CNN implementation using the testbench file. After verifying the implementation, “C Synthesis” is performed to convert the C++ implementation of the model into Verilog. Finally, we can export the new implementation as an Intellectual Property (IP) block by running the “Package” function on the Vitis IDE.

On Vivado, we then create a block diagram design which consists of the Zynq UltraScale+ CPU block, the DMA block, as well as the custom IP block that we just created in Vitis. By wiring the CPU block, DMA block, and the custom IP block correctly, we can setup the design to run the AI model inference on the FPGA. Afterward, we can then generate a bitstream which can be uploaded to the CPU on the Ultra96. On the Ultra96 itself, the Python Productivity for Zynq (pynq) library is used to upload the bitstream and run the model on the FPGA. The pynq library abstracts away all the lower-level implementation and DMA setup so we can much more easily transfer data to and from the FPGA via DMA.

Attached is the block diagram that I created in Vivado:

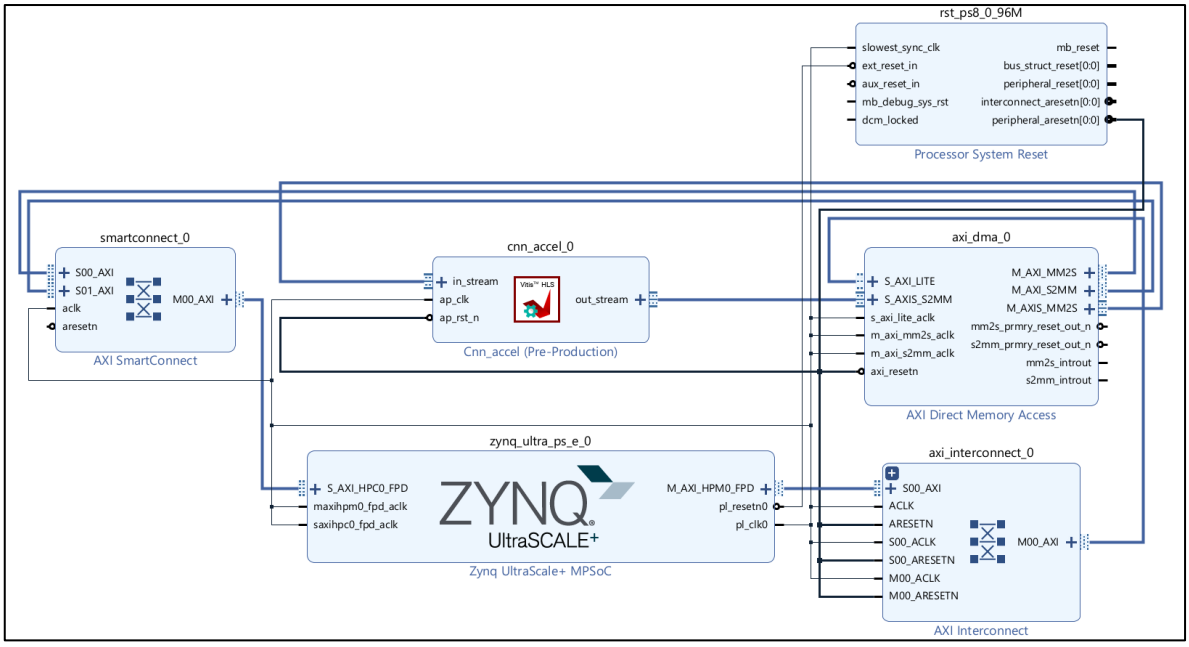


Fig 25. Vivado block diagram

For deployment, a script is constantly running to receive audio signals from the microphone and perform audio classification on it. The microphone records audio at 8kHz over a 2 second period, then sends the waveform to the Ultra96 via MQTT. The audio signal is then pre-processed, and audio classification is performed on it. Finally, the classification result is sent out via MQTT.

5.4 MQTT Limitations

Due to MQTT limitations, we are unable to send the entire audio wave in one transmission, so we decided to split it into three overlapping sections instead. For a 2 second audio recording, the waveform is split into 0s – 1s, 0.5s – 1.5s, and 1s – 2s respectively. They are then sent in succession to MQTT. There are three cases to consider here:

- Ultra96 receives all three packets successfully, so the 0s – 1s and 1s – 2s portions are merged back together to regenerate the original signal which is then sent to the AI model for inference.
- Ultra96 receives only first two packets, so only the 0.5s – 1.5s packet is sent to the AI model for inference. Even though this audio wave is only 1s in length, it is usually sufficient to capture the entire spoken audio as human reaction time has already been factored in. As the audio wave is only 1s which is shorter than the 2s that the AI model is expecting as input, I pad this audio wave on the left and right ends with zeroes.
- Ultra96 receives only one packet, the 0s – 1s portion. Factoring in human reaction time, this portion of the audio is usually unable to capture the entire spoken command, meaning that I cannot use it for audio recognition. Hence, there will be an error notification informing that the inference failed.

Section 6 Software Visualizer and Game engine

6.1 Visualizer Software Architecture

6.1.1 Software Framework and Libraries

We've built this project on a hybrid architecture that combines the Unity platform with custom native iOS frameworks to achieve its advanced functionality. The application's foundation is the Unity 2022.3.62f1 engine, which serves as the primary development environment for C# scripting, asset management, and UI.

For its core augmented reality features, our project utilizes AR Foundation as a high-level, cross-platform interface. On our target iOS platform, this directly leverages Apple's native ARKit framework to handle all world-tracking, ray casting, and camera management required for the "Object Placement" stage.

A secondary AR framework, the Niantic Lightship SDK, is integrated to power the "Object Detection Mode". This specialized SDK provides the machine learning models necessary for real-time, on-device object recognition.

Finally, to overcome critical limitations in standard C# libraries, native Objective-C framework from iOS is utilized. This native layer provides two essential functions not available in Unity: a high-security mTLS WebSocket client for server communication and a bridge to Apple's Photos framework for saving screenshots directly to the user's iOS gallery.

6.1.2 Modules Utilized

6.1.2.1 Core AR & State Management

- **`StageManager.cs`:** A high-level state machine that manages the application's three primary modes: `ObjectPlacement`, `ObjectDetection`, and `Settings`. It controls the visibility of all UI and AR components.
- **`ObjectManager.cs`**: Manages the complete lifecycle of all AR objects, including their instantiation, selection, deletion, and remote command handling.
- **`ControllableObject.cs`:** An individual component attached to AR objects that translates joystick and remote `VirtualJoystickState` data into physical movement and rotation.

6.1.2.2 Niantic Object Detection

- **`ObjectDetectionSample.cs`:** The primary logic script for the detection mode. It subscribes to Niantic's `ARObjectDetectionManager` and uses the `DrawRect` utility to render bounding boxes and labels for detected objects.

6.1.2.3 Native Networking & Wrappers

- **`WebSocketMTLS.mm`:** A custom Objective-C native module that implements a secure WebSocket client. It is uniquely capable of loading a P12 client certificate to perform mutual-TLS (mTLS) authentication.

- **`WS_Client.cs`:** The C# "wrapper" script that uses `[DllImport("__Internal")]` to create, connect, and send messages through the native `WebSocketMTLS` module.
- **`CommandHandler.cs`:** Parses all incoming JSON messages from the WebSocket and routes commands (e.g., `COMMAND_SELECT`, `S3_SYNC_RESPONSE`) to the appropriate manager.

