CG4002 Computer Engineering Capstone Project

“Laser Tag++”

Design Report

Group B10

Group **10**	Name	Student #	Primary Component	Secondary Component
Member #1	Ong Hee Jet	A0238981N	Hw Sensors	Comms Internal
Member #2	Chiew Yi Xiang	A0233756Y	Comms Internal	Comms External
Member #3	Ishita Mandal	A0244152N	Comms External	Sw Visualizer
Member #4	Lim Yu An	A0272173H	Sw/Hw AI	Hw Sensors
Member #5	Zeng Zheqi	A0258700H	Sw Visualizer	Sw/Hw AI

Table of Contents

Section 1 System Functionalities

In this project, we were tasked with creating an Augmented Reality Laser Tag Game. The feature list below documents the key functionalities of our system.

Responsiveness

1. AR visual effects must be displayed with low latency (100ms) upon completion of gestures.
2. Player scores update immediately when the opponent/self is down.
3. Detected moves must be sent to the evaluation server with low latency.
4. AI must be able to make inferences at frequency of at least 25Hz.

Enforcing game rules

1. Users must be able to shoot out IR lasers.
2. System must be able to detect the user's action through the use of sensors.
3. Users can logout from the game by performing a specified action.
4. Users can view their hit points and ammo displayed on their body.

Immersiveness

1. The visual effects must be able to interact with the environment (eg. projectiles colliding with walls)

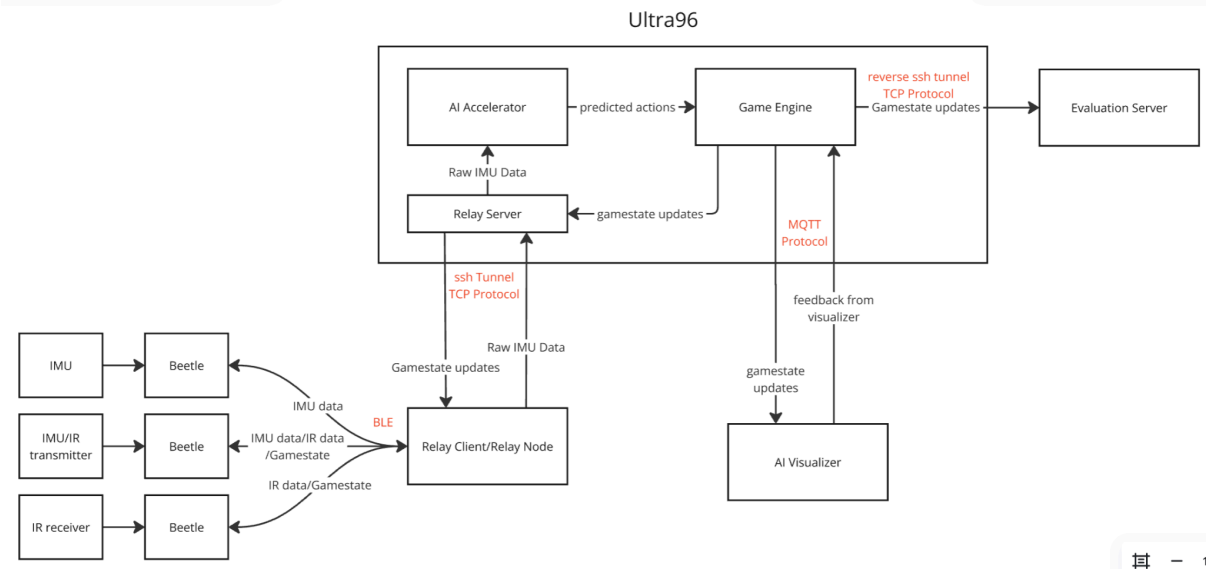
Connectivity and reliability

1. The system automatically reconnects Beetles to the relay laptop after a disconnection.
2. Reliable and secure communication between all components of the laser tag system.
3. The player's actions must be detected even with different people with slightly different ways of performing the actions.

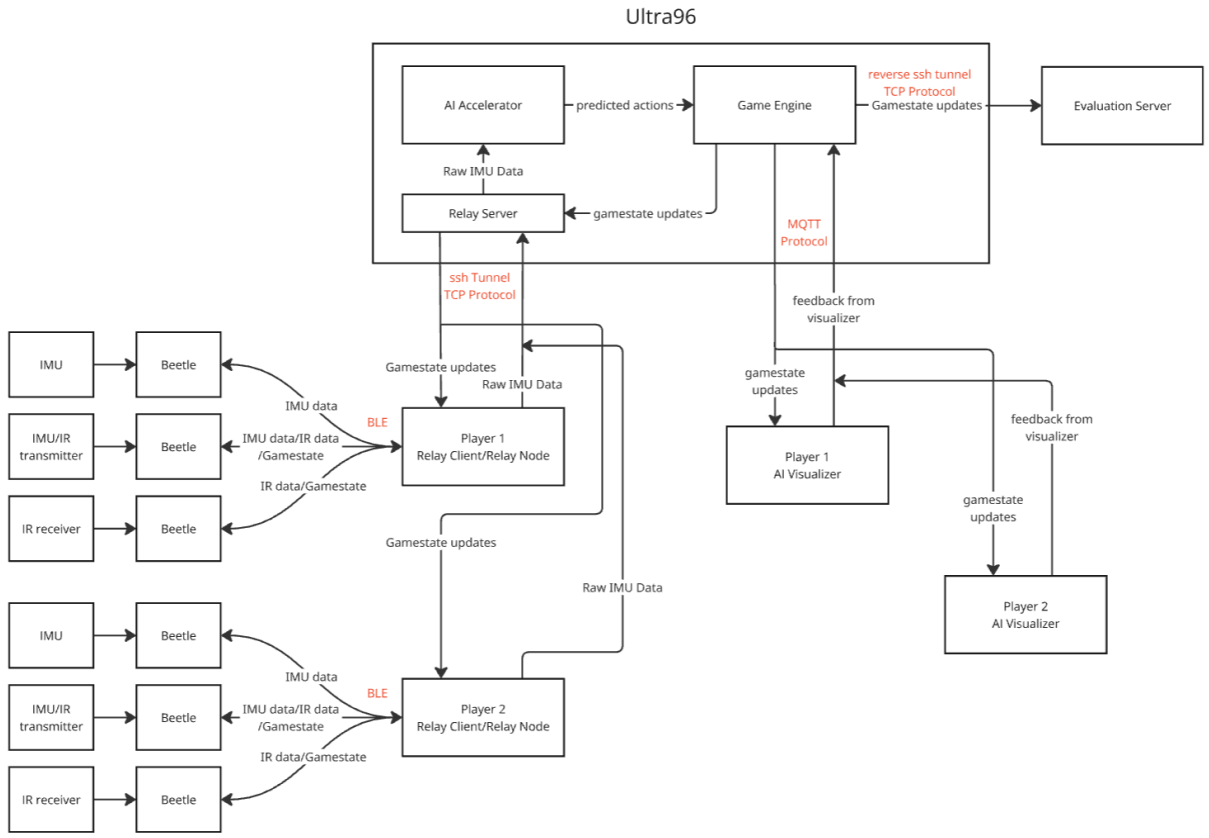
Section 2 Overall System Architecture

Section 2.1 High Level System Architecture

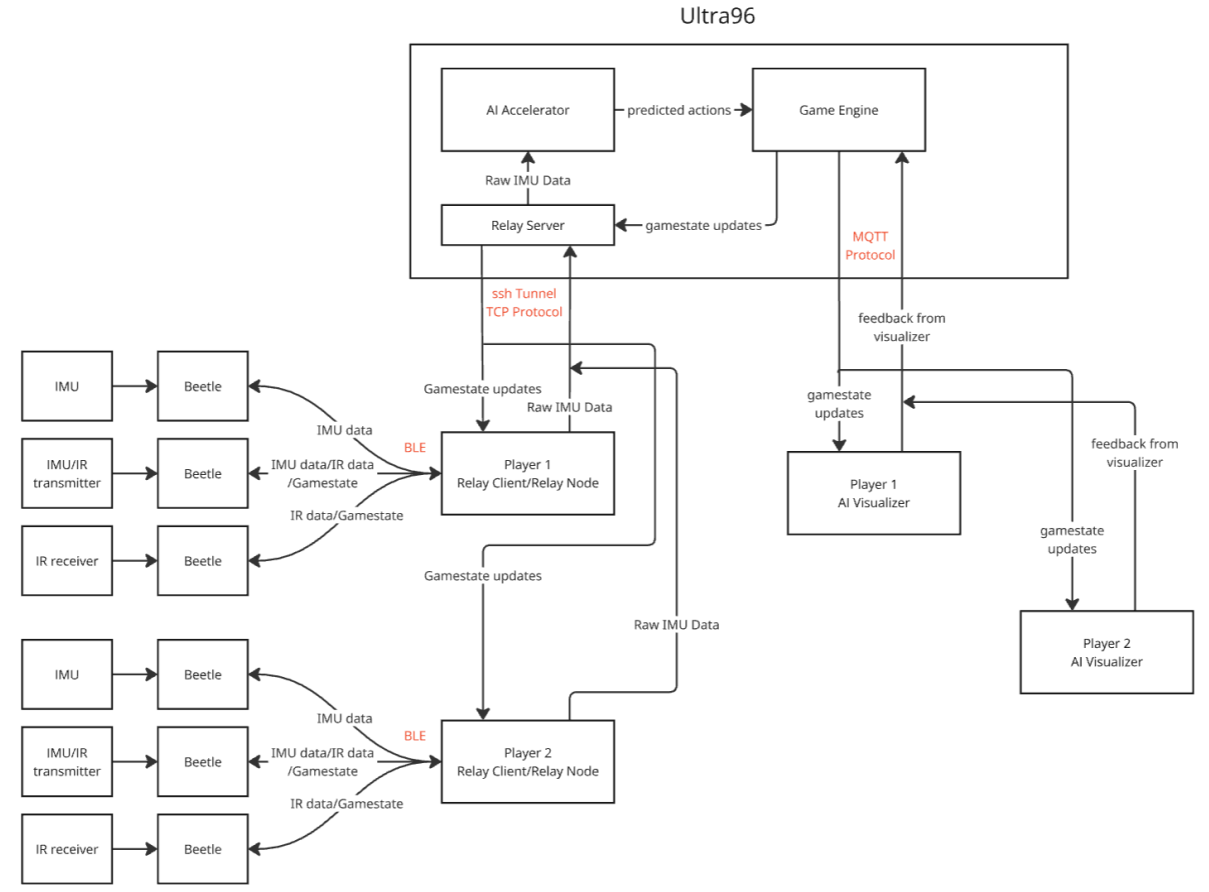
Section 2.1.1 Single Player System



Section 2.1.2 Two-Player System

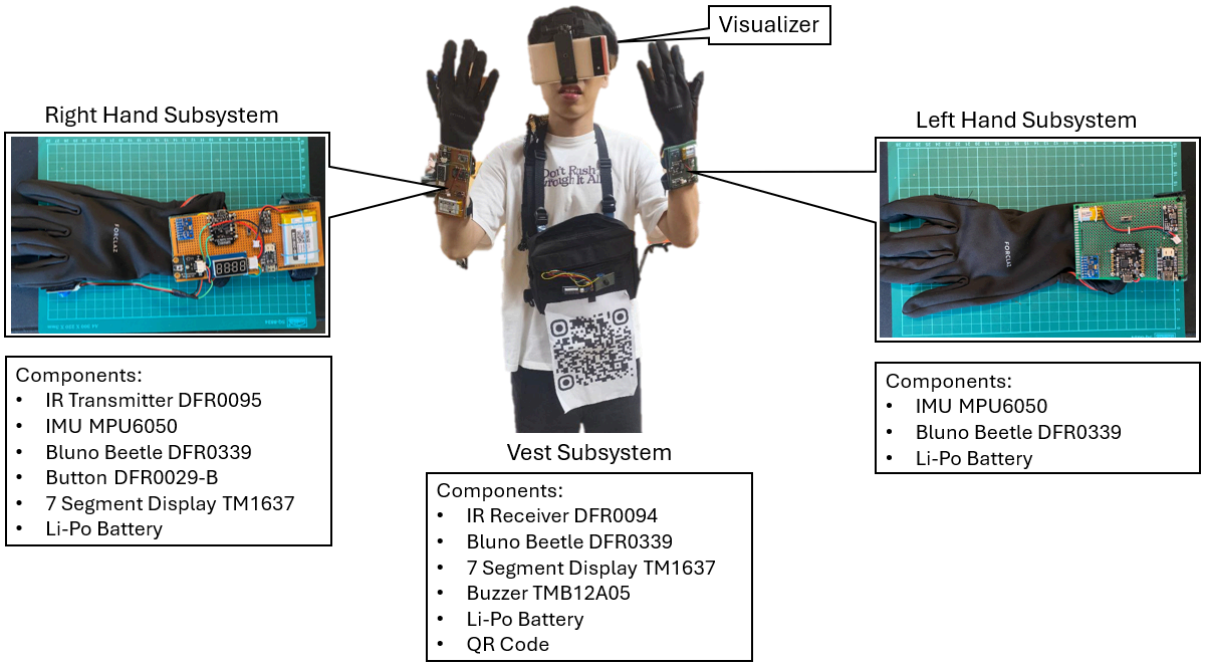


Section 2.1.3 Free-Gameplay System



Section 2.2 System's Intended Final Form

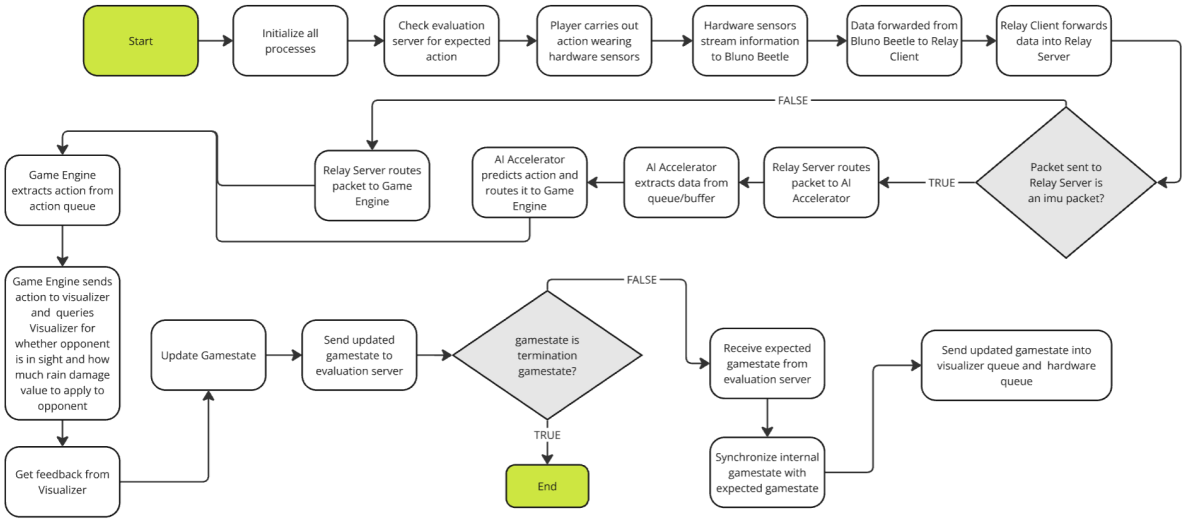
The hardware consists of 4 subsystems: Right Hand, Left Hand, Vest Subsystem and the Visualizer. The diagram below shows the placements of the hardware components.



