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

AI Tool Declaration

We used GPT-5 to source for alternative components when considering the most suitable components to be used for the devices. One such prompt would be “What other alternative power sources would be suitable for the DFRobot FireBeetle, which are lighter than using a power bank?”

We also used GPT-5 to rephrase my thoughts into prose form. The prompt used was: “Would it be possible to put these points into proper sentences? <insert points in point form>.”

We are responsible for the content and quality of the submitted work.

Table of Contents

Section 1 System Functionalities

Problem statement: Current 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:

- Plane detection via smartphone camera
- AR generated furniture in a virtual room anchored to a plane
- Hardware gesture controls that are interpreted by AI
- Movement & Rotation of furniture around all 3 planes
- Selection of furniture items through AR menu
- Swapping of furniture textures via gesture control

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 my hands, so that I can interact with the environment without using a controller.
- As a user, I want the app to recognise the gestures in real-time, so that the AR app feels smooth and natural.

Section 2 Overall System Architecture

2.1 Proposed System Architecture

2.1.1 High Level System Architecture

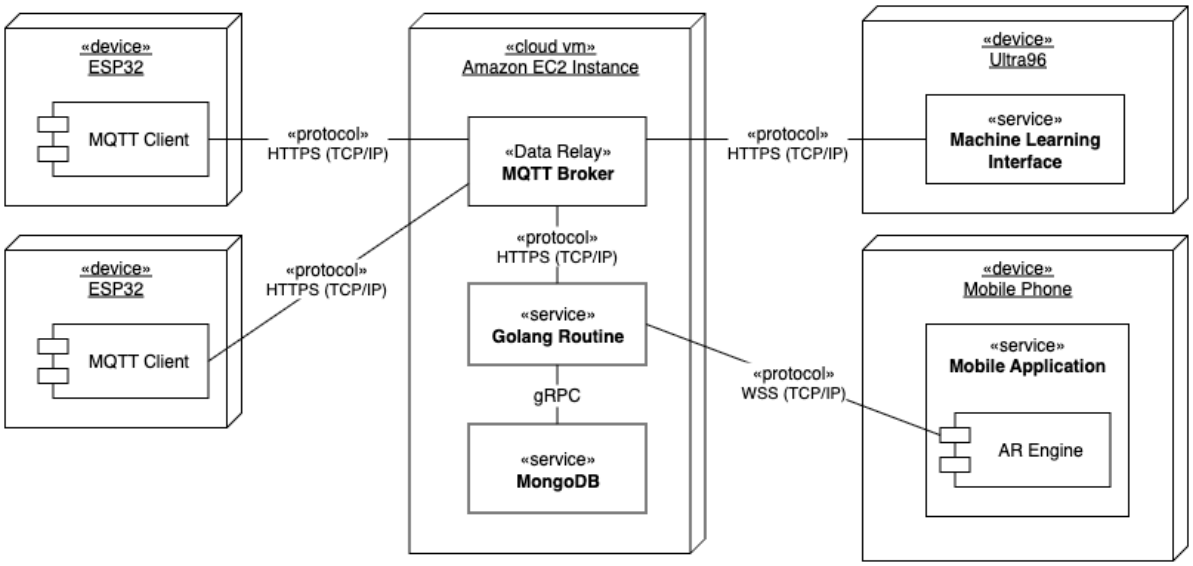

Fig 1. UML deployment diagram for proposed architecture

2.1.2 FireBeetle (ESP32)

The FireBeetle 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 FireBeetle to the Ultra96 is unidirectional, and further communications breakdown will be discussed in section 4.

2.1.3 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.

2.1.4 Backend on 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 service will be running in separate containers managed by docker compose.

2.1.5 Mobile Phone / Visualiser

The visualiser utilizes mobile phones running the Unity-based AR application that renders the virtual furniture models with the physical environment. It subscribes to updates from the MQTT broker, processes the gesture and object data to provide an interactive interface for users to place, manipulate, and anchor objects in the AR scene.

2.2 Main Algorithm

The user flow is as follows:

1. The user makes a hand gesture through the wearable, gesture data is recorded and published to MQTT via EC2.
2. The Ultra96 listens for incoming gesture data via MQTT. The data is sent into the ML accelerator for inference.
3. With a result, the Ultra96 publishes the gesture result to MQTT via EC2.
4. The Golang Service listens for incoming gesture data 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. When there are events generated (i.e. furniture placed, rotated, deleted etc), the visualiser will send an event to the Golang service.
6. The service will store and append that state into the database.
    a. The state will be stored per session.

2.3 System Description

The system comprises of a glove which the user will wear to transmit gesture data to the Ultra96 for inference. The glove will have a FireBeetle connected to a MPU6050, which senses the user's movements by measuring changes in the acceleration and angles. The additional features of the system include a pushbutton for alternate controls, as well as a battery percentage monitor.