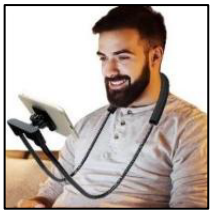
6.1.2.4 Cloud-Synced Gallery

- **`ScreenshotManagerIOS.cs`:** Manages screenshot capture, local file saving (to a user-specific folder), and calls the native gallery plugin. It also initiates the S3 upload process.
- **`iOSScreenshotPlugin.mm`:** The native Objective-C module that uses Apple's `PHPhotoLibrary` to save an image file to the iOS Photos app.
- **`GalleryViewController.cs`:** The UI controller that loads all local screenshot thumbnails and manages the gallery state. It initiates sync, delete, and download operations.
- **`ScreenshotSyncManager.cs`:** Implements the full bidirectional sync logic. It sends a list of local files to the server and processes the response of GET (download) and PUT (upload) presigned URLs.
- **`ScreenshotUploadManager.cs`:** Handles the process of uploading a single file to S3 using a presigned PUT URL.
- **`ScreenshotDeleteManager.cs`:** Manages the two-step deletion process: acquiring a presigned DELETE URL, executing the S3 delete, and then deleting the local file.
- **`ScreenshotDownloadManager.cs`:** Manages the user-initiated action of saving a screenshot from the in-app gallery to the iOS Photos app.

6.1.2.5 Architectural & Utility Modules

- **`UnityMainThreadDispatcher.cs`:** A critical thread-safety module. It implements a message queue (using a `lock` [**mutex**]) to allow the background network thread to safely pass work to Unity's main thread, preventing application crashes.
- **`iOSBuildPostProcessor.cs`:** A Unity Editor script that automates the iOS build process. It modifies the exported Xcode project to link the `Photos.framework` and add the necessary `Info.plist` permissions.
- **`Newtonsoft.Json`:** A third-party C# library used by the `CommandHandler`, `ScreenshotUploadManager`, `ScreenshotDownloadManager`, `ScreenshotDeleteManager`, `ScreenshotSyncManager` for all JSON parsing and deserialization.

6.1.3 Phone Placement



We used a mobile phone neck stand for our project as it provides a practical solution for hands-free interaction during the furniture layout process. By holding the phone in a stable and consistent position, the stand will allow the user to perform the gestures and interact with the hardware without manually handling the device.

Another key consideration when deciding to use this mobile phone neck stand was that this setup ensured a more consistent field of view for the AR camera, and maintaining the alignment of the virtual objects placed.

6.2 Visualizer Feature Specifications

The main purpose of the software visualizer is to provide an intuitive and reliable AR interface which allows users to place and manipulate virtual furniture in a physical environment. Using the different gestures and buttons, we will be able to navigate through the different features implemented in the application. We will also be employing different game states.

6.2.1 Object Placement

6.2.1.1 Relevant Images

Click HERE to be directed to the relevant images.

6.2.1.2 Data Displayed

Data Component	Description
Connection Status	Shows whether we are connected to the server
Current Stage	Displays “Object Placement Mode”
Current Object	Shows user the current selected object
Object Coordinates	Shows the object coordinates in (x, y, z) form

6.2.1.3 Joystick + Button Functionalities

Component	Description
Left Joystick*	Movements in the X and Y directions
Up Button*	Movement in the positive z direction
Down Button*	Movement in the negative z direction
Right Joystick*	Movement in the pitch and roll directions
Left Button*	Movement in the clockwise yaw direction
Right Button*	Movement in the anti-clockwise yaw direction
Object Detection Button	Go into Object Detection Mode
Screenshot Button	Take screenshot to save to app gallery
Connection Button	**Toggle switch #1** to connect/disconnect to the server
Settings Gear Button	Go into settings
Furniture Buttons	To select/instantiate^ the corresponding button
Delete Button	To delete the selected object

*These buttons become **inactive** once we connect to the server

^Instantiation occurs if the object was not already present in the scene

6.2.1.4 Detailed Explanation

A core design choice in the application's "Object Placement" stage was to simplify the scene and user controls by **only allowing one instantiation of each object type**. This logic is enforced by the `ObjectManager`, which maintains a dictionary of all active objects in the scene. When the user taps a furniture button in the `SelectionPanelManager`, the `ObjectManager`'s `HandleObjectButton` method is called. This method first checks if an object with that name (e.g., "Table") already exists in its dictionary. If it does, the existing object is simply re-selected; a new one is not created. A new object is only instantiated if it's the first time that specific button has been pressed.

To manage the object's physical state, the application uses a `Rigidbody` component that switches between two modes. When an object is first instantiated or when it is **deselected**, its `Rigidbody` is set to be **kinematic** (`isKinematic = true`) and to ignore gravity (`useGravity = false`). In this kinematic state, the object is effectively "physicsless"—it will not fall or be pushed by other physics forces, allowing it to remain floating exactly where the user places it in the AR space.

Conversely, when an object is **selected** by the user, its `Select()` method is called, which immediately sets its `Rigidbody` to be **non-kinematic** (`isKinematic = false`). This "wakes up" the component and engages the physics engine. All object movement from the joysticks is then applied using `_rigidbody.MovePosition()` and `_rigidbody.MoveRotation()`. This is a deliberate decision, as it delegates all **collision physics** directly to the Unity game engine. By using these physics-based methods, the object will realistically collide with and be blocked by other objects, rather than passing through them. Once the user is finished moving the object and deselects it, the `Rigidbody` is made kinematic again, "freezing" it in its new position.

6.2.2 Object Detection Model

6.2.2.1 Relevant Images

Click HERE to be directed to the relevant images.

6.2.2.2 Data Displayed

Data Component	Description
Rectangle Box	Box representing the object detected
Number in/near the box	Probability associated with the object detected

6.2.2.3 Detailed Explanation

A key architectural aspect of the Object Detection Model is that its functionality is entirely self-contained and operates independently of any server connection. Unlike the Object Placement and Gallery modes, which rely heavily on the `WS_Client` for multi-device remote commands and cloud-based S3 synchronization, the detection feature is processed 100% locally on the iOS device.

This mode is powered by the Niantic Lightship SDK, which performs all machine learning analysis on the device's hardware. The `ObjectDetectionSample` script subscribes to Niantic's

local `ARObjectDetectionManager` to receive detection results from the live camera feed. Because this entire process occurs on-device, the user does not need to be connected to the server to use this feature. The `StageManager` logic reinforces this, as switching to the `ObjectDetection` stage primarily involves activating the local Niantic SDK, a process that has no dependency on the network state.

6.2.3 Settings Page

6.2.3.1 Relevant Images

Click HERE to be directed to the relevant images.

6.2.3.2 Data + Button Functionalities

Data	Input Field	Description
Username*	Text Box	Username of user used to connect to server
Hide Controls	Button^	Visual display if the movement controls are hidden
Discrete Packet Debugging	Button	A single packet debugging mode used by our group for sending discrete packets
Sensitivity	Slider	Slide to increase/decrease sensitivity
Cross	Button	To close the settings page and return to Object Detection/Placement Stage

* without the username it is not possible to connect to the server
^ Pressing this button has no effect as it is checks and is controlled by connection to the server

6.2.3.3 Detailed Explanation

The Settings Page serves as the central configuration hub for the application, managed by the `SettingsPanelController` and `SettingsMenuController`. Its functionalities are critical as they directly influence networking, local data storage, and remote control responsiveness. All settings persist locally on the device using `PlayerPrefs`.

The **Username** field is the most important setting in the application. This value is used as the primary key for data isolation across the entire system. Without the username, a connection with the server cannot be established.

1. **Networking:** The `WS_Client` retrieves this username to dynamically construct the `userId` (e.g., `username-timestamp`) and `sessionId` (e.g., `username`) required to establish its mTLS WebSocket connection.
2. **Local Storage:** The `ScreenshotManagerIOS` uses the username to define the local save directory (e.g., `Application.persistentDataPath/screenshots/{username}/`), ensuring each user's screenshots are stored separately.
3. **Cloud Storage:** The `ScreenshotUploadManager`, `ScreenshotSyncManager` and `ScreenshotDeleteManager` use the username to construct the S3 object key (e.g., `{username}/{filename}.jpg`), linking the local files to their corresponding cloud-based counterparts.