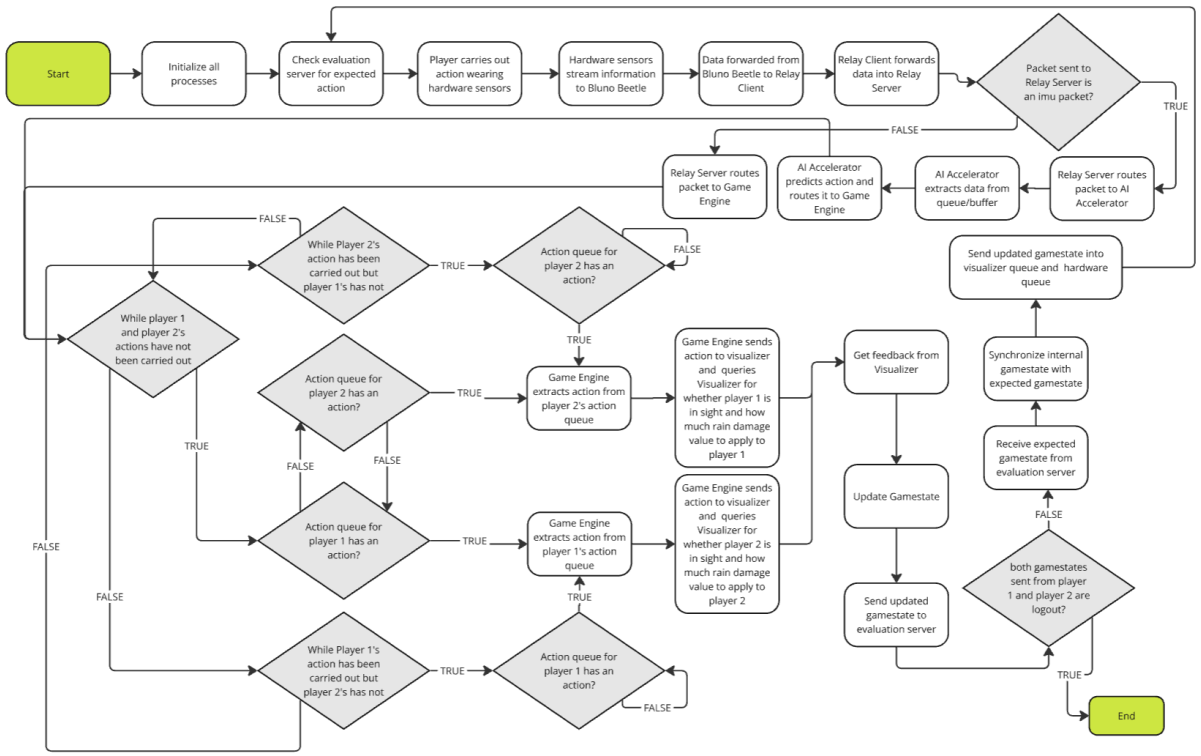
The Right Hand Subsystem consists of an IR Transmitter, IMU, Bluno Beetle, a Button and a 7 Segment Display. They will be mounted on the user's right forearm using a glove. A sponge is attached below the soldering board to provide comfort and a tighter fit for the user. We also attached straps to the sponges so that the hardware can be securely worn by the user. Similarly for the Left Hand Subsystem, the IMU, Bluno Beetle and Battery are mounted on another glove. For the Vest Subsystem, the IR Receiver, Bluno Beetle, 7 Segment Display and Buzzer are placed on a tactical chest bag. A tactical chest bag is used because of its molle mounting system [1] which makes it easier to mount the components. The bag can also be used to store the battery and Bluno Beetle. A visualizer android app is running on the smartphone.

Section 2.3 Main Algorithm

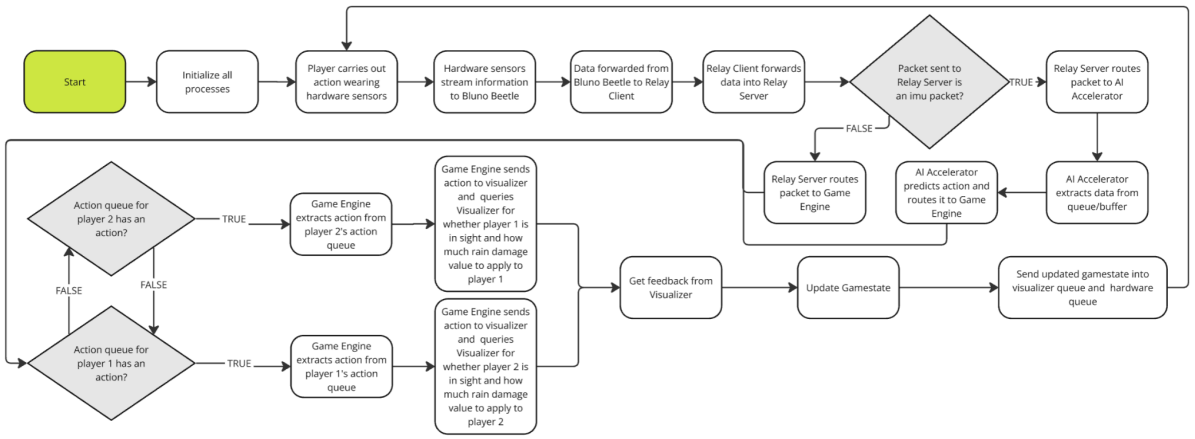
Section 2.3.1 Single Player System



Section 2.3.2 Two-Player System



Section 2.3.2 Free-Gameplay System



Section 3 Hardware Sensors

Section 3.1 Components Used

The table below shows the list of components that will be used in our design. The links to the datasheet are hyperlinked using the part number.

Component	Part Number	Quantity Used	Justification
IMU	MPU6050	4	-
Bluno Beetle	DFR0339	6	-

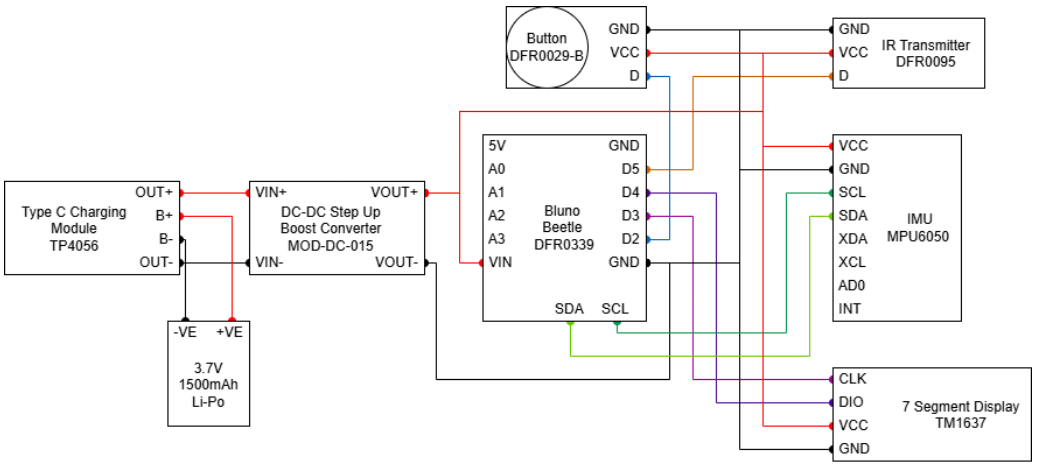
IR Transmitter	DFR0095	2	-
IR Receiver	DFR0094	2	-
7 Segment Display	TM1637	4	Able save pins on the Bluno Beetle as this 7 segment display can be controlled using 2 pins
Button	DFR0029-B	2	Easy plug and play, does not need pull down resistor
Buzzer	TMB12A05	2	Cheap and easy to use
DC-DC Step Up Boost Converter	MOD-DC-015	6	To step up the 3.7V lithium polymer battery to 5V
Type C Charging Module	TP4056	6	For easy charging of the lithium polymer battery and over discharge protection

Section 3.2 Pin Table and Schematics

After facing issues with the IMU (which will be elaborated in Section 3.5.1), the schematics for all the hardware have been changed from the initial design report. The components will now be powered by the VOUT of the DC-DC Boost Converter instead of the 5V pin of the Beetle.

Section 3.2.1 Right Hand Subsystem Pin Table and Schematics

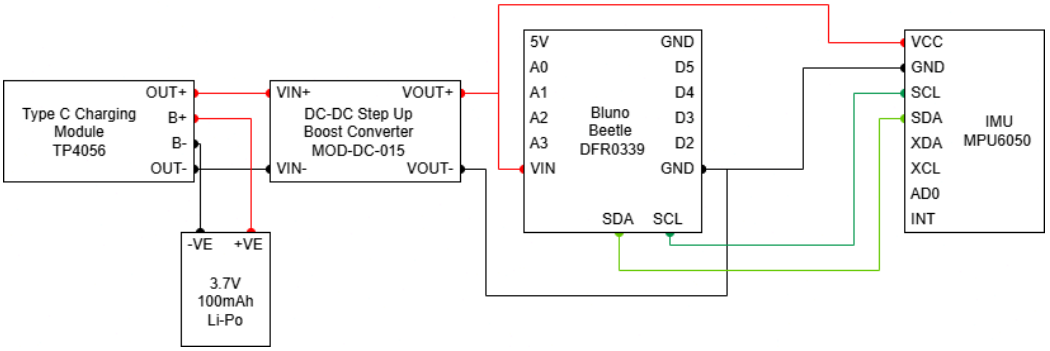
Bluno Beetle DFR0339 Pins	IMU MPU6050 Pins	Button DFR0029-B Pins	7 Segment Display TM1637 Pins	IR Transmitter DFR0095 Pins
SDA	SDA			
SCL	SCL			
D2		D		
D3			CLK	
D4			DIO	
D5				D
GND	GND	GND	GND	GND



Section 3.2.2 Left Hand Subsystem Pin Table and Schematics

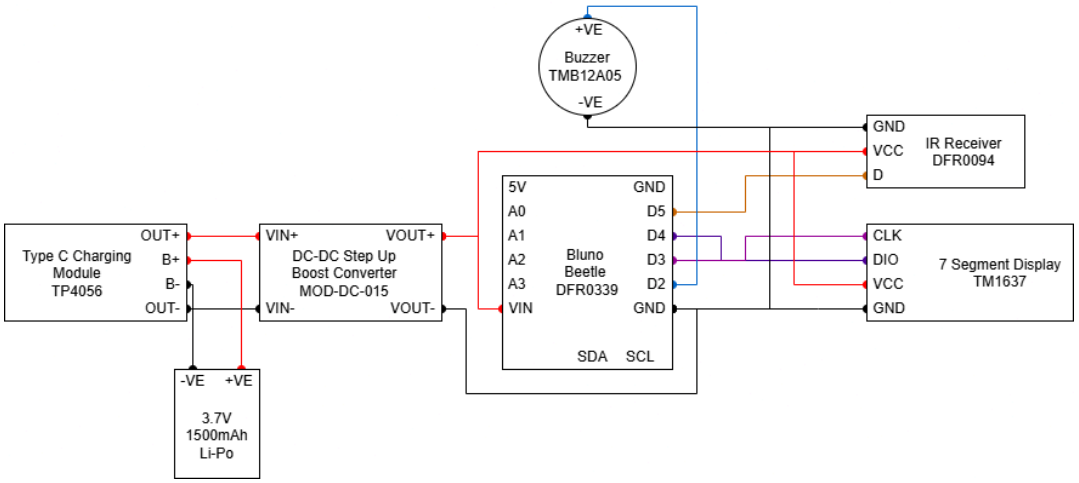
Bluno Beetle DFR0339 Pins	IMU MPU6050 Pins
SDA	SDA

SCL	SCL
GND	GND
SDA	SDA



Section 3.2.3 Vest Subsystem Pin Table and Schematics

Vest			
Bluno Beetle DFR0339 Pins	**Buzzer TMB12A05 Pins**	**7 Segment Display TM1637 Pins**	**IR Receiver DFR0094 Pins**
D2	+VE		
D3		CLK	
D4		DIO	
D5			D
GND	-VE	GND	GND



Section 3.3 Power Requirements

Section 3.3.1 Right Hand Subsystem Power Requirement

The table below lists the operating voltage and the current drawn by each component in the Right Hand subsystem.