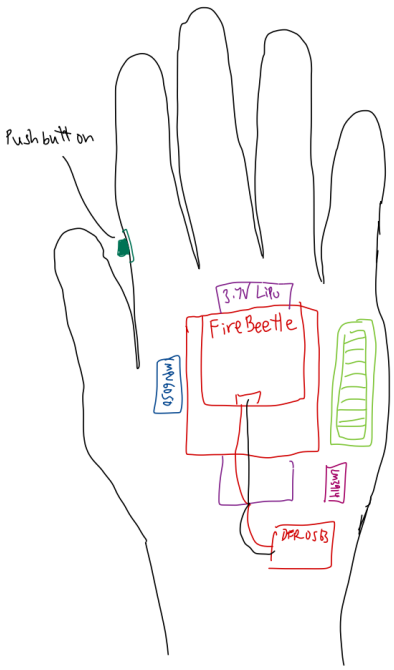

Fig 2. Diagram of locations of circuits on the wearable glove

Section 3 Hardware Sensors

3.1 Introduction

The Micro Controller Unit (MCU) that we are using is the DFRobot FireBeetle, which is a MCU powered by an ESP32. The main benefit this MCU 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 sensor that we will be using will be a MPU6050, as it has a 3-axis gyroscope and accelerometer respectively. This allows for collection of a wide range of data, for better classification of gestures and finer controls.

Additionally, we plan to include a button for separate controls, such as toggling from moving objects to rotating them, as well as selecting items via the menu.

To improve the user experience, we will also be attaching a fuel gauge for users to better estimate the remaining lifespan of the battery before it has to be recharged, so that users are aware of when they need to charge the battery.

Lastly, to power the entire system, we have chosen a LiPo battery. This is due to its lightweight form factor, as well as it's 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 choices before concluding with a pseudocode section of the main programme running within the FireBeetle.

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 DFRobot FireBeetle, powered by the ESP32. It contains 30 GPIO pins, which are multiplexed to perform other functions, such as I2C Communications, UART and so on. As aforementioned, Wi-Fi will be the main mode of communication between the FireBeetle and other devices, which the communication will be outlined in future sections.

The MPU6050 motion sensor 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 calculate the pitch, roll and yaw, as well as use the acceleration in each direction to 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, at 50Hz, which is within the maximum Serial Clock frequency. It is also logical to sample at 50Hz because a recognisable gesture should not be less than 20ms in duration.

I intend to append a pushbutton onto a free pin on the FireBeetle, as it allows me to create an additional layer of control for the user. This is done in a similar fashion to an interrupt, altering the gestures to perform a different task.

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 Type C port on the FireBeetle. The JST PH2.0 output also lends to another feature

of the product, using a LED Bar Graph to represent the remaining charge within the battery. This is done with a 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 FireBeetle through I2C.

Furthermore, from the I2C data received from the DFR0563, we can use logic to convert the battery percentage into an analog output, which can be passed into a LM3914 LED driver, to drive LEDs in the LED Bar Graph.

3.3 Schematics and Pinouts

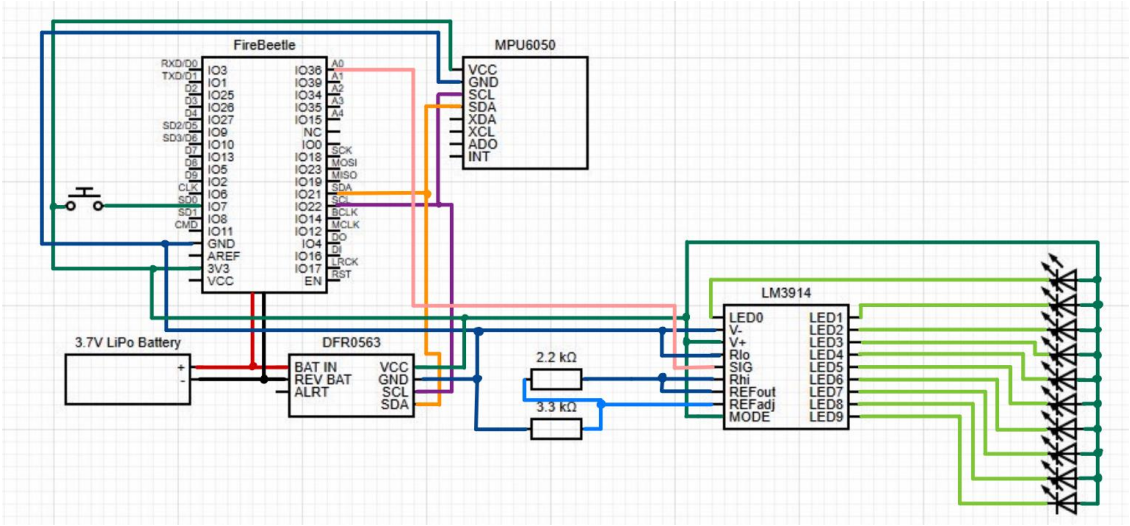

Fig 3. Proposed schematics of complete circuit

The FireBeetle 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 FireBeetle itself is powered by the 3.7V LiPo battery, through its JST 2.0 port.

One advantage of using the LM3914 to drive the LED bar is the ability to display the battery level in 'bar' mode, i.e. if the battery is 80-90% we can light up 9 LEDs, and so on. Another advantage is that we can adjust the current drawn by the LEDs, by controlling the current drawn in pin 7. The calculations will be done in Section 3.4 below.