The **Data Mode** toggle allows the user to switch the remote control logic between "Streaming" and "Discrete" packets. This setting directly modifies the `dataMode` enum in the `VirtualJoystickState` of all `ControllableObject`s. "Streaming" mode uses continuous `Mathf.Lerp` for smooth damping, while "Discrete" mode uses a time-based impulse decay system. This provides a way to test during debugging.

The **Controls Sensitivity** slider adjusts a multiplier (from 0.1x to 2.0x) that is also applied to the `VirtualJoystickState` of all objects. This multiplier directly scales the incoming accelerometer and gyroscope data in `Streaming` mode, allowing the user to tune how responsive the remote controls feel.

Finally, the **"Show Controls Indicator"** is a read-only UI element, not a user-configurable toggle. Its state ("ON" or "OFF") is set automatically by the `WS_Client` upon a successful connection or disconnection. This serves as a clear visual confirmation for the user, indicating whether the app is in "remote" mode (controls hidden) or "local" mode (controls visible). The settings panel also acts as the main navigation hub, containing the tabs to access the **Debug Viewer** and the **Screenshot Gallery**.

6.2.3.4 Tabs Structure

Tab Button	Description
Settings Tab (Default)	Go to the main settings Page
Debug Output Tab	Go to the Debug Output Page
Photo Gallery Tab	Go to photo gallery, sort images for user and check for sync (only when connected to server)

6.2.4 Debug Output Page

6.2.4.1 Relevant Images

Click HERE to be directed to the relevant images.

6.2.4.2 Data Displayed

Data Component	Description
Debug Output	Prints out debug logs from the relevant sections

6.2.4.3 Button Functionalities

Component	Description
Connect Button	**Switch #2** to connect to the server
Disconnect Button	**Switch #2** to disconnect from the server
Clear logs button	Clears all the logs (as shown here)
Send ping	Debug packet for earlier testing
Cross Button	To close the settings page and return to Object Detection/Placement Stage

6.2.4.4 Detailed Explanations

The Debug Output Page is a non-intrusive, in-app console that serves as the primary method for debugging the application, especially when deployed on a physical device. It is managed

by DebugViewController. Its core function is to provide a real-time, **scrollable view** of timestamped log messages generated from all major systems in the app, such as the `WS_Client`, `CommandHandler`, and various `ScreenshotManager` scripts.

To manage performance and memory, the debug view is **limited to a maximum of 100 lines**. This limit is implemented using a **Queue data structure** (`Queue<GameObject>`). When a new log message is added via the static `AddDebugMessage` function, a new UI text prefab is instantiated and added to the scroll view, and a reference to this `GameObject` is `Enqueue`d into the `logGameObjects` queue.

Simultaneously, the function checks if the queue's count now exceeds the `maxLogLines` limit. If it does, the `Dequeue()` method is called to retrieve the oldest `GameObject` (the first item added to the queue), which is then destroyed from the scene using `Destroy(oldestLog)`. This creates a "first-in, first-out" (FIFO) system, ensuring the log view always shows the 100 most recent messages without consuming unbounded memory.

6.2.5 Photo Gallery Page

6.2.5.1 Relevant Images

Click HERE to be directed to the relevant images.

6.2.5.2 Data Displayed

Data Component	Description
Connected to server Text	Visual reference to check connection status
Sync Required Text	Only enabled when we are connected to server and there are uploads and downloads required

6.2.5.3 Button Functionalities

Component	Description
Clickable Images	Click to open the image viewer
Cross Button	To close the settings page and return to Object Detection/Placement Stage

6.2.5.4 Detailed Explanations

The Photo Gallery Page serves as the user's persistent, cloud-synced library of all captured screenshots. This module is managed by the GalleryViewController.

Upon opening the tab, the `OnTabOpened` method triggers a `RefreshGallery` call. This process is **disk-based**, not session-based. It retrieves the current **username** from the `SettingsMenuController` and locates the corresponding local storage folder (e.g., `screenshots/{username}/`). It then reads all `.jpg` files from this directory, instantiates a `thumbnailPrefab` for each one, and loads the image data from disk into a `Texture2D` to populate the scrollable grid. All loaded thumbnail textures are cached in a list for proper memory management and are destroyed when the gallery is closed or refreshed.

A core feature of this page is its **event-driven, non-polling design**. The "Connected to Server" status text, for example, is updated by subscribing directly to the `WS_Client.OnConnectionStatusChanged` event. Furthermore, the page performs a **deferred sync check**; after the tab opens, it waits two seconds and checks for sufficient memory before calling `CheckSyncRequired`. This function uses the `ScreenshotSyncManager` to determine how many files are pending upload or download, updating the "Sync Required" text. The "Sync" button itself calls `ScreenshotSyncManager.Instance.PerformSync`, which initiates the full bidirectional synchronization with the AWS S3.

6.2.6 Image View Page

6.2.6.1 Relevant Images

Click HERE to be directed to the relevant images.

6.2.6.2 Data Displayed

Data Component	Description
Metadata	Information about the filename, time & date image was taken

6.2.6.3 Button Functionalities

Component	Description
Download Button	To download an image to iPhone gallery
Delete Button	To delete an image from server and/or locally
Close Button	To close the image viewer and return to Photo Gallery

6.2.6.4 Detailed Explanations

The "Image View Page" is the **full-screen overlay** (`fullScreenOverlay`) managed within the `GalleryViewController`. It is not a separate scene but rather a `GameObject` that is activated when a user taps any thumbnail in the gallery grid.

When a thumbnail is clicked, the `ShowFullScreenImage` method is executed. This method loads the **full-resolution** JPG from its file path into a new `Texture2D`, which is then displayed in the `fullScreenRawImage` component. This texture is held in memory (`currentDisplayedTexture`) only while the overlay is active. The method also parses the image's filename (which is a Unix epoch timestamp) to display a human-readable date and time, along with the filename itself, in the `timestampText` UI element.

This view provides two critical functionalities:

1. **Delete:** The "Delete" button calls `DeleteCurrentScreenshot`, which in turn triggers the `ScreenshotDeleteManager`. This manager requests a presigned DELETE URL from the server, deletes the file from S3, and upon success, deletes the local file. The `OnDeleteComplete` callback then hides the overlay and refreshes the gallery grid.
2. **Download:** The "Download Image" button calls `DownloadCurrentScreenshot`. This activates the `ScreenshotDownloadManager`, which uses `ScreenshotManagerIOS.Instance.SaveToIOSPhotos` to pass the local file path to the native Objective-C plugin (`iOSScreenshotPlugin.mm`), saving the image directly to the user's main iOS Photos library.

When the user closes the overlay, the `HideFullScreenImage` method is called, which immediately **destroys the full-resolution texture** (`Destroy(currentDisplayedTexture)`) to free up memory.

6.2.7 Parsing Commands

Refer to the websocket messages HERE for your reference.

All incoming WebSocket messages are parsed and routed by the `CommandHandler`, which acts as the central hub for all network instructions. When a message is received from the `WS_Client`, it is immediately passed to the `CommandHandler.HandleWebSocketMessage` method. This function first uses the **Newtonsoft.Json** library to parse the raw JSON string into a `JObject`. It then reads the top-level `eventType` string to identify the message's purpose. A large `switch` statement uses this `eventType` to route the `JObject` to the correct internal function, such as `HandleSelectCommand` for a `COMMAND_SELECT` message or `HandleS3SyncResponse` for an `S3_SYNC_RESPONSE`.