Component	Operating Voltage (V)	Current Drawn (mA)
Bluno Beetle DFR0339	< 8	10
IMU MPU6050	2.375-3.460	3.9

Button DFR0029-B	3.3-5	-
7 Segment Display TM1637	3.3-5.5	80
IR Transmitter DFR0095	3-5	100
DC-DC Step Up Boost Converter MOD-DC-015	2.5-5	-
Type C Charging Module TP4056	4.5-5.5	1000

A 3.7V lithium polymer battery is used to power this subsystem. To facilitate easy recharging of the battery, a charging module is attached to it. To step up the voltage of the battery to power the Bluno Beetle, a DC-DC Step Up Boost Converter is used. Since the operating voltages of the Button, IR Transmitter and 7 Segment Display are 5V, they can be powered by the Bluno Beetle. While the IMU has an operating voltage of 2.375-3.460V, the in-built GY-521 3.3V voltage regulator allows the IMU to be powered by the 5V pin of the Bluno Beetle [2].

The total current drawn by the above components is $10 + 3.9 + 80 + 100 = 193.9mA$. Since they are powered by the 5V pin of the Bluno Beetle, this gives a power consumption of $193.9 * 10\text{^}(-3) * 5 = 0.9695W$. Since the DC-DC Step Up Boost Converter has a power efficiency of 86% [3], we would need $0.9695/0.86 = 1.1273W$ of power to be supplied by the battery.

We expect to spend around 4 hours each session to work on this project. Thus we would need our battery to supply $0.9695 * 4 = 4.5092Wh$ of energy so that we do not have to spend additional time changing the batteries. We decided to use a lithium polymer battery because it is lightweight and has a large capacity [4]. The lithium polymer battery is also rechargeable which would save cost in the long run. Using a 3.7V lithium polymer battery, we would need $4.5092/3.7 * 1000 = 1218.7027mA$. Therefore, a 1500mAh lithium polymer battery is chosen.

Section 3.3.2 Left Hand Subsystem Power Requirement

The table below lists the operating voltage and the current drawn by each component in the Left Hand subsystem.

Component	Operating Voltage (V)	Current Drawn (mA)
Bluno Beetle DFR0339	< 8	10
IMU MPU6050	2.375-3.460	3.9
DC-DC Step Up Boost Converter MOD-DC-015	2.5-5	-
Type C Charging Module TP4056	4.5-5.5	1000

The battery set up and IMU connections are the same as the Right Hand Subsystem.

The total current drawn by the above components is $10 + 3.9 = 13.9mA$. Since they are powered by the 5V pin of the Bluno Beetle, this gives a power consumption of $13.9 * 10\text{^}(-3) * 5 = 0.0695W$. Factoring in the power efficiency of the DC-DC Step Up Boost Converter, we would need $0.0695/0.86 = 0.0808W$ of power to be supplied by the battery.

With the same 4 hours working time, we would need $0.0808 * 4 = 0.3233Wh$ of energy. Although the power requirement is small for this subsystem, we decided to stick with a lithium polymer battery since it is rechargeable to save cost in the long run. Using a 3.7V lithium polymer battery, we would need $0.3233/3.7 * 1000 = 87.3784mA$. Therefore, a 100mAh lithium polymer battery is chosen.

Section 3.3.3 Vest Subsystem Power Requirement

The table below lists the operating voltage and the current drawn by each component in the Vest subsystem.

Component	Operating Voltage (V)	Current Drawn (mA)
Bluno Beetle DFR0339	< 8V	10
IMU MPU6050	2.375-3.460V	3.9

7 Segment Display TM1637	3.3-5.5V	80
IR Receiver DFR0094	5V	100
Buzzer TMB12A05	4-7	30
DC-DC Step Up Boost Converter MOD-DC-015	2.5-5	-
Type C Charging Module TP4056	4.5-5.5	1000

The battery set up and IR connections are the same as the Right Hand Subsystem. The Buzzer has an operating voltage of 4-7V, allowing it to be powered by the Bluno Beetle.

The total current drawn by the above components is $10 + 3.9 + 80 + 100 + 30 = 223.9mA$. Since they are powered by the 5V pin of the Bluno Beetle, this gives a power consumption of $223.9 * 10\text{^}(-3) * 5 = 1.1195W$. Accounting for the power efficiency of the DC-DC Step Up Boost Converter, we would need $1.1195/0.86 = 1.3017W$ of power to be supplied by the battery.

With the same 4 hours working time, we would need $1.3017 * 4 = 5.2068Wh$ of energy. Using a 3.7V lithium polymer battery, we would need $5.2068/3.7 * 1000 = 11407.2432mA$. Therefore, a 1500mAh lithium polymer battery is chosen.

Section 3.4 Libraries Used and Feedback System

Section 3.4.1 IMU MPU6050 Library

The MPU6050 Library [5] by Electronic Cats will be used to interface with the MPU6050. This library is chosen because it makes using the DMP of the MPU6050 easier. This library is supported by the I2Cdevlib [6] which handles the I2C communication between the Bluno Beetle and the MPU6050.

We plan to utilize the DMP's internal FIFO buffer to keep track of the MPU6050's accelerometer and gyroscope data [7] to easily attain the Yaw, Pitch and Roll values which will be used as inputs for our AI model. A sample code snippet is shown below. This tutorial by mjwhite8119 [7] has been referenced when understanding the MPU6050.

```
if (mpu.dmpGetCurrentFIFOPacket(FIFOBuffer)) {
  mpu.dmpGetQuaternion(&q, FIFOBuffer);
  mpu.dmpGetGravity(&gravity, &q);
  mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

  // Yaw, Pitch, Roll in degrees
  yaw = ypr[0] * 180/M_PI;
  pitch = ypr[1] * 180/M_PI;
  roll = ypr[2] * 180/M_PI;
}
```

Section 3.4.2 IR Transmitter and Receiver Library

The Arduino IR-Remote Library [8] is used to send and receive IR packets. The NEC protocol will be used. Since the NEC protocol has a 8-bit address and 8-bit command [9], we decided to use the least significant 4 bits of the address field as the player ID. The command field will be the player ID concatenated with the general "shoot" 4-bit command ID (0x08). This is to prevent friendly fire and disruption from other group's IR signals. Player 1 will have an ID of 0x1 and Player 2 will have an ID of 0x2. A sample code snippet of how the library will be used is shown below. This tutorial by Drone Bot Workshop [10] has been referenced when understanding the Arduino IR-Remote Library.

```
// Player 1's Bluno Beetle connected to the IR Transmitter
IrSender.sendNEC(0x01, 0x18, 1);

// Player 2's Bluno Beetle connected to the IR Receiver
if (IrReceiver.decodedIRData.address == 0x01 &&
    IrReceiver.decodedIRData.command == 0x18) {
  // Register Player 2 as shot.
}
```

Section 3.4.3 TM1637 7 Segment Display Library

The TM1637 library [11] is used to control the TM1637 7 Segment Display. We will primarily be using the showNumberDec() function to show integers on the 7 Segment Display. A sample code snippet can be shown below.

```
// Displays _ _ _ 5 on the 7 Segment Display
display.showNumberDec(5, false);
```

Section 3.4.4 Feedback System

The 7 Segment Display and Buzzer forms part of the feedback system for the user. The 7 Segment display will be used to display the amount of ammunition and health the user has left. The Buzzer that is connected to the Vest Subsystem will sound whenever the user has been hit.

Section 3.5 Issues Faced

Section 3.5.1 Corrupted IMU Data

During collection of training data, we noticed that the Left Hand Subsystem for the 2nd set was constantly sending 0s. We then connected the Beetle to our laptops to check the output on the serial monitor. It was then we realized that the IMU was sending corrupted data. We then used a voltmeter to measure the voltage of the VCC pin of the IMU and realised that the input voltage at the VCC pin was 4.3V, which was significantly lower than the operating voltage of the IMU. We concluded that the reason the IMU data was corrupted was because the output voltage of the 5V pin of the Beetle was significantly lesser than 5V causing the IMU to not function as intended.

Upon further reflection, we realised that powering the entire system using the Bluno's 5V pin was not ideal since the output voltage was not reliable due to manufacturing defects. As such, we decided to change the wiring of all the hardware to use the VOUT pin of the DC-DC Boost Converter to power the IMU, 7 Segment Display, IR Receiver/ Transmitter and Buzzer. Using a voltmeter, we have verified that the VOUT pin of the DC-DC Boost Converter provides a steady 5V which will power our system much more reliably.

Section 3.5.2 Bluetooth Disconnections

During collection of training data, we noticed that the Right Hand Subsystem for the 2nd set was constantly disconnecting from the relay laptop. While using a voltmeter, we verified that the input voltage to the VIN pin of the Beetle was indeed 5V. Since the same code for bluetooth connection was used for all Beetles and only this particular Beetle was facing issues with connectivity, we concluded that the problem could lie in the bluetooth module in the Beetle.

To resolve this issue, we decided to change the Beetle. After changing the Beetle, we did not encounter any connectivity issues with that set.

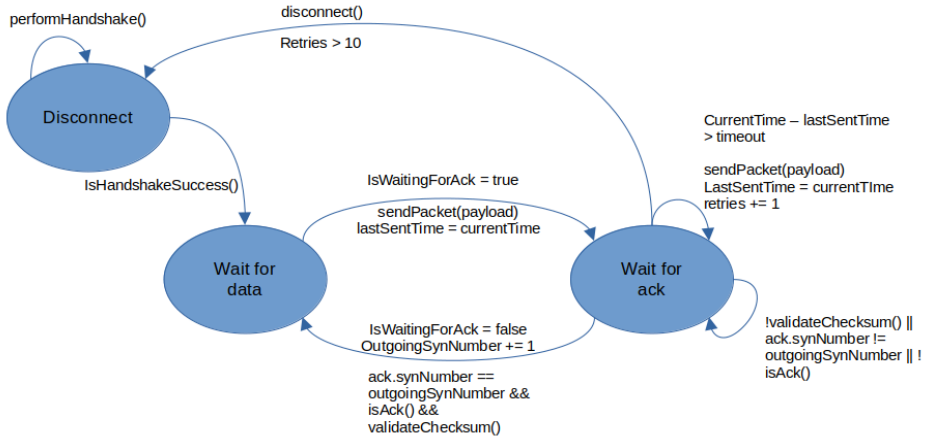
Section 4 Internal Communications

Section 4.1 Task Management

Section 4.1.1 Beetle

Tasks on the Beetles will be handled as follows:

1. Read Sensor Data from IMU, IR, etc.:
    - The Beetle continuously collects raw data from sensors.
2. Process Sensor Data into Packets:
    - Once the raw data is available, it is processed into packets with necessary headers, checksums, and sequence numbers.
3. Transmit Data Between Beetle and Laptop Through BLE:
    - The processed packet is transmitted over BLE.
    - Data transfer will be done using streamed by the Beetle to the laptop, which allows for real time transfer of data as compared to periodic pull by the laptop, which might result in lost or late data.
    - Depending on the communication protocol:
        - For **stop-and-wait**: The Beetle waits for an acknowledgment from the laptop before sending the next packet.
        - For **UDP-like**: The Beetle sends the data without waiting for an acknowledgment.
4. Process packets from Laptop
    - Process and parse packets from Beetle such as ACK and game state packets
    - ACK packets do not require further action, whereas game state packets will need to update necessary variables such as health and ammo of player
5. Three-Way Handshake (if required):
    - This step is part of connection initialization and **does not happen repeatedly** during normal operation. The handshake occurs once when establishing or reestablishing the BLE connection.



FSM for sender

Section 4.1.2 Relay laptop

Each Beetle is assigned a dedicated thread on the relay laptop to handle the communication independently which allows the relay laptop to manage multiple beetles without blocking
The responsibilities include:

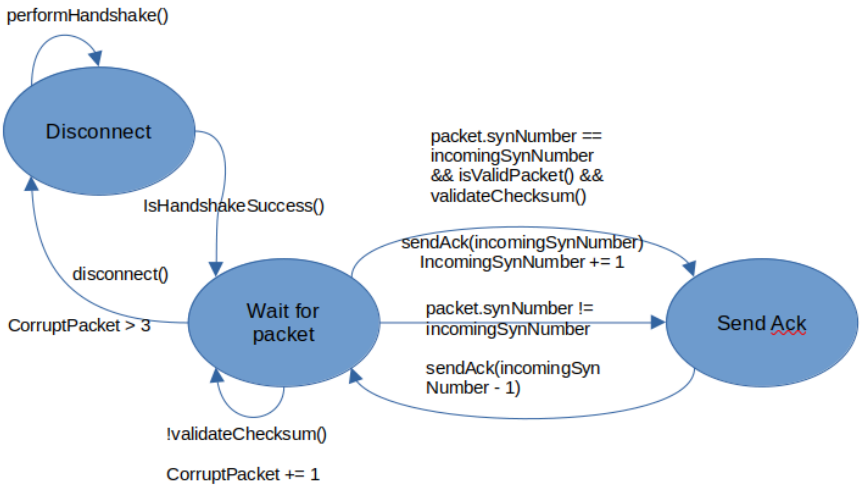
- Establishing a BLE connection between the laptop and Beetle
- 3 way handshake between laptop and Beetle, initiated by the laptop
- Sending and receiving data packets (handled by the BeetleDelegate)
    - Thread continuously listens for data packets from the Beetle via BLE notifications
    - Once the packets/fragments are added to the receiving buffer by the BeetleDelegate, the BeetleThread will proceed to validate and process 20 bytes of the buffer as a packet.
- Valid data received from Beetle is put into a shared deque
    - enables other threads or process to access the data asynchronously

The BeetleDelegate class will extend the DefaultDelegate [12] class provided by bluepy to handle notifications from a connected peripheral object (Beetle)
The responsibilities include:

- Handling notifications
    - when the beetle sends a BLE notification, the handleNotification method is triggered
    - Packets are then placed in a receiving buffer for parsing and validation by the BeetleThread

There will be a relay client thread on the laptop that is responsible for sending and receiving data between the laptop and Ultra96 when there is data in the shared queue/buffer.
Responsibilities include:

- Monitoring the shared outgoing queue for new data
- Forwarding the data to Ultra96
- Monitoring the incoming queue from Ultra96 for gamestate updates
- Process the gamestate updates and format them as packets
- Forward the gamestate packets to the relevant Beetles



FSM for receiver

Section 4.2 BLE Interfaces

Bluetooth Low Energy protocol will be used to communicate between the Beetles and the relay laptop. Libraries used will be bluepy, which is an interface to allow access to BLE devices from Python. Since the library uses bluez, which is native to linux, development will be done on Ubuntu.