The figures on the next page show the pinouts of the various devices being used:

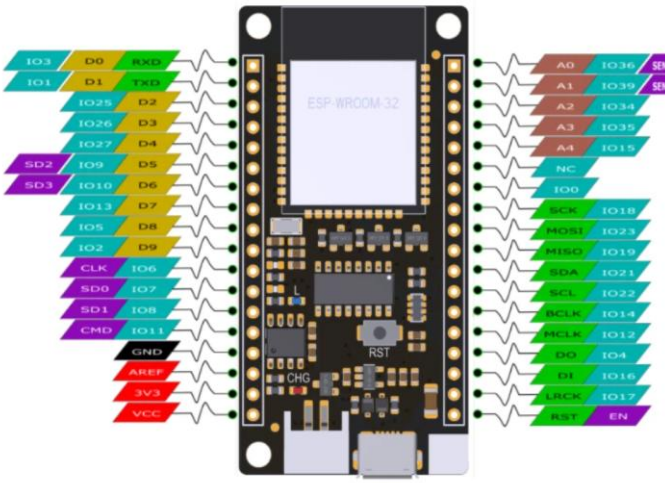

Fig 4. Pinout of DFR0478 FireBeetle

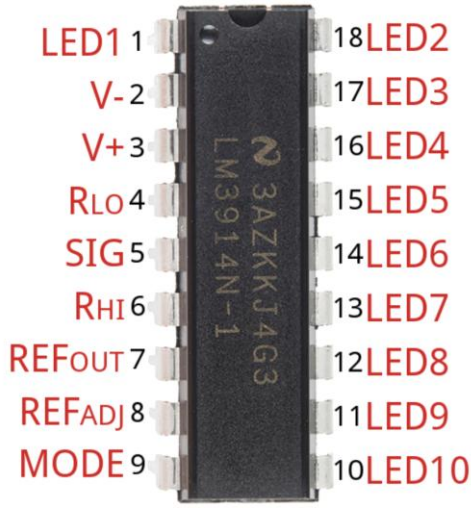

Fig 5. Pinout of LM3914

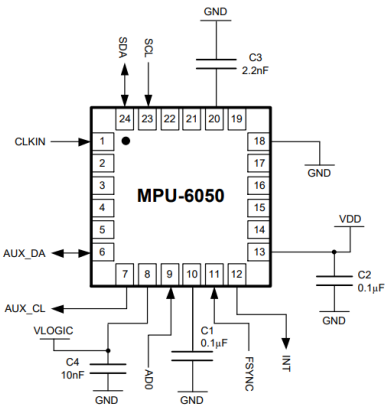

Fig 6. Pinout of MPU6050

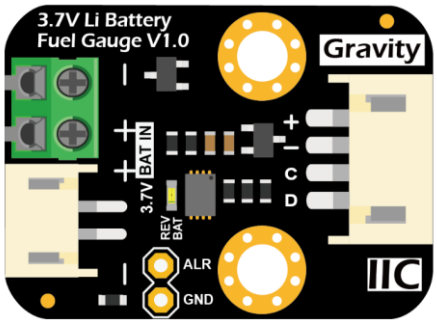

Fig 7. Pinout of DFR0563 Fuel Gauge

The table below describes the pin connections between the FireBeetle and the various components.

Component	Component Pin	FireBeetle 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
LM3914	V+	3V3	
	V-	GND	
	SIG	A0/IO36	Analog output
	Rlo	GND	
	Rhi	GND	Across 2.2kΩ Resistor
	REFout	GND	
	REFadj	GND	Across 3.3kΩ Resistor
	MODE	3V3	Enables Bar Mode
	LED0-9	LEDs	
Pushbutton	-	IO7	3V3 to IO7

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)
LM3914	3mAh
LED Bar	72mAh (10 x 7.2mAh per LED)

The Voltage for the LED Bar is controlled by the high end of a Voltage Divider, and the Current drawn for the LED is the same as the current drawn out of Pin 7 (REFout). For our use case, we will be omitting the Voltage Divider, and instead only control the current of the LEDs. The resulting I_{LED} equation can be approximated to:

$$I_{LED} = \frac{12.5V}{R_L\Omega}$$

For a balance between brightness and power management, the ideal current should be between 7-9mAh. Based on the available resistors, we will be using a 2.2kΩ and 3.3kΩ resistors, such that the resultant current is 7.2mAh.

For 2h of continuous use, total power required is

$$I_{total} = 2h * (80mAh + 240mAh + 4mAh + 75mAh)$$
$$= 798mA$$

Accounting for additional power draw due to surges and requirements for capacity loss over time, a factor of roughly 1.5x is used.

3.5 Libraries used

The FireBeetle itself can be programmed using the Arduino IDE, as it contains the relevant plugins to access the pins and flash the board efficiently. The main components that we have to program in conjunction with the FireBeetle are the MPU6050 and DFR0563, which uses the MAX17043 chipset.

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.

Sources of the libraries can be found in the References section below.

Section 4 Communications

4.1 Message Queue Telemetry Transport (MQTT)

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 gesture triggering a certain action.

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

4.1.1 /ESP32 Topic

This topic is used for the publishing of gesture data from the ESP32. When a gesture is detected, the data is packaged and sent via HTTPS to the broker, after which the broker will distribute to all subscribers (Ultra96). The data packet sent will be in JSON format.

4.1.2 /AR Topic

This topic is used for the publishing of processed gesture data after the Ultra96. When the gesture has been classified in the Ultra96, the result is sent via the /AR Topic. The Golang service will pick it up and transmit it to the AR, as well as stored into the database for logging purposes.

4.2 Amazon Elastic Compute Cloud (EC2) Instance

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 managed 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 FireBeetle to connect to the MQTT broker under any network, as long as the IP address is known. Additionally, the EC2 instance is able to reverse tunnel SSH into the Ultra96 under the Sunfire SoC VPN, allowing for us to work with both hardware and inference without local host discovery issues. Certain ports can be exposed by editing the instance's inbound rule to allow MQTT, as well as WebSocket connections.