Once routed to the specific function, a second, more detailed parse occurs to extract the payload from the `data` field of the `JObject`. For object-based commands like `COMMAND_SELECT`, it retrieves the object name string from `data["result"]`. For array-based commands like `COMMAND_MOVE`, it deserializes the `data` field into a `JArray` to extract the X, Y, and Z float values. This two-step parsing process allows the `CommandHandler` to efficiently translate generic JSON text into strongly-typed, specific instructions that are then passed to other managers, like the `ObjectManager` or `ScreenshotSyncManager`, to be executed.

6.2.8 Smooth movements of objects (when connected to server)

Smooth movement is achieved using a **damping and interpolation** system managed by the `VirtualJoystickState` component. When a remote command like `COMMAND_MOVE` arrives, the `CommandHandler` passes the raw data to this script, which stores it as a **target value** (e.g., `targetAxialHorizontal`). This `target` represents the *desired* speed, not the object's actual current speed.

This system prevents jerky motion by separating the "target" from the "current" speed. In its `Update()` loop, the `VirtualJoystickState` script uses `Mathf.Lerp` (Linear Interpolation) to gradually move its `current` speed towards the `target` speed over several frames. The `ControllableObject` then reads this constantly updating **`current` (smoothed) value** to apply movement to its `Rigidbody`. This interpolation ensures the object smoothly accelerates and decelerates instead of instantly snapping to new positions, even if the incoming network data is sporadic.

6.3 Sample flow of the run of the application

6.3.1 When not connected to server

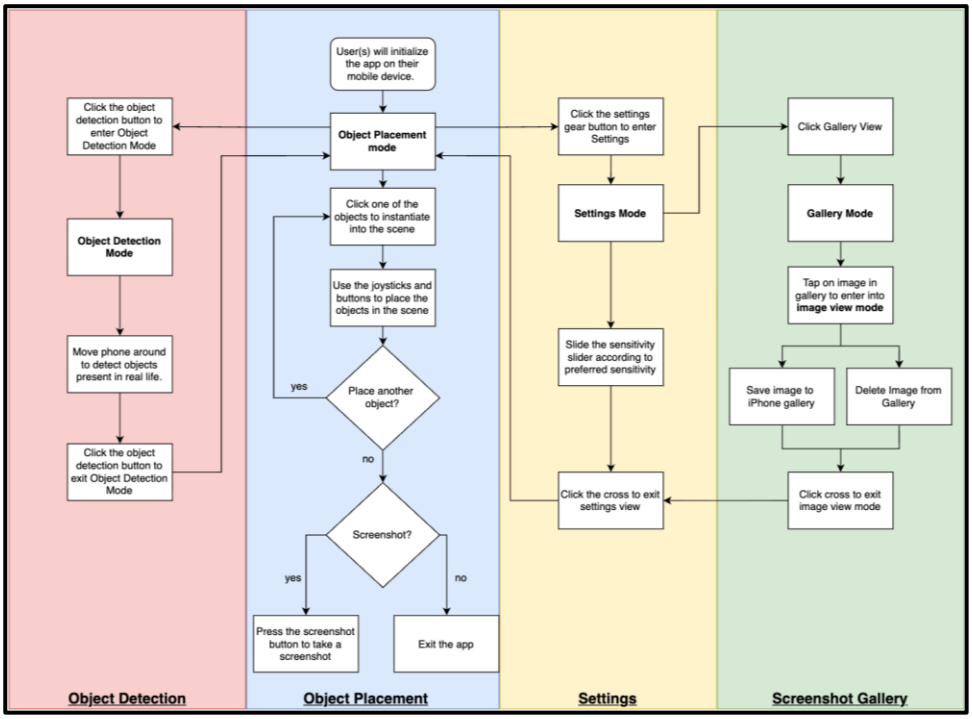


Fig 26. Sample User Flow (Not connected to server)

6.3.2 When connected to server

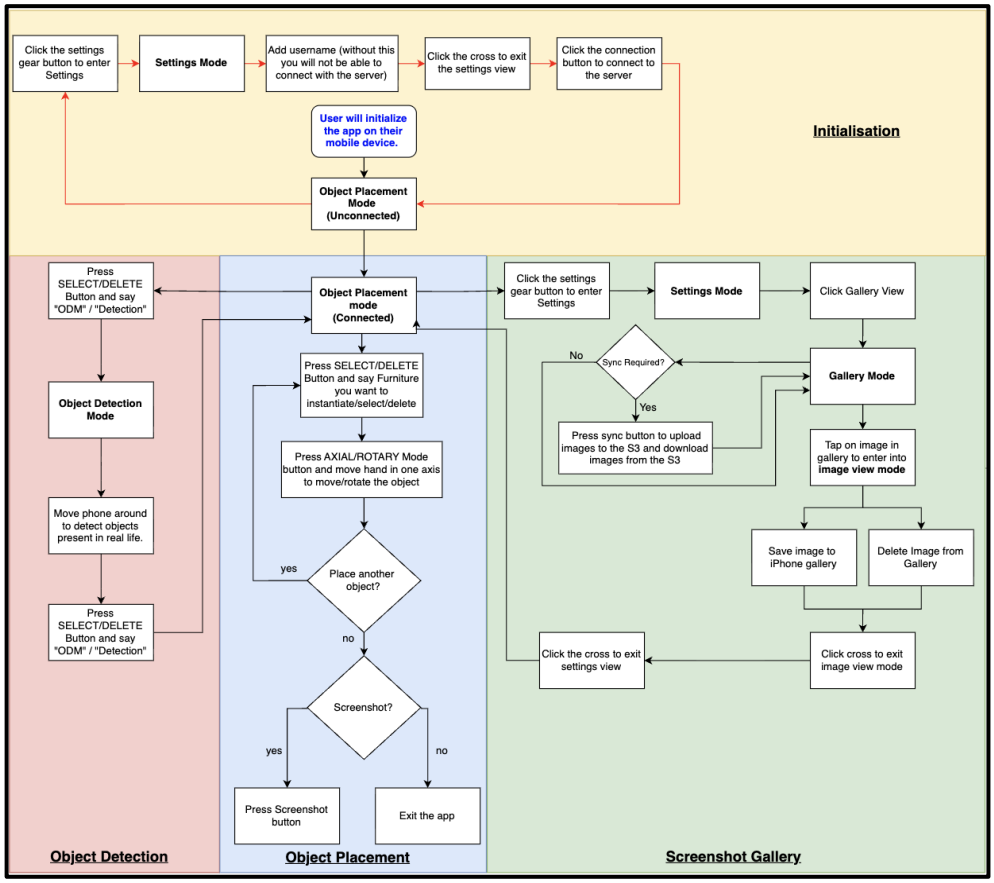


Fig 27. Sample User Flow (Connected to server)

Section 7 Issues Faced

7.1 Hardware

7.1.1 3D Print issue

The main issue with the 3D print was size concerns, as the goal was to minimize the size of the device. One big factor working against this principle was the PCBs that the peripherals already came with, such as the pushbuttons and the RGB LED. The workaround for this was to solder the wires at right angles when necessary, so that we could place the PCB sideways, while allowing the wires to run towards the back where the ESP32 was. The second big workaround used was to stack the peripherals vertically and staggered, such that they could still be accessed from the top, but the PCBs which did not need to be accessed could overlap with one another. Figs XX and YY below illustrate the 2 workarounds we used to fit all the peripherals within such a small area.