Setup of Beetle for communication will be as follows: [13]

1. Connect Beetle to laptop using a physical cable and select serial port
2. Open the serial monitor and select “No Line Ending” and “115200 baud” from the dropdown list.
3. Send ‘+++” message
4. ‘Enter AT Mode’ will be received if enter command mode successfully
5. Select “Both NL & CR” and “115200 baud” from the dropdown lists
6. Send the AT command (AT+SETTING=DEFAULT)
7. If the BLE is successfully configured, "OK" message will be received
8. Use “AT+EXIT” to exit AT Mode.

Section 4.3 Communication Protocol

Protocol used to communicate between the Beetles and laptop will be Stop-and-wait.

1. The Beetle sends a packet to the laptop.
2. The laptop sends an ACK packet on successful receipt
3. If no ACK is received within a timeout period, the Beetle retransmits the packet

For time sensitive data, we can explore using a variation of UDP, where no ACK is required after the data is sent.

1. Beetle sends packets without waiting for ACK from the laptop
2. Lost packets are ignored to prioritise timely delivery of data

Handshake process will be a three way handshake between each beetle and the laptop before data transmission. Laptop will send a SYN packet to beetle, and beetle will respond with an ACK packet, in which the laptop will return an ACK packet.

Section 4.3.1 Baud rates

Higher baud rate would mean faster data transmission which is needed for quicker communications between the Beetle and the relay laptop, which is why 115200 Hz is chosen, the maximum baud rate of the Beetle.

Section 4.3.2 Packet types

Packet type	Identifier	Description
SYN	'S'	SYN packet is sent to initialise the handshake process between the 2 devices
ACK	'A'	ACK packet is sent to acknowledge that packet is received
IMU	'M'	IMU packet contains motion data from the IMU
IR transmitter	'T'	'T' packet contains data from the IR transmitter
IR receiver	'R'	'R' packet contains data from the IR receiver
Game state	'G'	'G' packet contains game specific parameters such as health and ammo
~~'keep-alive'~~	~~'K'~~	~~'keep-alive' packet from Beetle to laptop to show connection status~~

Section 4.3.3 Packet format

First iteration

Packet type 1 byte	Device ID 1 byte	Sequence Number 1 byte	Data 16 bytes	Checksum 1 byte
header			body	footer

Second iteration

Packet metadata 1 byte	Packet type 1 byte	Sequence Number 1 byte	Data 16 bytes	Checksum 1 byte
header			body	footer

Final packet format removed Device ID and replaced it with packet metadata. This is to identify the start of a packet sent from the Beetle, in order to better deal with fragmentation. The packet/fragment will only be added to the receiving buffer if the packet/fragment starts with the Packet Metadata byte of 'P' or if the receiving buffer already has data inside.

Section 4.3.4 Devices

Device type	Device ID
Player 1 vest	1
Player 1 right hand	2

Player 1 left hand	3
Player 2 vest	4
Player 2 right hand	5
Player 2 left hand	6

Section 4.4 Reliability

To handle error detection and packet loss:

- checksum validation will be implemented, to validate packet integrity
- sender will wait for ACK and retransmit packet if there is no ACK received before timeout

To handle packet fragmentation:

First iteration:

- Since our packet size is fixed to 20 bytes, in the event when the received packet is smaller than our fixed size, the relay node will attempt to reassemble the fragmented packets before sending it to the ultra96

Second iteration:

- Fragmentation handling improved significantly after we introduced a metadata byte to each packet to indicate the start of a valid packet.
- With this enhancement, a fragment is only added to the receiving buffer under two conditions:
  - It contains the metadata byte, signaling it is the start of a new packet.
  - There is already existing data in the buffer, implying that the fragment is a continuation of an ongoing packet.
- This logic allows us to be more selective with the fragments we process. Instead of naively clearing the buffer whenever invalid or fragmented data is received, we can now intelligently drop unwanted fragments while preserving potentially valid partial data. This results in more robust and reliable handling of fragmented packets, especially over unstable Bluetooth connections.

To handle connection drops:

First iteration:

- Since start of action has been offloaded to hardware, there will be periods when there are no actions performed by the user, but the connection between the Beetle and the laptop must persist. This can be mitigated using ‘keep-alive’ packets that are sent periodically.
- When there is a connection drop, there will be no ‘keep-alive’ packets received by the laptop, and the laptop will attempt to reconnect to the Beetle through the 3-way handshake.

Second iteration:

- Beetle is constantly streaming IMU data to the relay laptop, hence there is no need for ‘keep-alive’ packets in the case for device 2, 3, 5, and 6.
- The relay laptop is able to detect a drop in connection between the relay laptop and the Beetle since an exception is thrown whenever the Beetle disconnects. There is no need for ‘keep-alive’ packets as long as we catch the exception and reconnect the Beetle to the relay laptop and perform handshake again if needed.

Section 4.5 Issues faced

1) Synchronising IMU data coming from 2 different devices

To address this, we maintain two separate deques—one for each hand. The relay client only processes and pops data from these deques when both have at least one entry. This ensures that data from the left and right hands are aligned and combined into a single data point before forwarding it to the Ultra96. If data from one hand is missing, default values are used to maintain consistent data formatting.

2) Running 7 threads in a single process to handle data from 2 players.

Due to Python’s Global Interpreter Lock (GIL), only one thread can execute Python bytecode at any given time. As a result, threads are not truly parallel for CPU-bound tasks. This led to performance bottlenecks and increased context switching overhead. To address this, I experimented with running each player in a separate

process. This allowed us to bypass the GIL, as each process has its own Python interpreter and GIL. Now, each process only needed to manage 4 threads, reducing the load and overhead in each.

However, this setup introduced significant connectivity issues with the Beetles—likely due to increased competition for Bluetooth resources or instability in handling concurrent BLE connections across processes. Eventually, we resolved the issue by distributing the workload across two separate laptops, each running its own relay client to handle one player's devices. This eliminated the connectivity problems and improved system stability.

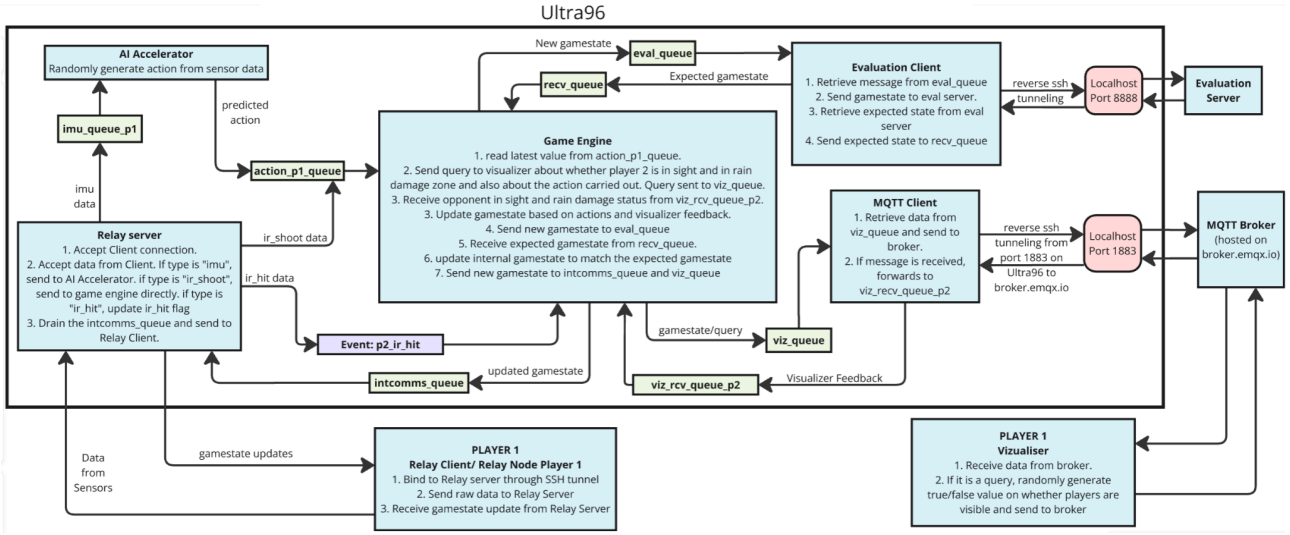
3) Frequent disconnections between Beetle and relay laptop

We observed that one of the Beetles experienced frequent disconnections from the relay laptop. To mitigate the delay caused by repeated handshakes during reconnection, we introduced a 5-second threshold. If a Beetle disconnects and reconnects within 5 seconds, the system skips the handshake process. This optimization significantly reduces reconnection time and improves overall communication stability.

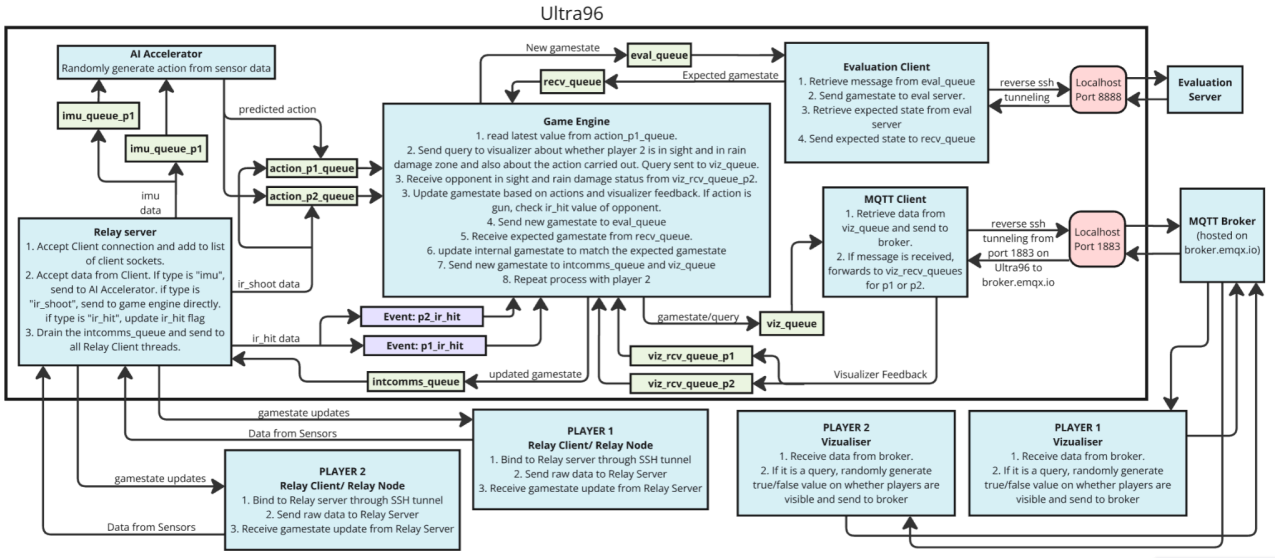
Section 5 External Communications

Section 5.1 High Level Architecture Overview

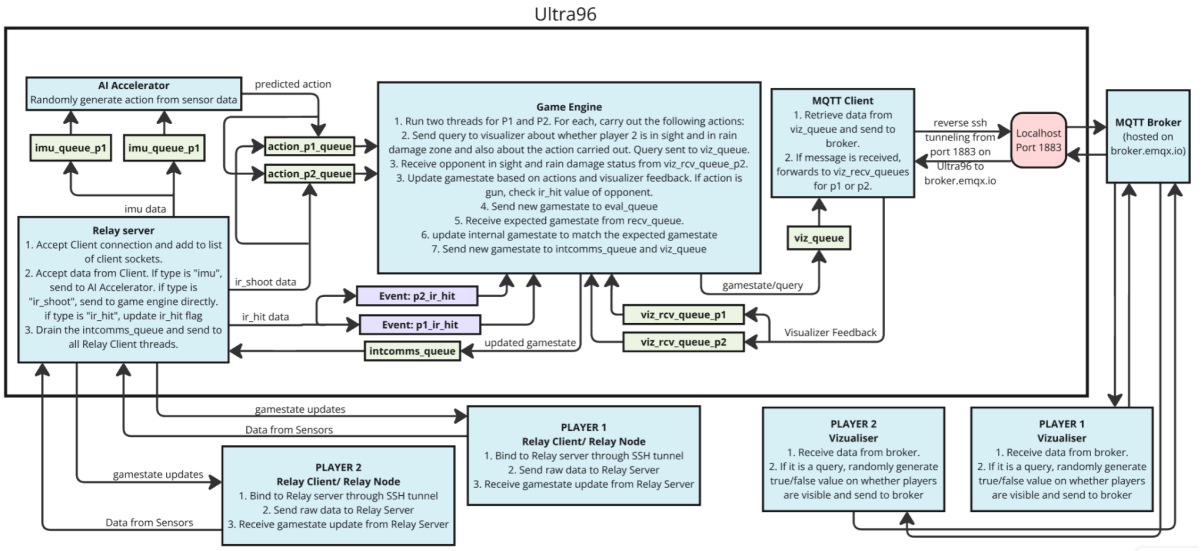
Section 5.1.1 Single Player System



Section 5.1.2 Two-Player System

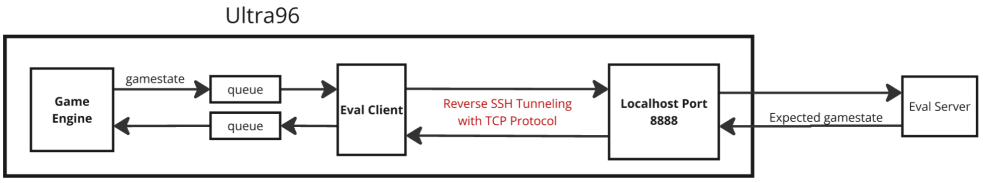


Section 5.1.3 Free-Gameplay System



Section 5.2 Communication between Ultra96 and Evaluation Server

The Ultra96 runs an Evaluation Client that opens a TCP socket to the Evaluation Server at the configured IP/port. The Evaluation Client sends the gamestate to the Evaluation Server and receives the expected next state.