4.2.1 Microservices

Within the EC2 instance, there will be a set of microservices orchestrated by docker compose. The services consist of the MQTT Broker, the Golang service as well as a database (MongoDB) for session storage. The Golang service aims to be the middleware between gesture information and the visualiser, by managing and manipulating the state of the visualiser and saving that state in the database.

The Golang service will interface with the MQTT broker by subscribing to the /AR topic, and communicate with the database via gRPC (faster than REST API). This allows for better separation of concerns and better maintainability of code and function. At the same time, the Golang service will connect to the visualiser through secure WebSocket.

4.3 Encryption

MQTT supports the usage of HTTPS, which requires the client to enter a username and password before being able to connect to a topic and publish to it. Within the Secure WebSocket (WSS) connection between the Golang service and the Visualiser, a self-generated key-cert pair is generated for use by the WebSocket server.

4.4 Availability and Consistency

Since there is no requirement for scaling, there is no consideration required for partition tolerance, allowing us to focus on both availability and consistency. With all connections using either HTTPS or WSS, there is a guarantee of packet delivery on the transport level due to both using TCP. By using MQTT for each gesture data generated by the FireBeetle and each gesture inferred by the Ultra96, along with MQTT's QoS guarantee, we are able to ensure that no gesture is lost. Any gestures sent into MQTT that is not received and acknowledged by the consumer would be retained and stored within the queue buffer.

The goal for this is eventual consistency, whereby the state of the visualiser is stored in the database. This also means that when there is no gestures being processed and sent, the visualiser is able to re-capture the last available state of the AR environment.

4.5 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.

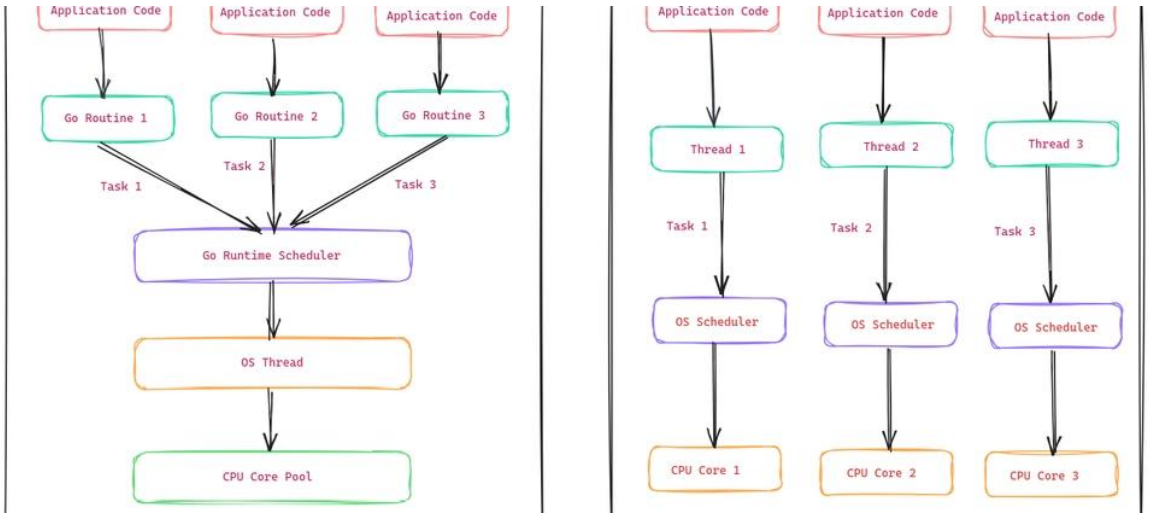

Fig 8. Go Routines vs traditional threads

4.6 Message Format

For readability and ease of parsing, all data will be in JSON format and base64 encoded. This format is also supported by MQTT, allowing us to send the data more easily.

```
{
  "gestureData": "abcdefg"
}
```

Fig 9. Message format from ESP32 to Ultra96

```
{
  "gesture": "place-item"
}
```

Fig 10. Message format from Ultra96 to Golang Service to Visualiser

```
{
  "items": {
    "chair": 1,
    "table": 2
  },
  "position": {
    "chair": [2, -1, 10],
    "table": [5, -5, 0]
  }
}
```

Fig 11. Message format from Visualiser to Golang service to Database

It is worth noting that these message formats are only a draft, and the final message format might be different from what is proposed.

Section 5 Software/Hardware AI

5.1 Neural Networks to be explored

The simplest neural network design, the Multi-Layer Perceptron (MLP), will first be analysed. This design consists of multiple layers of fully connected perceptrons in each layer with nonlinear activation functions and is popular for being able to distinguish data that is not linearly separable. By stacking several layers, the MLP is able to progressively learn higher-level features from the input data, making it a versatile model for a wide range of tasks such as classification, regression, and function approximation. Although relatively simple compared to more advanced architectures like CNNs and RNNs, the MLP remains a fundamental building block in deep learning and often serves as a baseline model for evaluating performance.

Our input data consists of sensor readings collected by the accelerometer and gyroscope over a small window. Each sensor provides reading in the x, y and z axes, so we have 6 readings per timestep. Each sample will consist of readings from multiple timesteps to capture the entire gesture, which will then be flattened to form the input vector for the MLP. In this way the model can identify patterns within the input data and improve its classification accordingly.

We are also looking to explore Recurrent Neural Networks (RNN), which are specifically designed to process sequential data where the order of elements is important. In this case, the action of performing a certain gesture can have multiple mini-actions that string together over time, such as the initial movement, transition, and final position of the hand. Capturing these temporal dynamics is crucial, as two gestures may contain similar individual poses but differ in the sequence and timing of movements. RNNs are well suited for this task since they maintain a form of memory that allows them to model how sensor readings evolve across time, enabling more accurate recognition of gestures that are defined by their progression rather than any single snapshot.