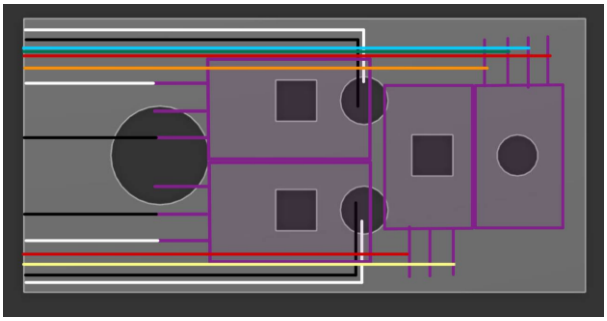


Fig 28. Top Section of CAD with drawing of wire layout

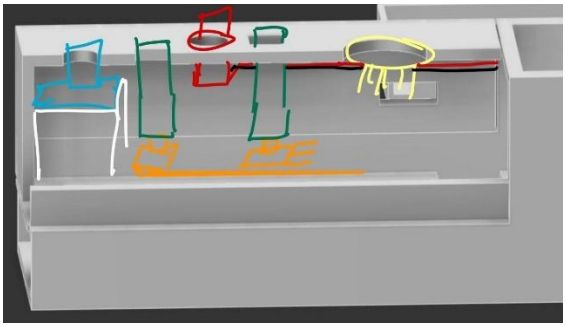


Fig 29. Cross Section of CAD with peripherals layout

7.1.2 Wiring issues

The next issue I faced was the compact wiring of the ESP32. With the large number of pins used, wiring was a large issue. In order to have a strong contact, a reasonable amount of solder was required. At the same time, it was difficult to solder many consecutive pins without some form of system. The solution I used was to only wrap the wire once around the pin, and solder extremely carefully, trimming any excess wire after soldering before moving onto the next joint. The second part of the solution was also to stagger consecutive pin solders vertically, so that each solder ball has space between one another, preventing unwanted short circuits.

7.1.3 MPU6050 Drift

Due to the lack of a magnetometer within the MPU6050, one main issue was the drift in the accelerometer readings. To counteract this, we programmatically removed noise by using the first reading as “calibration” readings, using those values as our starting values. Then, we took the maximum value as the axis of choice, where the first axis to reach 5 largest values is considered the chosen axis (x, y or z, pitch, yaw or roll). This is not foolproof as there is still drift, but it largely mitigates a majority of wrong movements. We also locked movements and rotations to one axis at a time, as this increases users’ precision with object placements.

7.2 Communications

7.2.1 Certificate Issues with ESP32

During the initial stages of testing, I attempted to get the PubSubClient to verify client key and certificates during the handshake with the MQTT Broker. However, due to it being an old library, the TLS implementation is unable to verify the server's certificate using its SAN field and is only able to connect if the CN field matches the server's address. To work around this, a separate port on the MQTT broker had to be created, where the server's certificates only has a CN field, while all other devices such as Ultra96 and visualisers use the same certificates.

7.2.2 Packet splitting for Voice Data

With the ESP32, there is a packet limit of 64kb when sending a payload using PubSubClient. Due to the serialisation required when using JSON, the voice data could not fit inside a single JSON payload. We realised that there were a lot of wasted spaces from delimiting values with a space (1 bytes). Since we are able to manage the sending and receiving of the data packet, we opted to send raw binary data from the ESP32 to the Ultra96, with the first bit designated as a flag to tell us what the mode was (SELECT / DELETE, see Data Packet Schema). The window of data was also broken into 3.

7.2.3 Learning AWS as a tech stack

I had learnt of the respective AWS resources before, but I have never touched it before. There was some difficulty when creating the CD pipeline, as well as getting EC2 to interface with S3 due to AWS' internal Identity Access Management (IAM) roles.

7.2.4 School WiFi vs Personal Hotspot

The initial design was created to allow the devices to connect anywhere as long as internet connection is provided. However, with latency testing, we realised that while internal wifi provided flexibility, it was more susceptible to network traffic and caused some packets to be dropped. With that, we decided to switch to using Personal Hotspot instead.

7.3 Hardware AI

7.3.1 Finding a suitable model architecture

Initially, I was quite lost on how to select the model architecture as there are so many available. My initial approach was the simplest Multi-Layer Perceptron (MLP) which did not bear fruitful results. I realized an MLP was too simple for our project's use case, hence I had to look for more complex models to be able to accurately analyse the voice data. An RNN did not make much sense as we were not working with time series data, so the natural choice for me was to look more into CNNs. This was where I found the ResNet-18 architecture which my finalized architecture was based on.

7.3.2 Managing the size of the model

Due to FPGA limitations, I am unable to use a model that is too complex in architecture. Since my input size is already considerably large, I had to reduce the model complexity multiple times in order for the RTL synthesis to take a reasonable amount of time. I started out using

the original ResNet-18 architecture, which took over an hour to synthesize on Vitis, but was impossible to generate the bitstream with on Vivado. From there, I just started removing layers which I thought were not that useful in our use case to simplify the model. Compared to ResNet-18, I used lesser channels, removed the skip connections and batch normalization layers, and arrived at the final architecture.

7.4 Software Visualiser

7.4.1 Using Unity Sample Project to map gestures

One of the initial development attempts involved trying to adapt the touch commands from Unity's standard AR sample project to function with our application's joystick and button-based controls. This approach proved to be highly impractical. While the sample project was useful for learning the basic setup of an AR scene, its internal logic for gesture-mapping (e.g., tap-to-place, drag-to-move) was deeply integrated and not modular, making it difficult to abstract and re-purpose for a different control scheme.

Ultimately, it was far more efficient to build the entire control system from scratch. By implementing our own `ObjectManager` to handle instantiation and selection, and a `ControllableObject` script that reads directly from our joystick components, we created a system tailored to our specific needs. This experience suggests that developers, especially those new to Unity, should be advised to code their own control logic to fit their project's requirements rather than modifying the complicated implementations of the sample project.

7.4.2 Race conditions when we use libraries

During an earlier development phase, an attempt was made to implement a stereoscopic VR mode using the Google Cardboard VR library. This feature was ultimately abandoned due to a critical software conflict with the Niantic Lightship SDK. We encountered a **race condition** where both libraries simultaneously attempted to acquire exclusive control of the device's primary camera asset. This conflict prevented the camera from initializing correctly, which in turn broke the functionality of both systems, as the Niantic SDK could not access the camera for object detection and the Cardboard SDK could not use it for its VR pass-through. In the end the group decided that given the timeline, it would be advisable for us to drop the VR idea.

7.4.3 Issues with Websocket on iOS

To implement the required **mTLS (mutual-TLS) WebSocket connection**, we faced a major limitation as no standard Unity or C# library supports this advanced security protocol on iOS. We overcame this by developing a **custom Objective-C plugin** (`WebSocketMTLS.mm`) that uses native Apple `NSURLSession` APIs to load the P12 client certificate and establish the secure connection. A **C# wrapper script** (`WS_Client.cs`) was then created to call this native code from Unity using `[DllImport]`. This **multi-threaded** architecture, with networking on a background thread and Unity on the main thread, required a thread-safe message queue (`UnityMainThreadDispatcher.cs`) that uses a **mutex** (`lock`) to safely pass data, preventing race conditions and application crashes.

Section 8 Future Work: Societal and Ethical Impact, Extension

8.1 Societal and Ethical Impact

8.1.1 Privacy Concerns

There could be concerns regarding privacy and user data during the use of our product. For one, users could be worried about the voice data being stored for commands. However, the app does not actually store the data and actually discards the data immediately after inference.

The second potential concern that users may have is that the app has access to the user's gallery, in order to download the screenshots. To assure users, we only download images into their gallery, but do not perform any reading of the gallery.

With the current implementation, any user can spoof as a different user and obtain the images without permission, potentially stealing information or even flooding the object storage with information. User authentication can help to mitigate this, along with a rate limiter for the image upload.