Section 5.2.1 Message Format

All messages are length-prefixed and encrypted.

<length>_<encrypted_message>

Payload structure (before encryption)

encrypted_message = {
player: <Player_number>
action: <player_action>
gamestate: <updated_gamestate>
}

Section 5.2.2 Secure Communication

Data is encrypted with AES-128 CBC. A fixed 16-byte IV adds diffusion, and a predefined 16-byte key is shared by both ends for encryption/decryption.

Section 5.2.3 Tunneling

Because the SoC firewall blocks outbound traffic, the board opens a reverse-SSH tunnel that maps localhost:8888 on the Ultra96 to the Evaluation Server. The Evaluation Client therefore connects only to 127.0.0.1:8888; SSH forwards those packets to the remote server.

Section 5.2.4 Library APIs

- **socket module:** To establish and manage the TCP connection.
- **pycryptodome library:** For AES encryption and decryption.
- **json module:** For serializing and deserializing message payloads.

Section 5.2.5 Robustness and Error Handling

To handle unexpected errors and ensure reliability, the following design considerations were used:

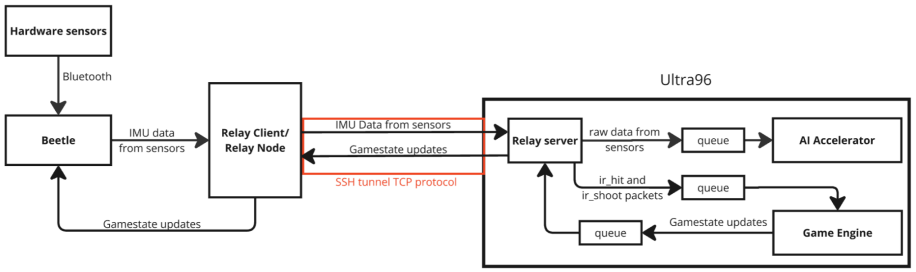
- **Connection Failure**: If the client fails to connect to the server, an appropriate error message is logged, and the connection attempt is retried.
- **Queue Management:** Incoming and outgoing messages are managed using Python's multiprocessing Queue to ensure thread-safe communication between processes.
- **Timeouts and Recovery**: If the system detects dropped connections,a ConnectionError is raised and unwinds the loop, ensuring sockets close and preventing resource leaks.

Section 5.2.6 Evolution of Evaluation Client with different systems

The same client code supports both Single-Player and Two-Player modes. In Free-Gameplay mode the Evaluation Server is not needed, so the Evaluation Client is omitted entirely.

Section 5.3 Communication between Ultra96 and Relay Client

The Relay Client (running on an external laptop) receives raw IMU data from the Beetles over Bluetooth and forwards it to the Relay Server hosted on the Ultra96. Connection is made through a reverse-SSH tunnel that carries ordinary TCP traffic. Game-state updates generated by the Game Engine travel back along the same tunnel and are then relayed to the Beetles.



Section 5.3.1 Relay Client/Relay Node (External)

The Relay Client has four main functions:

1. **Bluetooth ingestion**: Collect raw IMU packets from the Beetles and Base64-encode each payload.
2. **Tunnel setup:** Open an SSH tunnel to the Ultra96 (remote bind address localhost:11000). SSH exposes that remote port on a free local port, say 127.0.0.1:<local_port>.
3. **TCP forwarder**: Create a TCP socket to 127.0.0.1:<local_port>. Each Base64 payload is wrapped in the length-prefix format and sent to the Relay Server. <length>_<payload>
4. **Return path:** Game-state replies travel back over the tunnel, are decoded by the client, and forwarded to the Beetles via Bluetooth.

Section 5.3.2 Relay Server (Ultra96)

The Relay Server has three main functions:

Receive & decode

Receive packets from the Relay Client, read the <length>_ prefix, then read exactly that many bytes of payload. Then deserialize the JSON payload into a Python dict.

Route by message type

Type	Action
imu	Enqueue sensorData on imu_queue_p1 or imu_queue_p2 (based on playerID and num_player) for the AI accelerator.
ir_tx	Put "gun" on action_p1_queue or action_p2_queue (shoot event).
ir_rx	Set p1_ir_hit or p2_ir_hit Event flag (hit event) so the Game Engine deducts health.

Broadcast game-state updates

Pull each update from the int_comms_queue (a dict with p1/p2 sub-fields), serialize this JSON and send it—again with <length>_ framing to every connected client socket.

Section 5.3.3 Message Format

Format of Message Received from the Relay Client:

{
“type” : <Either imu/ir_tx/ir_rx>
“playerID” : <1 or 2>
“sensorData”: <optional value only sent if type is ‘imu’>
}

Format of Message sent to Relay Client

{
  "type": "game_state",
  "p1": { "ammo": …, "health": … },
  "p2": { "ammo": …, "health": … }
}

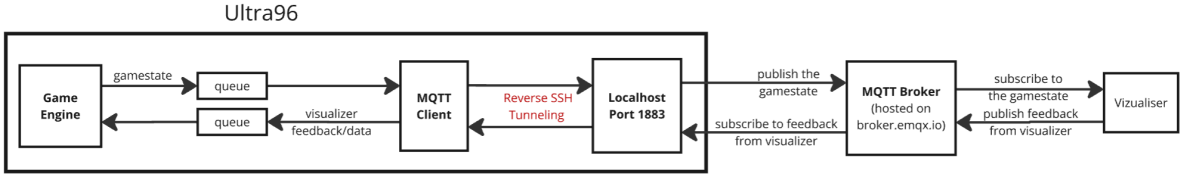
Section 5.3.4 Evolution of Relay Client with different systems

For Single Player, the Relay Client is a single sub-process of the main process which accepts an inbound client,receives its messages, and drains int_comms_queue to send gamestate updates to it.

However, for two-players and free-gameplay, even though the Relay Server is a single sub-process, there are multiple laptops that connect to it. If the same process as single player is used, then draining the int_comms_queue will only send gamestate data to whichever client connects to it first, causing a race condition. In order to overcome that, I introduced threads. Each client is wrapped in its own separate thread and has a private queue. Additionally, every time a client socket connects to the server, it is added to a list with its own private queue and client_handler. Now, the server fans out every game-state update from intcomms_queue to all per-client queues, preventing race conditions.

Section 5.4 Communication between Ultra96 and Visualizer

The communication between the Ultra96 board and the Visualizer is implemented using the MQTT (Message Queuing Telemetry Transport) protocol because the protocol is lightweight, supports pub/sub, and incurs very little latency.



Section 5.4.1 MQTT Broker and Tunneling

The MQTT broker acts as the intermediary that routes messages between the Ultra96 board (publisher/subscriber) and the Visualizer (subscriber/publisher). I decided to host the broker externally on the EMQX cloud instead of locally on the Ultra96 board. Because the SoC firewall blocks outbound traffic, I opened a reverse-SSH tunnel that maps localhost:1883 on the Ultra96 to broker.emqx.io:1883 on the cloud. Hosting it on the cloud increased its availability and scalability and made it simpler to implement.

Section 5.4.2 MQTT Client

The MQTT Client is hosted locally on the Ultra96 board. It performs dual roles:

Role	Details
Publisher	Pops (msg_type, data) tuples from viz_queue and publishes JSON: {"type": msg_type, "data": data} to topic cg4002grpB10/ultra96.
Subscriber	Subscribes to cg4002grpB10/visualizer. On each message: ▪ JSON-decode▪ If id == 1 → enqueue to viz_rcv_queue_p2 ▪ Else → enqueue to viz_rcv_queue_p1

Section 5.4.3 Message Format

Messages exchanged between the Ultra96 board and Visualizer are structured to ensure efficient and consistent communication.

1. **Outgoing Messages (Publisher → Broker → Subscriber):** Outgoing messages are of two types:
   a. {
   “type”: “QUERY_DETECTED_STATE”
   “data”: {
   “playerID” : <1 or 2>
   “player_action”: <Action>
   }
   }
   b. {
   “type”: “STATE”
   “data”: <Gamestate>
   }

If the type is “STATE”, then it is a gamestate update. if the type is “QUERY_DETECTED_STATE”, then it is a query which asks for the visibility and rain damage status of “playerID” and the action carried out by the opponent of “playerID”.

2. **Incoming Messages (Subscriber → Broker → Publisher):**

   a. {
   “id”: <1 or 2>
   “player1_in_sight”: <True/False>
   “player2_in_sight”: <True/False>
   “p1_in_zone”: <Integer positive value>
   “p2_in_zone”: <Integer positive value>
   }

If the player id is 1, then the MQTT Client places the message in viz_recv_queue_p1. Otherwise it places it in viz_recv_queue_p2. The Game Engine later dequeues these values to decide visibility and rain-damage effects. Player_in_sight is a boolean value implying if the player with the “id” value is visible or not. The p1_in_zone and p2_in_zone variables are integers indicating rain damage values. So for example, if the value is 0, then it implies no rain_damage. if it is 2, it means the player is in two bomb zones and the health points for rain damage need to be deducted twice.

Section 5.4.4 Library APIs Used

- **paho-mqtt:** MQTT protocol implementation in python.
- **json:** Serialization and deserialization of message payloads.
- **multiprocessing:** Provides process-based parallelism.

Section 5.4.5 Error Handling and Robustness

- Connection Failures: The MQTT Client attempts to reconnect to the broker if the connection is lost. A loop is used to retry until the connection is successfully re-established.
- Exception Handling: loop_start() / loop_stop() are paired in a try / finally block to guarantee the network loop terminates even on exceptions.

Section 5.4.6 Evolution of MQTT Client with different systems

As the system evolved from single player to two-player/free-gameplay, the MQTT Client logic itself did not change. Only the message format for the messages received from the visualizer changed to include "player1_in_sight" and "p1_in_zone" because now player 2 would carry out actions as well. Consequently, viz_recv_queue_p1 was defined to enqueue the above values and pass to the Game Engine.

Section 5.5 Inter-Process Communication (IPC) and Concurrency Management

All the components communicate with each other using queues which is a thread safe data structure for multiprocessing. These include:

- **imu_queue_p1 and imu_queue_p2:** These queues store imu data transferred from the Relay Server to the AI Accelerator for further processing and action prediction.
- **action_p1_queue and action_p2_queue**: These queues transfer information either from the Relay Server (shoot packets) or the AI Accelerator (predicted actions) to the Game Engine.
- **viz_queue:** This queue transfers information (query on detected state, gamestate updates) from the Game Engine to the MQTT Client.
- **viz_rcv_queue_p1 and viz_rcv_queue_p2:** These queues transfer visualizer feedback from the MQTT Client to the Game Engine.
- **eval_queue:** This queue transfer gamestate information from the Game Engine to the Evaluation Client.
- **recv_queue:** This queue transfers the expected gamestate from the evaluation client to the Game Engine.
- **intcomms_queue:** This queue transfers gamestate information from the Game Engine to the Relay server which is further passed on to the Relay Client.

Some of the communications are handled using Event flags. These include:

- **p1_ir_hit and p2_ir_hit:** These events are updated whenever the Relay Server receives an ir_hit packet. The Game Engine retrieves the value of these flags when carrying out an action and reduces opponent point accordingly. One an action has been carried out, the Game Engine resets the flag values.

The reason I decided to use queues for inter-process communication is because it ensures modularity. Each communication channel is handled independently, making the system more modular and easier to maintain. also made it easier to scale individual communication processes as well as handle failures without affecting the Game Engine. With this method, dedicated communication processes can run in parallel with the Game Engine, reducing bottlenecks. Each communication process can also be optimized for its specific protocol (e.g., MQTT, sockets). Additionally, Queue.get() can block or timeout, letting each module choose push-based or poll-based flow control without busy-waiting.

Section 5.6 Overall evolution of External Comms

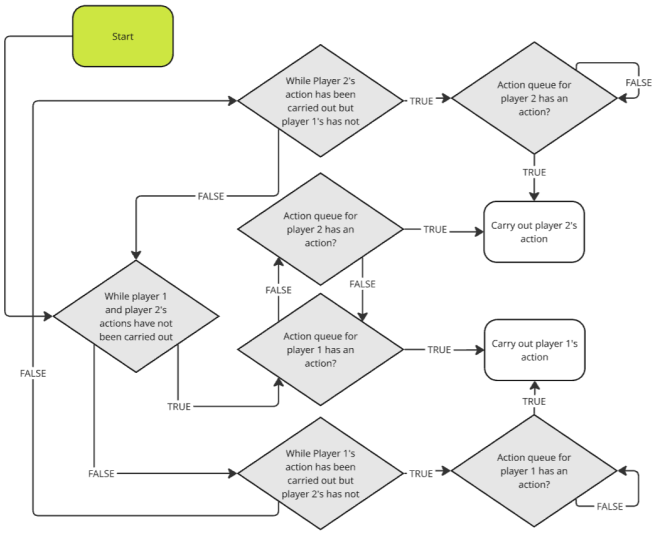
Section 5.6.1 General Issues

Over the course of the project, the external communication system faced a number of issues either due to the updates made to the SoC firewall or due to the evolution from single player system to two-player or free-gameplay system. These are some of the issues that were faced and how they were resolved.