The below figure illustrates the difference between the RNN (left) and MLP (right) architecture.

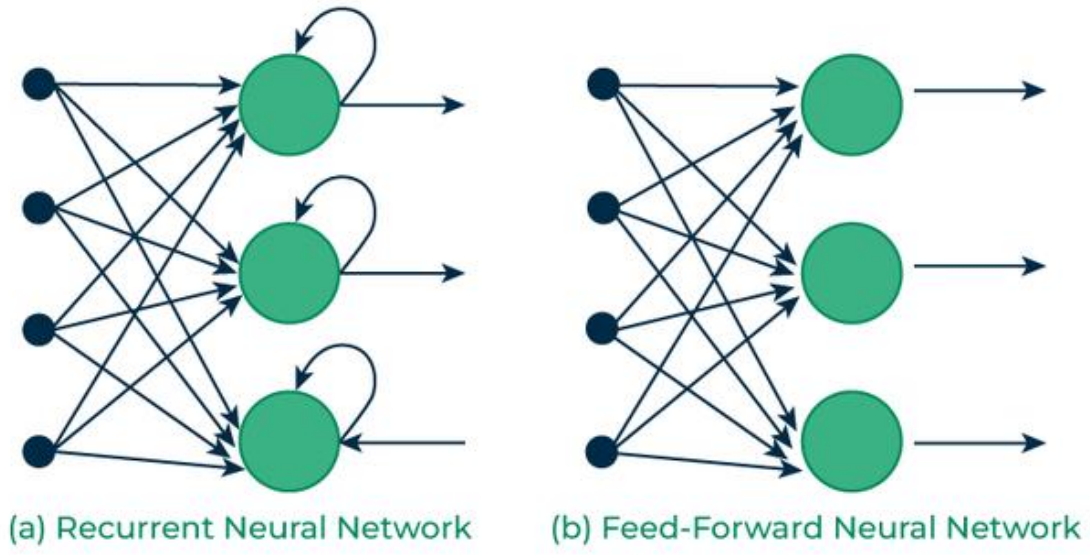

Fig 12. RNN and MLP architecture

5.2 Training and Testing of model

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

Data collection

This phase is crucial as the quality of data used in training our model is of utmost importance. We need to ensure that each gesture is performed exactly as intended to gather the most consistent readings from our accelerometer and gyroscope. Readings will be captured in a small window to ensure the entire gesture is recorded from start to end.

Data preprocessing + Feature engineering

Firstly, normalization would have to be done to standardize the ranges across the different sensor readings.

Feature engineering can also be done here to select more useful features to reduce the dimensionality of the input. Techniques such as Principal Component Analysis (PCA) can be utilised to create new features that are more helpful in classifying the input data.

Model training

In the model training phase, the dataset is first separated into training, validation and test data. The network depth and width for the neural network is also decided. Initially the model will be trained on the training data and evaluated with the validation data. While training, we can adjust certain hyperparameters for the neural network and the optimizer to improve the model performance. This can be done through cross-validation, and techniques such as k-Fold Cross Validation or even Leave One Out Cross-Validation can be used here. Once we have found the best hyperparameter setup for the model, we retrain the model with both the training and validation data together.

Model evaluation

The model is finally evaluated on the test data that was unseen during the training phase. We are interested in various metrics such as accuracy, recall, precision and F1-score. Analysing these metrics together will provide a good representation of the model quality, which can help guide us in improving it.

5.3 Realising the model on the FPGA

For the ML models that we will be exploring, the training is first done on Python for ease of implementation. We can do this via libraries like PyTorch and TensorFlow. After we are satisfied with the model performance, the model weights can then be extracted out for usage later.

Next, our chosen ML model has to be reimplemented in C++ for Register Transfer Level (RTL) synthesization in the Unified Vitis IDE. In this stage, the model is first recreated in C++, then a testbench is run on the recreated code to make sure that the reimplemented model gives the same output as the Python model. We can do this through the “C Simulation” function available on the Vitis IDE. Afterward, the C++ implementation can be converted to Hardware Description Language (HDL) via “C Synthesis”. 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 and include our custom ML IP block into it. 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 we also need to run a

Python script using the Python Productivity for Zynq (pynq) to 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.

5.4 Power efficiency estimation

We will be using the Xilinx Power Estimator spreadsheet provided by AMD to estimate the power usage of our Ultra96.

Below is a snapshot of the spreadsheet that we will be using:

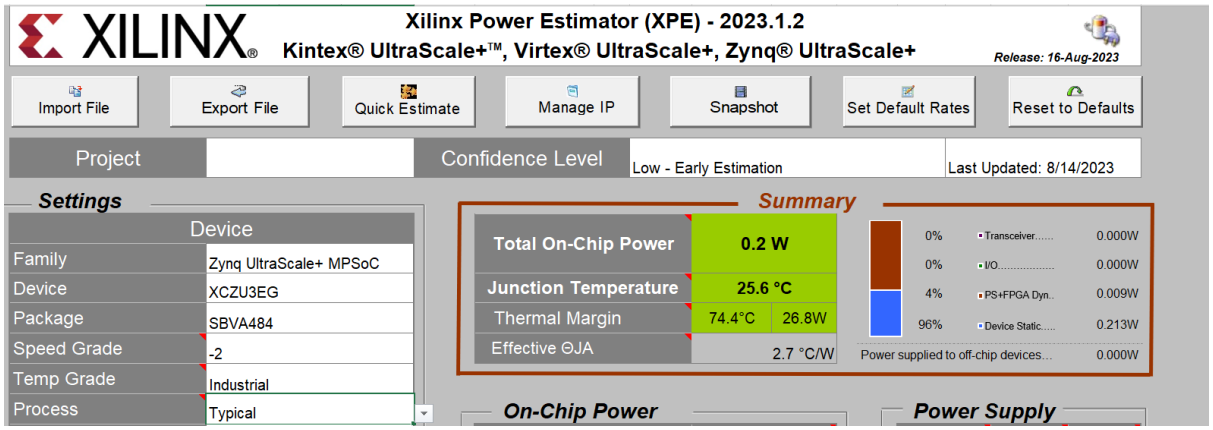

Fig 13. Xilinx Power Estimator spreadsheet

Section 6 Software Visualizer and Game Engine

6.1 Visualizer Software Architecture

6.1.1 Software Framework and Libraries

The visualizer is developed using **Unity 2022 LTS with the Built-in Render Pipeline**, which provides the foundation for rendering the augmented reality environment, displaying system data, and overlaying virtual furniture models within the physical space.

Communication between the phone visualizer(s) and the backend running on the EC2 instance will be conducted over TLS, ensuring encrypted and reliable data exchange across platforms.

6.1.2 Modules Utilized

The visualizer will rely on **AR Foundation** to provide the essential augmented reality functionality, including plane detection, ray casting, anchoring, and the overlay of virtual furniture models within the physical environment. It will also support the user interface and overlay components, enabling the integration of menus, selection tools, and feedback displays directly into the AR scene. Gesture inputs will be used to navigate these interfaces and apply selection choices. A dedicated persistence and anchor management module will ensure that placed objects remain stable and continuous across sessions by creating and tracking anchors, saving their positions, and generating coordinate data on the master device for synchronized updates across all connected devices.

To handle communication, the system will use TLS-secured channels between the phone visualizer and the EC2 backend, which orchestrates all device interactions. The backend, in turn, interfaces with the MQTT broker and the Ultra96. This framework will publish updates on object placements, transformations, and gestures, while also subscribing to incoming data from external sensors and the FPGA. Reliability and security will be supported through IL2CPP-compatible MQTT libraries and encryption mechanisms. Complementing these layers, **ARKit/ARCore** will provide device-specific functionality such as motion tracking, environment understanding, and camera access, ensuring accurate rendering and seamless interaction with the AR environment.

6.1.3 Phone Placement

We will be utilizing 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 needing to manually handle the device.

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

6.2 Visualizer Design 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 Data Displayed on Phone Screen

- Connection Status
  - Informs user of the connection with the Amazon EC2 Instance
- State Status
  - Informs user of the current state (Initialisation/Object Detection/Object Rendering)
- Object Status
  - Object currently being handled (Table/Chair/Lamp)
  - State of motion (Axial/Rotary Motion – Controlled using the button)
  - Object Anchor Status (Anchored or Still Moving)

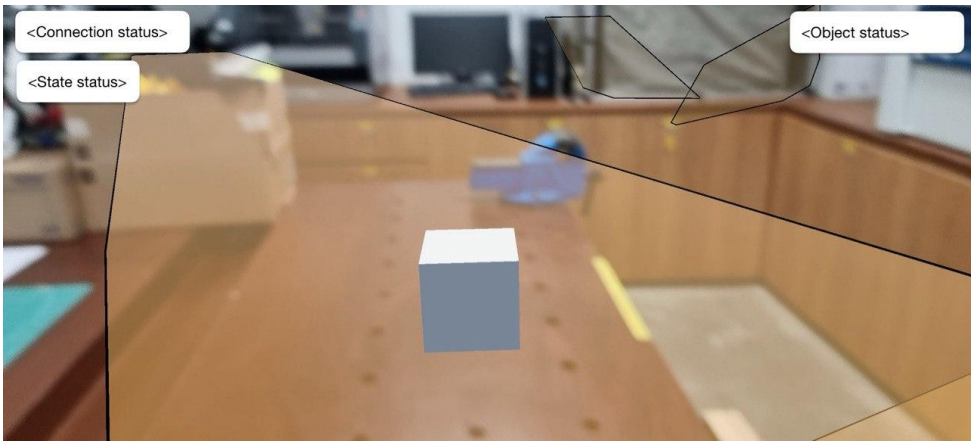

Fig 14. Example of data displayed on mobile devices

6.2.2 Object Detection Model

When the visualizer operates in object detection mode, a unique gesture will be used to trigger capture of the current camera frame within Unity (via the AR camera feed). The captured image will then be pre-processed and passed to an on-device inference pipeline integrated into the app. The model will identify common household objects within the frame and return class labels together with their corresponding 2D bounding boxes.

These bounding boxes will then be used to render labels directly on the AR camera feed, allowing the detected objects to be highlighted and annotated in real time. This Unity-native workflow eliminates server roundtrips, reduces latency, and enables real-time recognition and labelling to operate entirely on-device.

6.2.3 Object Placement

Once the system transitions into the object placement state, axial gestures will be used to select the furniture model that is to be introduced into the scene.

The selected object will initially be positioned on the closest stable ground plane detected by the AR session, ensuring that placement begins with a realistic and consistent foundation. Following this, the user will be able to refine the position and orientation of the object by employing axial and rotary gestures. These gestures will allow translation, rotation, and scaling adjustments to be made interactively, giving the user control over the exact layout of the virtual furniture.