8.1.2 Accessibility

Through an interactive AR application, users with less mobility and/or design skills are able to create personalised environments without too much effort. The AR also enables users to comprehend spatial arrangements more intuitively, helping users with visual or learning disabilities in understanding how furniture pieces fit and look in their respective environments.

8.2 Expansion

8.2.1 Hardware

Future improvements to the form factor could be using a printed circuit board (PCB) to join the peripherals. The flat wires would help reduce the form factor by quite a large amount, as well as save a lot of hassle with wiring. The PCB would reduce the height of the remote.

Another improvement we could change with the hardware would be to make the battery more easily replaceable, and to use a smaller battery. Since we purchased the battery before we made the final changes to the remote, we ended up using a slightly larger battery than necessary.

8.2.2 Communications

There were initial plans to implement session-based user authentication to allow persistent AR state across devices. However, due to time constraints and prioritisation, we were unable to implement it.

A second improvement would be to allow the ESP32 to be configured for a specific user. By doing this, we can scale vertically to support multiple concurrent users for the AR application, as long as username matches what the ESP32 sends.

8.2.3 Hardware AI

For the audio recognition, the model can certainly be improved to add more vocabulary for recognition i.e. more furniture or more functionality within the application. The classification accuracy could also be much improved by training/testing with more recorded audio samples. It could even be possible to perform more than just command recognition where short phrases or sentences are recognized instead, making it more like an industry level speech-to-text model.

8.2.4 Software Visualiser

A significant future expansion would be to enhance the **Object Detection Model (ODM)**. This could involve training a custom Niantic model to improve prediction accuracy for specific furniture types. This data could then power a **context-aware suggestion engine**. For example, if the ODM identifies a "Sofa" but *not* a "Table", the application could proactively suggest that the user browse and place a table to complete the scene.

The current user identification, which relies on a single, locally saved **username**, could be upgraded to a **secure authentication system**. Implementing a proper login with verified usernames and passwords, tied to a backend database, would allow users to securely access their saved AR layouts and screenshot galleries across multiple devices, replacing the current simple `sessionId` system.

The **Photo Gallery's** functionality could also be expanded with advanced sorting and filtering. Currently, screenshots are sorted by date. A future improvement would be to add UI elements that leverage the existing metadata—such as the **epoch timestamp** used as the filename—to allow users to filter the gallery and view only screenshots taken on a specific date or within a certain time range.

Finally, a **gallery grouping feature** could be introduced to allow users to organize screenshots by topic. This would function like an "album" system within the `GalleryViewController`. Users could create custom groups (e.g., "Living Room Designs," "Bedroom Ideas") and assign their saved images to them, making it easier to manage and find specific design inspirations.

References

Data Packet Schema

MQTT Topics	Message Format
esp32/command	`{ "type": "MOVE" | "ROTATE" | "SCREENSHOT", "axes": [0.00, 0.00, 0.00] // optional float values }`
esp32/voice_data	<Binary data, first bit is flag> Flag: 0 – Select, 1 - Delete

ultra96/voice_result	`{ "status": "SUCCESS", // Returned if inferred successfully "info": { "command": "SELECT" | "DELETE", "object": "TABLE" | "CHAIR" | "LAMP" | "TV" | "BED" | "PLANT" "ODM" | "UP" | "DOWN", } }`
	
	`{ "status": "FAILED", // Returned if inference failed "info": { "error": <Error Message> } }`
	
	`{ "status": "DEBUG", // Returned if in debug mode "info": { "receiveTime": 1762447968643, "inferenceTime": 1000, "sendTime": 1762447971661 } }`

debug/status	`{ "type": "DEBUG", "timestamp": 12345 // time of data send }`

Fig 30. Message format from Visualiser to Golang service to Database

Websocket	Message Format
WebsocketEvent	`{ "EventType": "COMMAND_SELECT" | "COMMAND_DELETE" | "COMMAND_MOVE" | "COMMAND_ROTATE" | "COMMAND_SCREENSHOT" | "COMMAND_SET" | "COMMAND_DEBUG" | "DEBUG_VOICE_PING" | "DEBUG_VOICE_PONG" | "DEBUG_GESTURE_PING" | "DEBUG_GESTURE_PONG" | "SCREENSHOT_SEND" | "SCREENSHOT_RECEIVE" | "S3_UPLOAD_REQUEST" | "S3_UPLOAD_RESPONSE" | "S3_SYNC_REQUEST" | "S3_SYNC_RESPONSE" | "S3_DELETE_REQUEST" | "S3_DELETE_RESPONSE" | "S3_ERROR" , "UserId": <username>-<timestamp>, "SessionId": <username>, "Timestamp": <timestamp-unix-epoch>, "Data": <can be any data> }`

Fig 31. Message format from Visualiser to Golang service to Database

Software Visualiser Images

Images for section 6.2.1

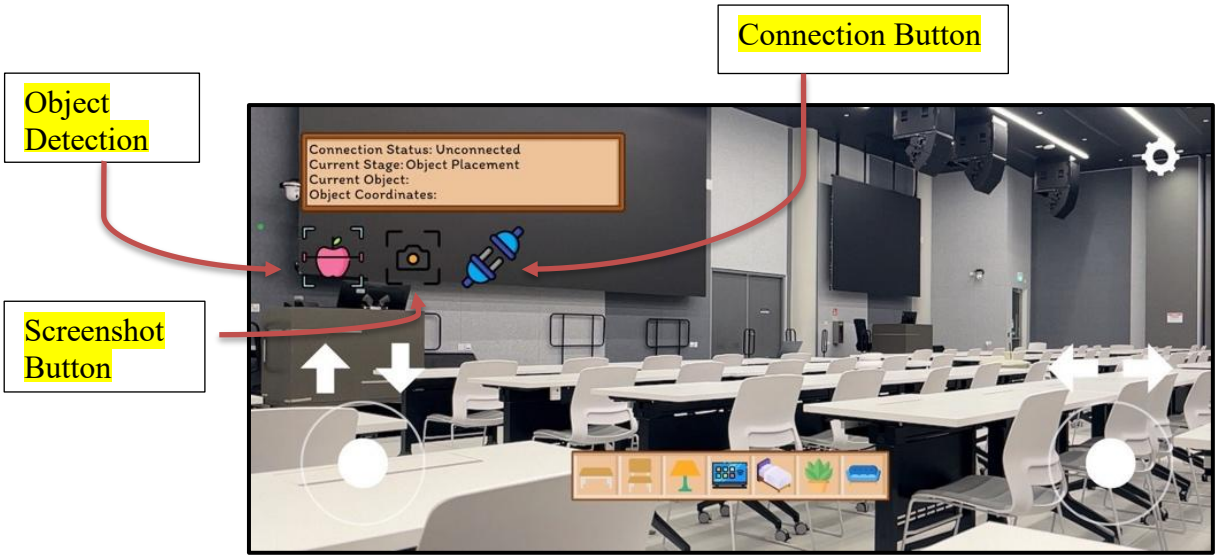


Fig 32. Example of Object Placement without object ***(unconnected to server)***

Fig 33. Example of Object Placement with object ***(unconnected to server)***

Fig 34. Example of Object Placement without object ***(connected to server)***

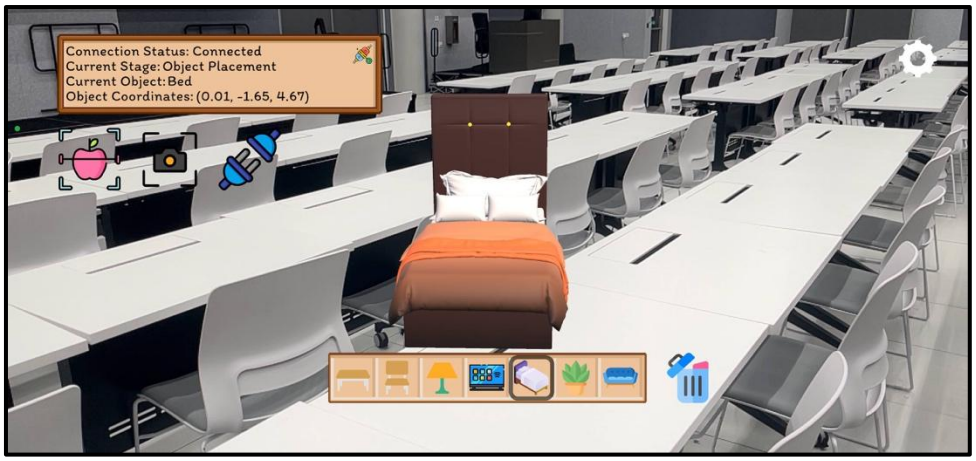


Fig 35. Example of Object Placement with object ***(connected to server)***

Click HERE to return back to section 6.2.1 of report.