Issue Faced	Cause	Resolution
Gamestate not broadcasted to all Relay Clients	Relay Server was a single process and only wrote to the first client socket, causing race conditions.	I created a separate thread for every client connected to it and initialised a private queue for each client socket. Then I ensured that data from the int_comms_queue is relayed to all the private queues so that all the sockets receive the gamestate updates.
Visualizer response lag	Initially, I had Event flags for data from the visualizer. The MQTT Client would update these flags and the Game Engine would access these flags when carrying out health reduction. Relying on shared Event flags blocked the Game Engine until the MQTT update arrived.	I shifted from using event flags to using queues and I added a timeout for data from the visualizer. If I did not receive any data within the timeout, I would proceed with default values of setting visible to True and rain damage to 0.
SoC firewall blocked EMQX	Outbound port 1883 closed mid-project.	I added a reverse SSH Tunnel: localhost:1883 → broker.emqx.io:1883.
Duplicate/multiple action predictions from the AI Accelerator	The AI Accelerator would often predict more than 1 action within the required time-frame and that action would be processed by the Game Engine from the action queues.	I flushed the action queues after carrying out a particular action and whenever the Game Engine dequeued a new action frm the action queue, it would only dequeue the latest one to prevent stale commands.

Section 5.6.2 Game Engine Evolution

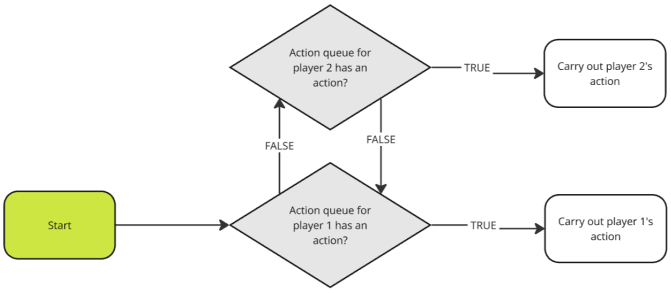
The biggest challenge during the entire process was deciding on using threads for each player in the Game Engine. Initially for two-player game, I had a single process for the Game Engine and two separate functions to handle player 1 and player 2. Then, I had the following “barrier” logic to handle the player actions:



Essentially, whichever player did their action first would be processed followed by the next player. The problem with this approach was that in case any player got timed out, the Game Engine would not follow this logic in the next iteration. It would still wait for the timed out player to do their action before moving on to the next player.

This caused an issue for us in our two-player game evaluation because the hardware disconnected for player 2 and as a result, player 1's actions were not being processed either, hence affecting the entire gameplay.

To resolve this, I tried to shift to creating separate threads for player 1 and player 2.



The issue with this was that if the AI Accelerator ended up predicting duplicate/multiple actions for player 1 before player 2, all those packets would end up being processed in player 1's thread and sent to the evaluation server. The evaluation server would discard duplicate packets and move on to the next frame and this would keep continuing till all the packets were processed, thus affecting gameplay.

In the end, we decided to go with threads only for free-gameplay and use the original "barrier" logic for two-player game because the team decided that we would rather have a barrier stopping the Game Engine from processing the first player again unless the second player has been processed.

Section 6 Software/Hardware AI

Section 6.1 Overall Methodology

We decided to forgo "start of action" detection. Rather, we run our model inference at a fixed frequency, allowing it to predict actions at any moment without any hard-coded limits on sensor values by adding an additional "idle" class to the model.

The steps taken to synthesize our final model into a hardware configuration for the FPGA on board the Ultra96 are as follows:

1. **Dataset collection & curation**: Collecting & processing training data to train the model
2. **Model Training**: Experimenting with various model architectures with quantization-aware training using Brevitas, and export as a .pth model.
3. **Synthesis, Implementation & Bitstream generation**: Generate a hardware executable from the model specification using FINN.
4. **Driver creation:** Tailor the PYNQ driver automatically generated by FINN to convert it to a process that can be run concurrently with the game engine on the Ultra96.

Because the conversion of the software model to the bitstream is automatic with the FINN library, there is no need for simulation to check for functional correctness. Rather, an end-to-end verification of functional correctness can be done by running the same training examples through the software model and the hardware model, and ensuring their outputs match completely.

Section 6.2 Data collection & transformation

Section 6.2.1 Data collection

Sensor data

The model was trained on labelled data from the hardware sensors.
The raw sensor data from the MPU6050s were used to calculate the yaw, pitch and roll of the sensors. These 3 readings are then streamed together with the acceleration and angular velocity (obtained directly from the sensors) to the relay node at a frequency of 25 Hz after converting them to unsigned 8-bit integers. (The original

sample rate of 50Hz was halved due to the unreliability of the Bluetooth connection between the Beetles and the relay node.)

Dataset collection
The training data was labelled by users using a button that streams its output (on/off) together with the sensor data. The user will hold the button down while performing the intended action, allowing us to differentiate between idle actions and when the user is performing the action.
When the Beetle is disconnected from the relay laptop, the relay continues to record data at the same sample rate, flagging disconnection by writing all sensor readings to be 0s.

Section 6.2.2 Feature engineering

To handle rows of 0s present in the data caused by temporary disconnections, valid sensor values are forward-filled, and an additional feature is used as a flag to indicate if that sample has been forward-filled or if it is a new set of readings from the IMUs. The feature holds a value of 32 if it is a new sample, 0 otherwise.

Additionally, the label (whether the button is pressed by the user during training data collection) is assumed to be 0 ("idle") for forward-filled samples. This ensures that the model learns to predict the idle action during disconnection.

A sliding window of size 25 is used to produce 1 second-long intervals of time-series sensor data, forming a potential training example. If the training example purely consists of forward-filled sensor readings, (i.e. disconnection for the entire duration of the sample), it is discarded. If more than 80% of the labels are not 0 (i.e. some action is being performed), the example will be labelled with the appropriate class. If not, it is used as an idle training example.

Overall, the final processed dataset consists of about 827,000 examples, with 25 time-steps and 19 features each. However, about 728,000 of these examples belong to the idle class; each player action only has about 9000 examples.

Section 6.3 Model training

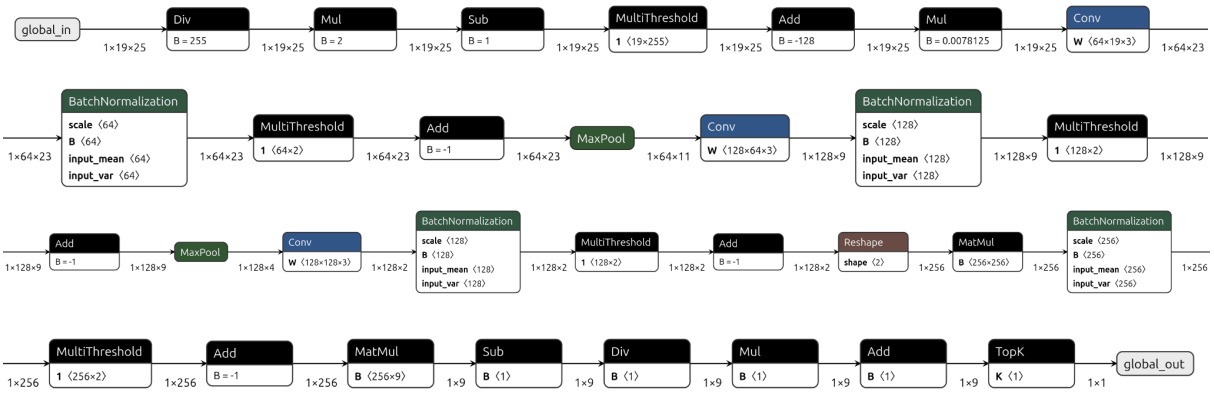
Section 6.3.1 Neural Network Architecture

A Convolutional Neural Network (CNN) was chosen as the model architecture. CNNs are suited for classification of time-series data because 1D convolutions allow the model to detect local temporal patterns (like sudden changes in motion) which can be critical for classifying activities or gestures. The model outputs the logits that model the possible classes of actions.

Additionally, the CNN was trained with Quantization Aware Training (QAT) using Brevitas. To minimize resource usage on the FPGA, the weights and activation bit widths were all set to 2 bits. To make up for the loss in precision, a more complex CNN with more channels was used. Through heavy experimentation and iteration, the following model architecture yielded the best test loss:

- 3 convolutional layers (64, 128, 128 channels respectively, last layer without pooling)
- 2 fully connected layers (256, 256 neurons respectively)

Batch Normalization layers were used between each convolutional / fully connected layer.



A top-k selector was added at the tail end of the model to select the index of the class with the highest value.

Section 6.3.2 Quantized CNN training loop

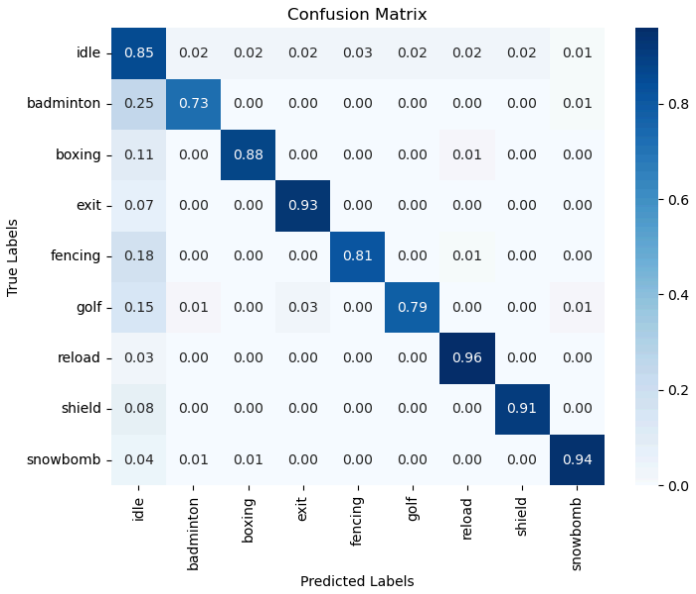
The CNN is trained using Brevitas, with Categorical Cross Entropy as its loss function. The loss function weight for the idle class was set as 2-3 times larger than the other classes. This was to ensure that the model was penalised more heavily for predicting an idle action as a player action.

Due to severe class imbalance ("idle" class with ~80x more examples), a weighted random sampler was employed to avoid model bias towards the idle class.

The model was trained using an Adam optimizer with a learning rate of 0.02 and a batch size of 3000 over 5 epochs. 5 epochs was enough to achieve convergence due to the large dataset size.

Section 6.3.3 CNN performance

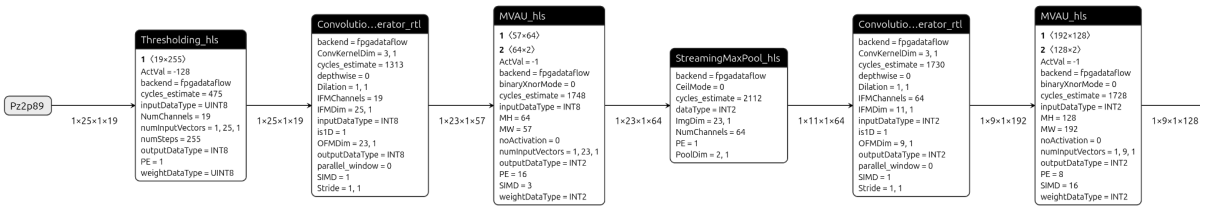
Overall, the final model achieved a training accuracy of 96.43% and a test accuracy of 85.57%. Below is the confusion matrix of the model on test data:

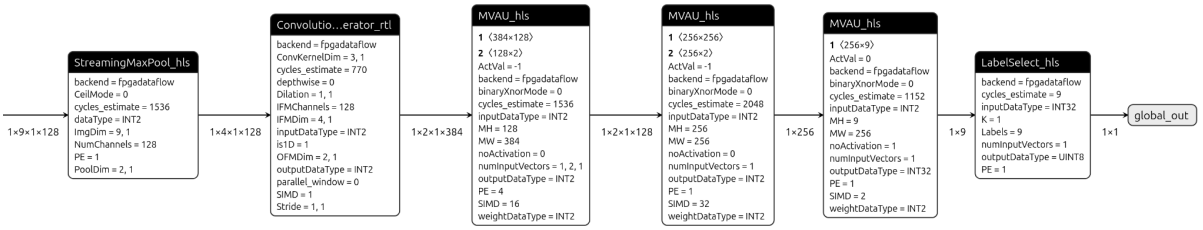


Section 6.4 Hardware implementation

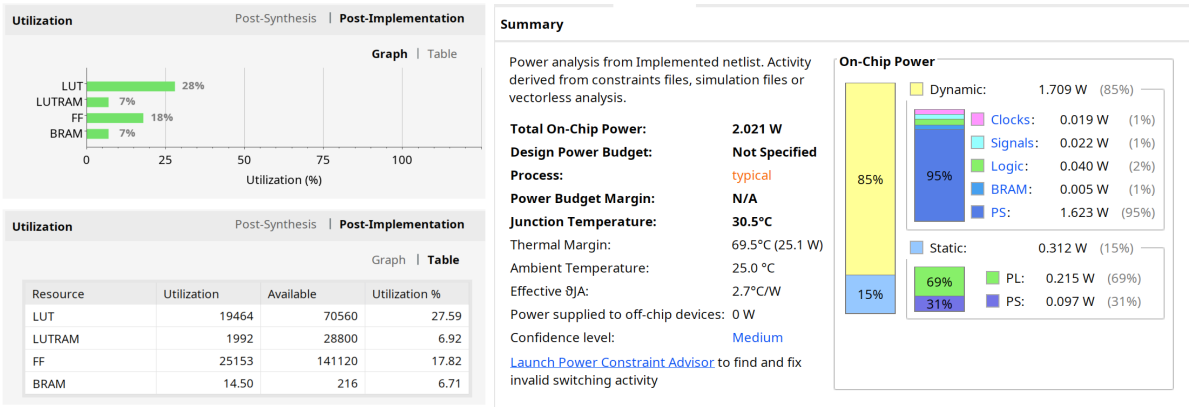
Section 6.4.1 FINN hardware generation

The conversion of the quantized model to a bitstream on the FPGA was achieved using the FINN library. The model is first converted to an ONNX specification. Successive transformations are then applied to the ONNX computation graph until all nodes in the graph can be synthesized by hardware. A Vivado project is then automatically generated by FINN with the stitched IP design. The model timing performance / resource usage tradeoff was configured using the parallelization parameters PE (Processing Elements) and SIMD (Single Instruction Multiple Data), which could be set for each node in the graph. Timing performance was prioritized over resource and power usage because there were no competing processes running concurrently with model inference on the FPGA.