When the desired position has been achieved, a button input will be used to anchor the object definitively to the selected location. The placement of each object will also be recorded relative to the initial frame of reference established during system initialization.

This information, including the object identity and its coordinate data, will then be transmitted to the server so that the same updates can be synchronized across other connected slave devices.

6.2.4 Integration of multiple devices

To support collaboration, the system will allow up more than one mobile device to be initialized within the same environment. The first device to complete initialization will assume the role of the master, while the subsequent device(s) will operate as slaves.

All actions within the visualizer will be performed through the current master device, and the corresponding data describing object placements and label anchors will be transmitted to the server. The server will then broadcast these data packets to all connected slave devices, enabling them to update their scenes accordingly. This master-slave configuration is required to maintain a consistent and conflict-free scene across all devices. Allowing multiple devices to make changes simultaneously could lead to race conditions, where conflicting updates to the same object would cause inconsistencies in placement, orientation, or anchoring.

As all devices are initialized from the same starting point in the environment, their coordinate frames remain aligned, allowing slave devices to reproduce the same virtual scene as the master. As a result, each participant will observe an identical virtual scene within their respective displays.

Control of the session will not remain fixed to a single device. A dedicated gesture will be used to transfer master control from one device to another device, ensuring flexibility in collaboration. The newly assigned master device will assume responsibility for transmitting data packets to the EC2 instance, while the EC2 instance continues to manage broadcast to the slave devices.

This architecture ensures that the augmented reality experience remains consistent and coherent across all connected devices, regardless of which device currently holds master control.

6.3 Sample flow of the run of the application

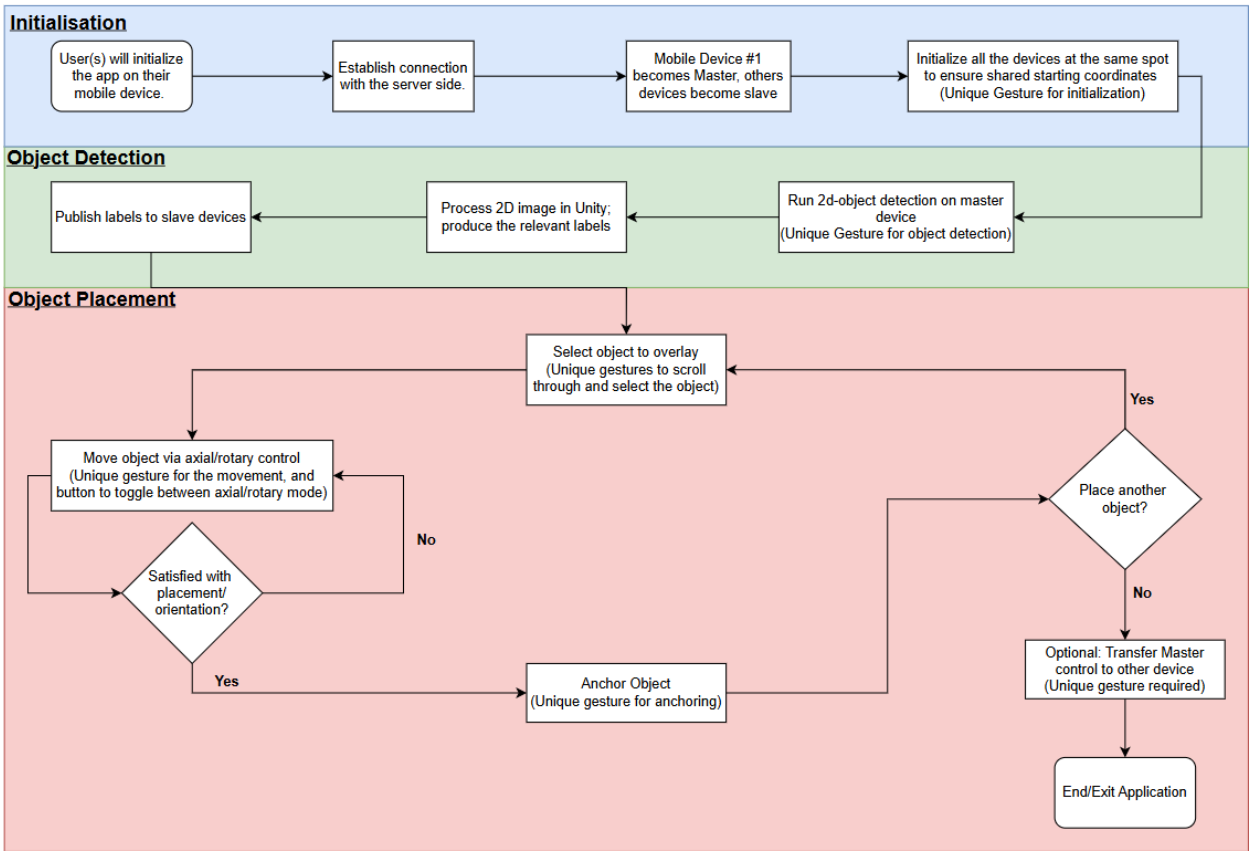

Fig 15. Sample flow run

6.4 Other additional features being considered

6.4.1 Google Cardboard Box Implementation with a VR headset

As an experimental extension, we are considering supporting the Google Cardboard framework to allow the visualizer to be used with low-cost VR headsets. This implementation would provide stereoscopic rendering of the AR content, giving users an immersive mixed reality experience.