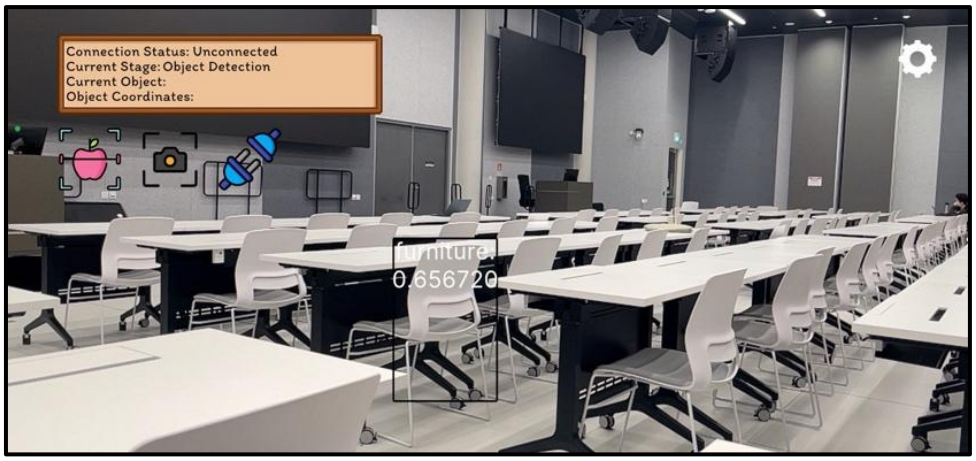


Fig 36. Example of data displayed on mobile

Click HERE to return back to section 6.2.2 of report.

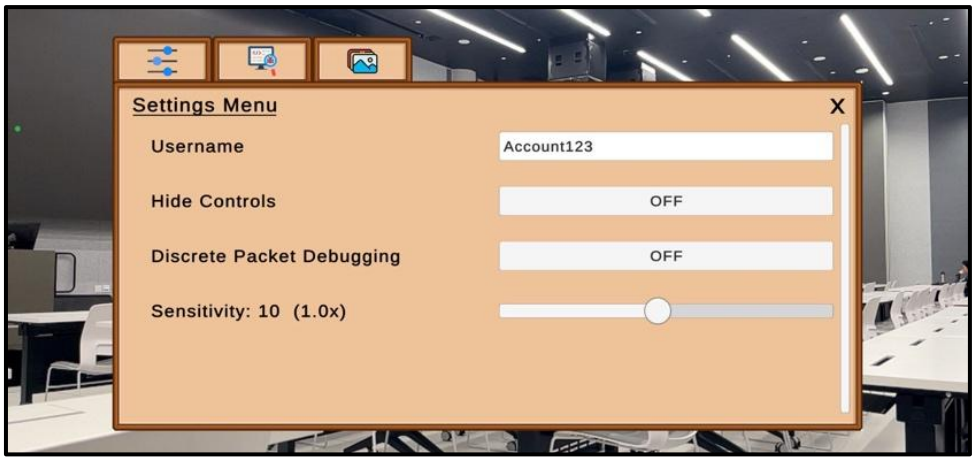


Fig 37. Example of settings page ***(unconnected to server)***

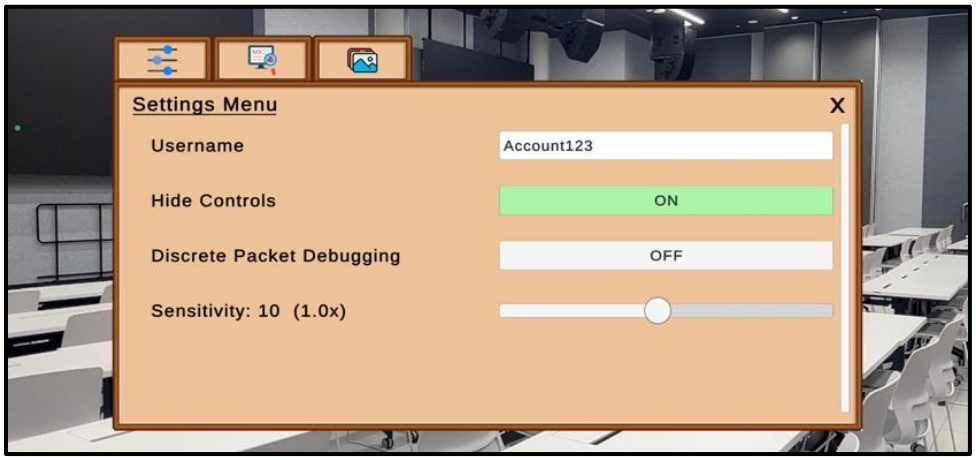


Fig 38. Example of settings page ***(connected to server)***

Click HERE to return back to section 6.2.3 of report.

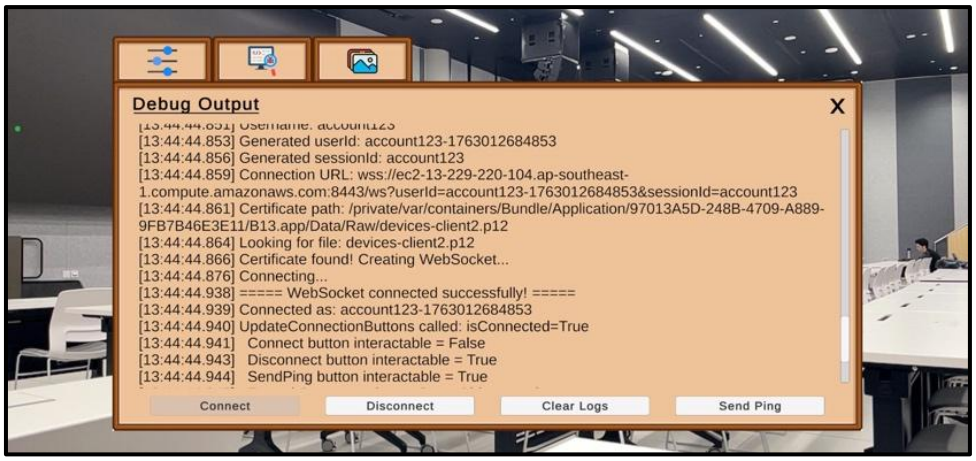


Fig 39. Example of debugs page ***(connected to server)***

Fig 40. Example of debugs page with logs cleared ***(connected to server)***

Click HERE to return back to section 6.2.4 of report.