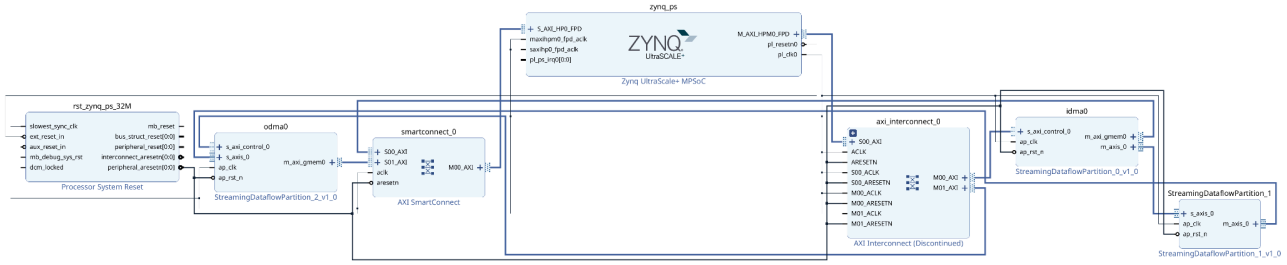




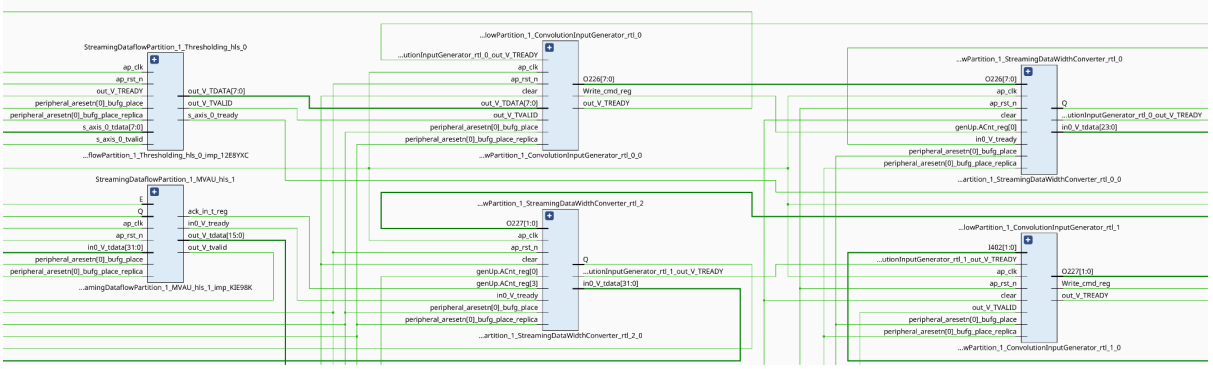
With the above combination of node parameters, the final hardware design had the following utilization levels:



FINN also automatically generates peripheral blocks, such as DMAs so that data can be transferred efficiently from the Zynq UltraScale MPSoC to the FPGA. They can be seen in the top-level block diagram generated by FINN:



Zooming in to the schematic of the Streaming Dataflow Partition 1 where all the hardware implementing the logic for model inference is located, we can see the hardware nodes corresponding to the ONNX graph. Here we see some of the convolutional layers:



Section 6.4.2 Functional verification & timing performance

FINN automatically generates a PYNQ driver for testing by using the MakePYNQDriver() transformation. The driver packs the data into appropriate shapes, taking into consideration the parallelization parameters specified beforehand, as that would impact the data width of the input module. The packed data is then copied over to the FPGA using DMAs.

The final bitstream and hardware handoff files were functionally verified using the same test dataset, using the FINN driver. The hardware implementation showed a 100% match to the predictions produced by the software model, indicating its functional correctness.

The hardware implementation processed 98504 examples in 80 seconds; yielding a latency of 0.82ms per inference, or a maximum inference frequency of 1223 Hz, which is way above our minimum requirement of 50Hz (25 Hz per player). Given the satisfactory timing performance, the PE and SIMD parallelization factors were kept the same for future iterations.

Section 6.4.3 Driver & SW process for HW/SW integration

The FINN driver was modified into a Python process, where it received IMU data from queues and put non-idle actions into queues for each player.

To further safeguard against accidental predictions when the player is idle, the FPGA is required to predict the same action over 8 consecutive time steps (0.32 seconds) before the prediction is put into the queue. After an action is predicted, a 1.6 second cooldown is employed to prevent multiple wrong predictions from flooding the queue. Additionally, new actions may only be predicted after the consecutive prediction streak from the previous prediction is broken. This is to prevent multiple predictions of the same type over the period where the player does only one action.

Section 6.5 Difficulties encountered

The difficulties encountered can be separated into the training stage and implementation stage:

Training Stage

1. **Unreliability of bluetooth connections between the Beetle and relay laptop.** The frequent disconnections meant that it was difficult to collect continuous chunks of valid data. This led to the need for further feature engineering; and made data augmentation impossible.
2. **Calibration of IMUs.** The initial resting position of the left and right hand subsystems affected the IMU values, since the IMUs would calculate an offset based on the direction of gravity and its relative position. Inconsistencies in the calibration of IMUs in the training stage led to undefined behaviour during inference when the calibration was not the same as during the data collection stage.
3. **Tedium in data collection.** A large amount of time was dedicated to collecting data for actions.

Implementation Stage

1. **FINN tooling.** The FINN tool often throws errors and exceptions deep within the library when certain graph transformations are unable to complete due to changes in model architecture. As such, model architecture experimentation was heavily restricted. For example, it was necessary to include Batch Normalization layers between every layer for FINN to infer Thresholding or MVAU modules. Additional restrictions that are imposed by FINN's complex dependencies (such as a data width of 8192 on some HLS modules, imposed by Vivado) can also lead to errors that are hard to understand, diagnose and debug with highly cryptic error messages.
2. **Driver creation & integration.** The integration of the hardware implementation with external communications also led to a few difficulties. The logic for handling invalid data had to be meticulously designed to replicate that of the feature engineering process for the training data fed to the model. For example, in the event of a disconnection, the first feature had to be set as 0, and the previous valid sensor values had to be forward-filled into the buffer for prediction.

Section 7 Software Visualizer and Game engine

Section 7.1 Visualizer Display and Design

Section 7.1.1 Data to Display

Player Status	**HP:** Green(player1) and red(player2) horizontal bars on the top of the screen, with numeric values **Shield HP**: Shorter yellow bar below the HP bars

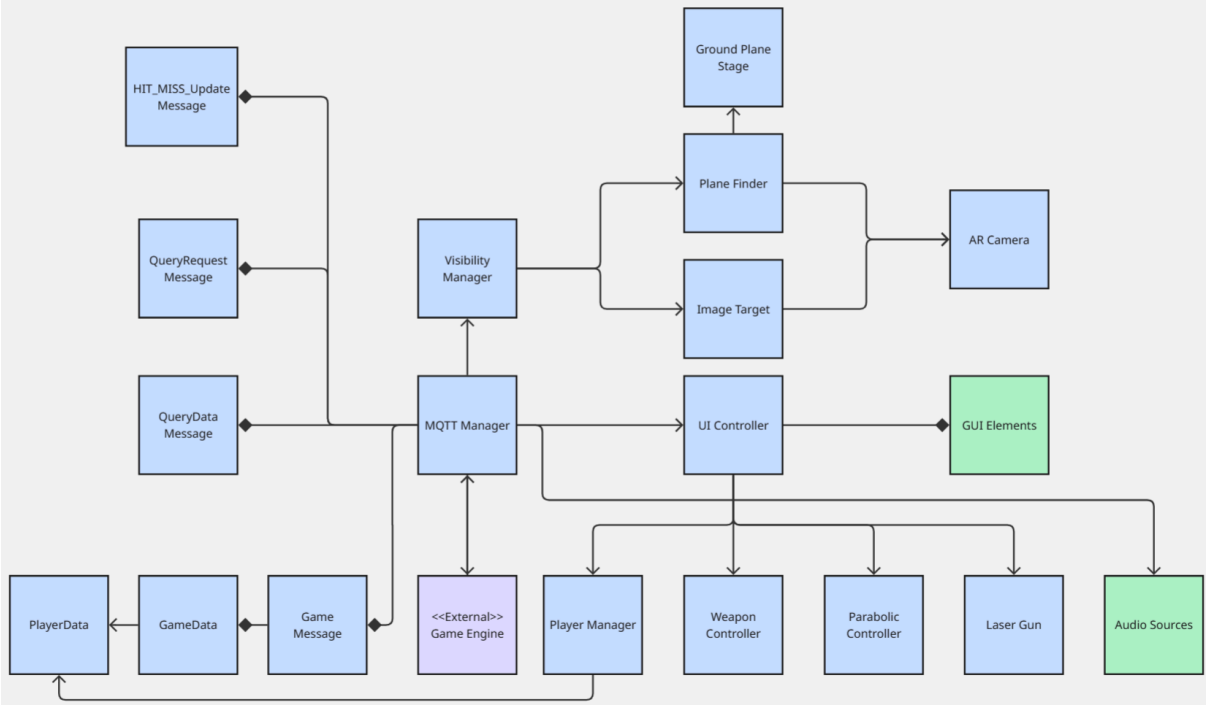
	respectively, with numeric values and only displayed when shield is activated **Score**: numeric value beside the HP bars respectively
Game props Status	**Located at the bottom left and righ for player 1 and 2 respectively** **Bullet Count:** Numeric value displayed beside laser beam icon **Bomb Count:** Numeric value displayed beside snow ball icon **Shield Count:** Numeric value displayed beside shield icon
Connection Status	**Displayed as a small square at top center, indicating connection status with the game engine**

Section 7.1.2 Design Constraints

1. **Limited Screen Space**:
   The screen size restricts the amount of visible space for the UI. UI elements must be compact and clearly organized to avoid cluttering.
2. **Real-Time Updates**:
   Gameplay stats and AR effects must update seamlessly to avoid breaking the game experience.
3. **Lighting Variations**:
   AR tracking may face challenges in environments with low light or excessive glare.
4. **Player Movement**:
   Rapid movement during gameplay may affect AR tracking accuracy, particularly for opponent detection and AR effect anchoring.
5. **Head Mount**:
   Visualizer is clamped on a head mount and worn by each player. As they do the actions, the camera could be temporarily blocked or may look away from the image target.

Section 7.2 Software Architecture

Section 7.2.1 High-level architecture of the visualizer subsystem



The system design follows a Facade pattern, with the MQTT Manager serving as a unified interface between external comms and internal visualizer modules. It orchestrates multiple independent components within the visualizer: UI Controller, Laser Gun, Weapon Controller, Parabolic Controller, and Visibility Manager. The MQTT Manager also connects to a cloud-hosted MQTT broker, where it subscribes to the topic "cg4002grpB10/ultra96" to receive player actions and game state updates, and publishes messages to "cg4002grpB10/visualizer" to report opponent visibility and the number of snow zones the opponent is currently in.

Incoming messages are handled through the DecodeMessage callback function. Based on the message type and content, the MQTT manager will call the appropriate service. For example, when a QueryRequestMessage is received, it invokes UIController.PerformAction to display visual effects for player actions, and UIController.SimulateDamage to reflect damage effects. It then uses visibility data from VisibilityManager.isOpponentVisible and the camCount attribute to construct and publish a HIT_MISS_UpdateMessage back to the game engine. Additionally, upon receiving a GameMessage, the MQTT Manager calls PlayerManager.UpdateAllData to refresh the game state displayed on the GUI, ensuring real-time synchronization between player input, game logic, and visual feedback.

Section 7.2.2 Frameworks and libraries

1. **Vuforia Engine**
   Provides real-time image recognition and tracking, and ground plane detection.

2. **Unity Canvas System**
   Used for responsive UI design to overlay player stats, some visual effects, and game state.

3. **M2MQTT for Unity**
   Facilitates real-time data exchange between the game engine and the visualizer app.

4. **UnityEngine**
   Core Unity framework used for building all gameplay components.
   Provides GameObject management, coroutine handling, physics simulation, and rendering.

5. **System.Text**
   Used for string encoding and decoding operations.
   Essential for converting text to byte arrays (Encoding.UTF8) when sending MQTT messages.

Section 7.2.3 Modules

1. **MQTT Manager Module**
   · Handles communication between the game engine and visualizer via MQTT.
   · Processes incoming messages and send to/trigger the appropriate modules.

2. **Game UI Module**
   · Displays game states and controls the implementer modules of action visualizations.
   · Dynamically updates UI elements based on gameplay data received via MQTT. Also plays a different voice for each action to improve the UX.

3. **Weapon Controller Module**
   · Manages the melee weapons: boxing gloves and fencing sword.
   · Triggers animations and effects based on player actions.