Although not essential for the core system, this feature would allow us to evaluate whether hands-free, headset-based interaction improves usability compared to the handheld or neck-stand modes.

6.4.2 Usage of 3d Projection Data as an alternative to the 2d Image

Another potential extension is the use of 3D projection data from ARKit's scene depth (or LiDAR on supported devices) as an alternative to relying solely on 2D images for object detection.

By leveraging dense depth maps, the visualizer could more accurately localize objects in three-dimensional space, improving the stability of anchors and reducing ambiguity in placement. This approach also enables exploration of hybrid detection techniques, where 2D object recognition is combined with depth information to enhance precision in annotating real-world objects.

Section 7 Project Management Plan

7.1 Team Structure

Drawing inspiration from one of our most enjoyable modules, CS2113 (Software Engineering & Object-Oriented Programming), we decided to adopt an egoless team structure.

This approach allows every member to participate equally in decision-making and problem-solving, ensuring that diverse perspectives are considered. It encourages collaboration and helps us arrive at well-rounded solutions.

To reduce the risk of losing direction due to the absence of a fixed leader, we'd assigned our primary components based on each member's strengths and weaknesses during the capstone planning stage. In situations where consensus cannot be reached, the final decision will rest with the member responsible for the relevant component.

7.2 Overall Timeline

Week	Main actions	Remarks
2	Research + Begin individual tasks	
3	Begin individual tasks + Initial Report	Complete & Submit Design Report by 28 August 2025
4	Continue individual tasks	
5	Continue individual tasks + Modular Testing	
6	Continue individual tasks + Modular Testing + Integration Prep	**Gear up for integration**
R	Complete individual tasks + Modular Testing + Integration Prep	**Gear up for integration**
7	**Begin** integration + **Prep for Progress check**	
8	Integration	**Progress check**
9	Troubleshoot Issues + Refine Product	
10	Troubleshoot Issues + Refine Product + Testing	Modular + Integrated Testing Required
11	Complete product + Testing + Final Report	**Start Final Report**
12	Improve reliability + Testing + Final Report	**Peer review**
13	Final Report submission	Project Presentation & Final demo

7.3 Detailed Gantt Chart

Component	Tasks	Dependencies	Weeks												
			1	2	3	4	5	6	R	7	8	9	10	11	12	13
Hardware	Research on Hardware Components															
	Complete Schematic Design															
	Complete Initial Report															
	Program FireBeetle with MPU6050															
	Program FireBeetle with Pushbutton Interrupt															
	Configure Communication protocols for integration															
	Add Battery Fuel Gauge to FireBeetle															
	Create wearable gadget															
	Refine prototype															
	Complete prototype and testing															
	Write report															

Component	Tasks	Dependencies	Weeks												
			1	2	3	4	5	6	R	7	8	9	10	11	12	13
Communications	Research on Communication Protocols / Learn Golang and RPC															
	Set-up EC2 Cloud instance with public facing IP address															
	Configure MQTT for testing locally (Docker Container)															
	Add MQTT Clients for Firebeetle															
	Add MQTT Clients for Ultra96															
	Write Message Handling Service															
	Create and interface with database storage															
	Setup deployment pipeline for EC2 (docker service)															
	Perform E2E testing															
	Write Report															

Component	Tasks	Dependencies	Weeks												
			1	2	3	4	5	6	R	7	8	9	10	11	12	13
AI	Research on AI Models and how to realize the hardware AI															
	Develop and train model on Python with dummy data															
	Reimplement model in C++ for RTL synthesis															
	Synthesize IP block with Vitis, create bitstream in Vivado															
	Set up Python script (with pynq) on Ultra96 to access FPGA															
	Generate actual training/test data															
	Train and run model with the actual data															
	Write report															

Component	Tasks	Dependencies	Weeks												
			1	2	3	4	5	6	R	7	8	9	10	11	12	13
Visualiser	Research on Unity Engine and Vuforia															
	Installing and learning Unity															
	Implement Basic Plane Detection Model with object anchoring															
	Add more objects (Furniture) and test movement in all axis															
	Add object detection model															
	Implement communication protocols with other parts of the project															
	Implement using multiple iOS devices															
	Integrate with other components															
	Testing of overall integration															
	Add enhancements															
	Write report															

References

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

LM3914N – 10 LED driver
https://www.ti.com/lit/ds/symlink/lm3914.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

Mobile Phone Neck Stand Images
Image 1:
https://shopee.sg/product/451027126/25783303823?gads_t_sig=VTJGc2RHVmtYMTlxTFVSVVRrdENkY0N5akVpcE5OamJEdjRRTHBOTzhmbkIvbGdWUVdBQTFPaWtNa25nNGFyWFBYM3NPRmxkakxQY0JHWmYvWVBrZTVITUloK21zNDNwdHhvM24vaVFzUFg2VFFscFJHaWVJK0lnaGNzWWZaQUw&gad_source=1&gad_campaignid=1691142424&gbraid=0AAAADPpRQTaKZw20SDD9mTyKpOVIpXWP&gclid=EAIaIQobChMIzLyE3omrjwMVU6VmAh3wrgHcEAQYAiABEgLj2_D_BwE

Image 2:
https://www.amazon.sg/ZC-GEL-Upgraded-360%C2%B0Rotation-Gooseneck/dp/B09JNWYYVN

Egoless Team Structure Idea + Image (CS2113):
https://nus-cs2113-ay2324s2.github.io/website/se-book-adapted/chapters/teamwork.html