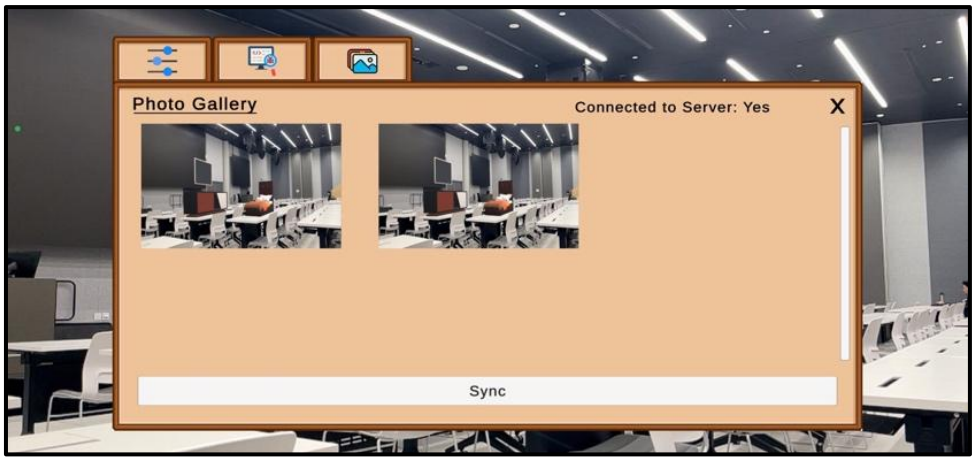


Fig 41. Example of photo gallery page with no sync required ***(connected to server)***

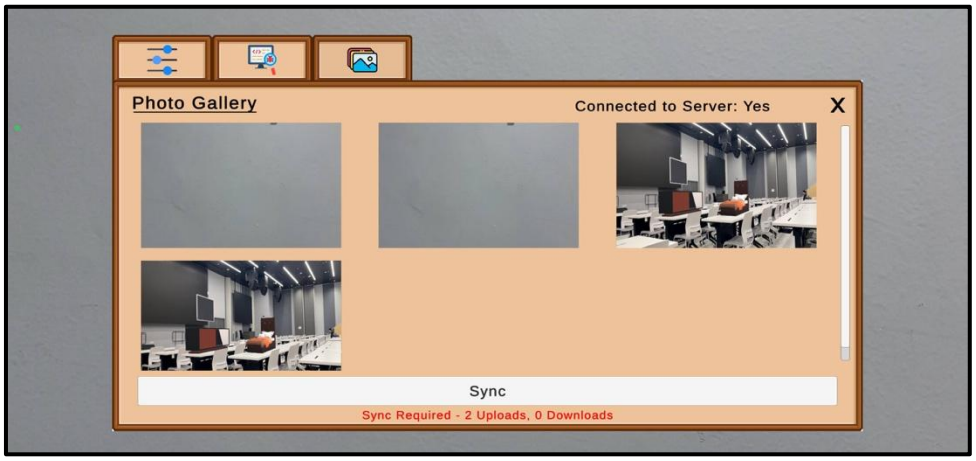


Fig 42. Example of photo gallery page with syncs required ***(connected to server)***

Click HERE to return back to section 6.2.5 of report.

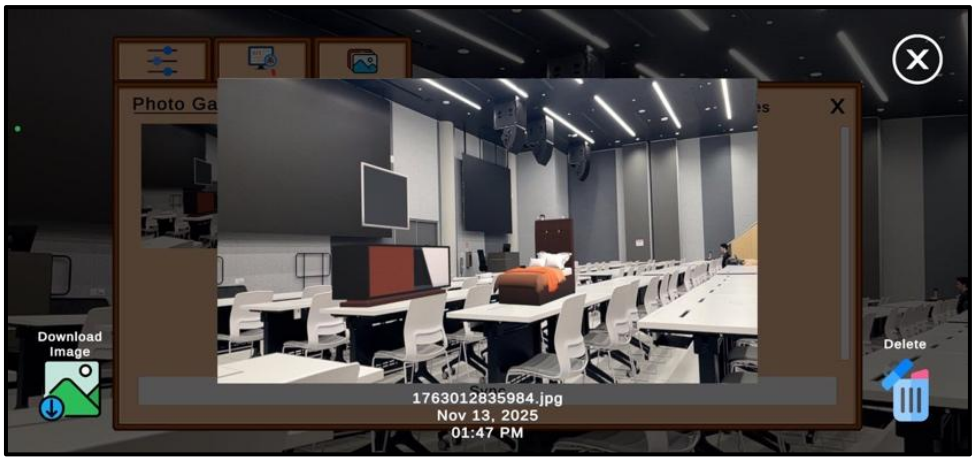


Fig 43. Example of image view page

Click HERE to return back to section 6.2.6 of report.

Commands used for audio recognition

There were 11 commands used in total for this project, listed as follows:
Furniture commands: table, chair, lamp, plant, TV, bed, sofa
Functionality commands: ODM, detection, up, down

Citations

FireBeetle ESP32 DFR0478
https://www.application-datasheet.com/pdf/dfrobot/dfr0478.pdf
https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

MPU6050
https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

Battery: 3.7V 1200mAh LiPo battery
https://www.mouser.sg/datasheet/2/419/TDS_UBP001-1122024.pdf

MAX17043 – 1 cell LiPo fuel gauge
https://www.analog.com/media/en/technical-documentation/data-sheets/max17043-max17044.pdf

INMP441
https://invensense.tdk.com/wp-content/uploads/2015/02/INMP441.pdf

MPU6050 Arduino Library by Electronic Cats
https://github.com/ElectronicCats/mpu6050

MAX1704X Fuel Gauge Arduino Library by SparkFun
https://github.com/sparkfun/SparkFun_MAX1704x_Fuel_Gauge_Arduino_Library

PubSubClient by knolleary
https://github.com/knolleary/pubsubclient

Reference for the Visualiser Object Detection Model Scripts
https://youtu.be/aZ9qAbeRwQ8?si=oQ6xOYaxvtjmOuvB

Reference for assets used in our project
https://www.youtube.com/watch?v=lF26yGJbsQk

Unity Assets:

1. https://assetstore.unity.com/packages/2d/gui/flat-pack-gui-307236
2. https://assetstore.unity.com/packages/templates/systems/free-inventory-system-for-2d-games-301357
3. https://assetstore.unity.com/packages/3d/props/interior/house-interior-free-258782
4. https://assetstore.unity.com/packages/tools/input-management/joystick-pack-107631

Icons used in the project:

1. https://www.flaticon.com/free-icon/image_1186124?term=image&page=1&position=20&origin=search&related_id=1186124
2. https://www.flaticon.com/free-icon/plug_6003122?term=plug+in&page=1&position=40&origin=search&related_id=6003122
3. https://www.flaticon.com/free-icon/stop_545666?term=square&page=1&position=2&origin=search&related_id=545666
4. https://www.flaticon.com/free-icon/delete_3221897
5. https://www.flaticon.com/free-icon/gallery_1375106?term=images&page=1&position=10&origin=search&related_id=1375106
6. https://www.flaticon.com/free-icon/debugging_15097118?term=debug&related_id=15097118
7. https://www.flaticon.com/free-icon/control_839395?term=settings&page=1&position=91&origin=tag&related_id=839395
8. https://www.flaticon.com/free-icon/object_10328775?term=object+detection&related_id=10328775
9. https://www.flaticon.com/free-icon/capture_8428546?term=screenshot&related_id=8428546
10. https://www.flaticon.com/free-icon/bed_3837744?term=bed&page=1&position=16&origin=search&related_id=3837744
11. https://www.flaticon.com/free-icon/plant_628283?term=plant+pot&page=1&position=7&origin=search&related_id=628283
12. https://www.flaticon.com/free-icon/smart-tv_1023572?term=tv&page=1&position=13&origin=search&related_id=1023572