4. **Parabolic Controller Module**
   · Controls the physics and trajectory of thrown projectiles: snow bomb, golf, and shuttle
   · Simulates arc-shaped projectile movement and impact behaviors in AR space.

5. **Laser Shooter Module**
   · Handles the logic for firing laser projectiles from the player's weapon.
   · Calculates the direction and target position, spawns the projectile, and simulates its movement and collision detection.

6. **Visibility Manager Module**
   · Tracks opponent visibility using image detection (a QR code[1] used as marker) and camera feedback.
   · Determines whether a hit or miss should be registered based on marker presence.

7. **Player Manager Module**
   · Keep track of player data such as HP, shield count, bullet count, etc.
   · Updates internal game state and syncs with the UI module.

Section 7.3 Game UI Design

To overlay the scoreboard and other UI elements on top of the live camera feed, Unity's Canvas system is used by configuring in Screen Space to overlay mode. This allows UI components, such as player health bar, ammo count, shield status, and weapon objects to be rendered directly on top of the camera view provided by Vuforia engine. There was an update of the slider bar to improve the aesthetic aspect during the final evaluation.[2]

The camera feed itself is rendered as the background via the AR engine, while the Canvas elements remain fixed to the screen space, ensuring that the scoreboard stays visible regardless of the player's movement or orientation. This approach keeps the UI intuitive and non-intrusive, while maintaining full visibility of the AR gameplay.

The scoreboard and game stats are designed using Unity's built-in UI components like TextMeshPro, Slider Bar, Images, and Panels, etc., grouped logically under the Game UI Module. The UI Controller script dynamically updates these elements in real-time based on data received via MQTT, ensuring the UI reflects the current game state accurately.

Section 7.4 AR Visual Effects for Game Actions

To create a rich and immersive AR gameplay experience, the visualizer renders game actions directly on or around the opponent using Unity's particle effects, Canvas system, and physics-driven projectiles.[3] These visual effects are dynamically triggered based on player actions received via MQTT and are synchronized with real-world object positions using marker tracking and spatial logic.

The core decision-making for action visualization is handled in the PerformAction() method, which maps each gameplay action (e.g., shooting, shielding, throwing, melee attacks) to its corresponding AR effect. Before rendering, the system checks whether the opponent is visible and in an active quadrant using the

[1] Refer to Appendix Fig 7.1 for QR code image
[2] Refer to Appendix Fig 7.2 for comparison of the new and old GUI
[3] Refer to Appendix Table 7.3 for the individual visual effects

VisibilityManager. If conditions are met, the effect is instantiated at the target's position to visually represent successful interaction.

Section 7.4.1 Gun Shots

Originally, laser beams were intended to be rendered using Unity's Line Renderer with raycasting from the weapon to the hit location. This evolved into a more dynamic solution using a prefab-based projectile with a laser trail particle system. The projectile is moved in real time using a coroutine, which performs frame-by-frame raycasting to detect hits. This change resulted in more realistic visual effects and better compatibility with Unity's physics and AR tracking systems.

Section 7.4.2 Area-Based Throws (Snow Bomb, Golf Ball, Shuttle)

The initial design suggested using dedicated animation controllers for each projectile type, with specific arcs and transitions. In the final system, these projectiles are implemented through physics-based force application managed by the ParabolicController. They auto-aim at the opponent if visible, or follow a default forward arc otherwise. In the 2-player testing, a quadrant based anchoring was used to give an accurate snow damage zone count. This system ensures stable detection even with slight AR drift by anchoring the effects to markers on the ground rather than relying on pure distance. However, it was changed to a vision based snow zone detection in the unseen player test due to restrictions on such approach.

Section 7.4.3 Melee Attacks (Boxing, Fencing)

Close-range actions are handled by the WeaponController, which animates virtual gloves or swords with a quick forward movement to simulate punching or fencing. The attack is accompanied by a particle-based impact effect at the point of contact, providing satisfying visual feedback.

Section 7.4.4 Opponent Shield Effect

Shield actions are rendered by activating shield visuals on the opponent. If the shield is triggered and the opponent is currently visible, the system directly enables the shield GameObject at the corresponding target position. In cases where the opponent is not yet visible, a flag (needDelayShield) is set, and the CheckDelayShield() method is called from the image target's event handler to activate the shield as soon as visibility is restored. This ensures that shield visuals are always synchronized with game logic, even in cases of delayed marker image detection.

The overall system progressed through three development phases. The single-player phase focused on creating and testing AR visual effects in isolation. The two-player phase introduced real-time communication through MQTT and tied player actions to remote visual effects. The final phase introduced handling for partially tracked or unseen opponents by deferring effect rendering and syncing them when marker visibility was restored.

Section 7.5 Challenges and Solutions

1. **Reliable Collision Detection for Fast-Moving Projectiles**

One of the early challenges I encountered was ensuring that high-speed projectiles, specifically laser beams reliably registered collisions with targets. Initially, I used Unity's built-in collision detection via OnTriggerEnter and OnCollisionEnter, but during the 1-player testing I discovered that these callbacks were inconsistent for fast-moving objects. The laser bullet would often travel so quickly between frames that it would pass through the target object without triggering the hit, a common issue known as "tunneling" in physics-based simulations.

To solve this, I explored several alternatives, including using continuous collision detection. However, the real breakthrough came when I implemented a coroutine-based solution for the 2-player testing, which I refined with guidance from Unity resources. Instead of relying solely on Unity's event callbacks, I continuously checked the distance between the projectile and the target within a coroutine.[4] This allowed me to detect when the projectile was approaching the target closely enough to trigger an effect, without relying on the unreliable physics events. Additionally, during the final evaluation, I added a guard clause in the bullets script to continuously check if it is within a 0.1 unit distance to the target, and destroy itself within the range if the coroutine did not handle it properly. It provided both smoothness in the visual effects and precision, especially important for a high-speed gameplay element like a laser shot.

[4] Refer to Appendix Fig 7.4

2. Handling MQTT Connection Stability and Feedback

During the development of the networked components of the project, one of the key challenges involved maintaining a reliable connection to the MQTT broker. In early testing, there were instances where the app would appear to be running, but the connection to the broker was either never established or had silently dropped after the app had been idle for some time. This lack of visibility made it difficult to diagnose failures. In fact, during the 1-player test, the system failed simply because the connection hadn't been properly initialized, and there was no indication of that on the user side.

I have learned this lesson and to address this issue, I implemented a more robust connection-handling system using M2MQTT's event-based callbacks: OnConnectionFailure, OnConnectionLost, and OnDisconnected[5], which is customizable by overriding the corresponding base class methods. These allowed me to detect abnormal disconnections or failures and respond by automatically attempting to reconnect to the broker. This significantly improved reliability during idle periods and unexpected network interruptions.

In addition to the backend fix, I also introduced a visual feedback mechanism into the UI. Since the 2-player testing, a small status square was added to the game UI to reflect the connection state in real time: green for connected, blue for connecting, red for disconnected, and yellow for a previously active connection that had dropped. This gave both users and developers immediate insight into the network status and helped with faster debugging during testing.

3. Inaccurate AR Spatial Detection for Snow Bomb Damage

One of the more technical challenges came from implementing the area-based damage system for the snow bomb mechanic. The goal was to determine whether a player was within the blast radius of a snow bomb after it landed. Initially, this system relied on two factors: the visibility of the snow bomb within the AR environment, and the distance between the snow bomb's center and the player's target hitbox.

However, a recurring issue stemmed from AR's inherent inaccuracy in spatial distance measurement, particularly in mobile AR setups. Due to slight inconsistencies in positional tracking, the distance check would sometimes falsely report a player as being inside or outside the damage area. This led to unreliable behavior — sometimes the snow effect would trigger correctly, but other times, it wouldn't activate even when it visually appeared like it should.

To solve this, we replaced the continuous distance-based detection with a more stable, quadrant-based system. We physically divided the play area into four quadrants using four visible AR image markers placed on the ground. When a snow bomb landed, it was hard-assigned to the nearest visible quadrant marker. The system would then check whether the other player was visible in that same quadrant — if they were, the snow effect and damage were applied. This approach traded spatial precision for consistency and robustness, which was much more suitable for AR testing conditions. It ensured 100% reliability as long as the relevant markers were tracked.

4. Switching from AR Foundation to Vuforia for Reliable Target Detection

A major architectural challenge in the project was the transition between two different AR frameworks. The project was initially built using Unity's AR Foundation, which provided a unified interface for handling image tracking, plane detection, and camera input across different devices. While AR Foundation offered solid ground plane detection and generally stable performance, a significant limitation emerged during testing: its image target detection was unreliable at distances greater than one meter. This was still acceptable in the 1-player testing as there was no physical movement involved. However, as we moved on to the 2-player game, this limitation severely impacted gameplay, as image targets placed just a short distance away from the camera would frequently fail to register or maintain tracking, something that's critical in a marker-based AR experience.

Given the nature of the project, which relied heavily on image-based tracking for positioning key objects like players and snow bombs, we made the decision to switch to Vuforia engine, a specialized AR SDK known for its robust image target tracking. After benchmark testing of new images[6] (apart from QR code image target) on the AR Foundation framework, it was concluded that AR Foundation is fundamentally inferior to Vuforia Engine in terms of image detection, regardless of the image quality or pattern complexity. Therefore the switch was inevitable. This switch introduced a new set of challenges, as it required reworking a significant portion of the existing codebase and learning the new framework, particularly in areas that handled visibility detection and target lifecycle management.

[5] Refer to Appendix Fig 7.5
[6] Refer to Appendix Fig 7.6 for the alternative images tested

To make the transition, I adapted many of the original AR Foundation methods to their Vuforia equivalents by reading the online documents. This included rewriting how tracking state was handled, how snow bombs were anchored to the ground, and how visibility was monitored over time. While the migration involved a non-trivial amount of code refactoring, the end result provided a much more reliable AR experience, especially for scenarios where markers needed to be detected from farther away.

Section 8 Future Work: Societal and Ethical Impact, Extension

Section 8.1 Societal Impact

Section 8.1.1 Sports Training

Our system's ability to stream IMU data from the wearables can be used to enhance training for athletes. IMU data could be collected for each individual athlete and the AI can be trained to learn the optimal stroke/ technique for their sport. Furthermore, the data can provide insights into potential areas where an athlete needs improvement. IMU data from match play can also be crucial for recreating gamelike scenarios which would be useful when developing training plans.

Section 8.1.2 Physiotherapy

The Covid-19 pandemic had led to the rise in Tele-Physiotherapy, where patients can now do physiotherapy in the comfort of their own homes through online conferencing platforms. However, one drawback of tele-physiotherapy is that the physiotherapists have to rely solely on visual cues from a fixed camera angle to guide their patients. Our system could be modified to stream data from IMUs attached to the patients to the physiotherapist. The AI can then be trained to interpret these data to provide the physiotherapist a more reliable means to determine if the patient is doing the exercises correctly. Furthermore, the AI can be modified to take the role of the physiotherapist, providing real-time feedback to the patient, freeing up manpower for other tasks.

Section 8.2 Ethical Impact

As with any system that collects user data, particularly body movement and spatial behavior, there are several ethical considerations associated with our system. The streaming and processing of IMU and camera-based data raise concerns about privacy, consent, and data security. If extended into sports training or physiotherapy contexts, user movement data could be highly personal, revealing sensitive health or performance information. Ensuring that data is anonymized, encrypted, and used solely with informed consent would be essential in any real-world deployment.

Section 8.3 Extension

The system could also be adapted into VR environments or hybrid XR setups, allowing players to interact both in AR and in virtual simulations. This would open doors to serious games, remote team-building exercises, and simulation-based training.

Another potential extension involves enhancing the networking layer, enabling support for more players, game modes, or even distributed multiplayer sessions across different locations. Integrating cloud-based MQTT brokers or WebRTC could allow the system to scale beyond local networks.

References

Links

[1] "MOLLE," Wikipedia. Dec. 20, 2024. Accessed: Jan. 24, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=MOLLE&oldid=1264182480

[2] H. Site, "MPU6050: Arduino 6 Axis Accelerometer + Gyro - GY 521 Test & 3D Simulation," Instructables. Accessed: Jan. 24, 2025. [Online]. Available: https://www.instructables.com/MPU6050-Arduino-6-Axis-Accelerometer-Gyro-GY-521-B/

[3] "1/5PCS Mini DC-DC Step-Up Boost Voltage Converter 3.7V to 12V Power Module," eBay. Accessed: Jan. 24, 2025. [Online]. Available: https://www.ebay.com.sg/itm/145847408103

[4] "10 Advantages of Lithium Polymer Battery." Accessed: Jan. 24, 2025. [Online]. Available: https://www.grepow.com/blog/10-advantages-of-lithium-polymer-batteries.html

[5] ElectronicCats/mpu6050. (Jan. 18, 2025). C++. ElectronicCats. Accessed: Jan. 24, 2025. [Online]. Available: https://github.com/ElectronicCats/mpu6050

[6] J. Rowberg, jrowberg/i2cdevlib. (Jan. 24, 2025). C++. Accessed: Jan. 24, 2025. [Online]. Available: https://github.com/jrowberg/i2cdevlib

[7] "The MPU6050 Explained," Programming Robots. Accessed: Jan. 24, 2025. [Online]. Available: http://mjwhite8119.github.io/Robots/mpu6050

[8] Arduino-IRremote/Arduino-IRremote. (Jan. 23, 2025). C++. Arduino-IRremote. Accessed: Jan. 24, 2025. [Online]. Available: https://github.com/Arduino-IRremote/Arduino-IRremote

[9] "SB-Projects - IR - NEC Protocol." Accessed: Jan. 24, 2025. [Online]. Available: https://www.sbprojects.net/knowledge/ir/nec.php

[10] D. Workshop, "IR Remotes Revisited - 2023," DroneBot Workshop. Accessed: Jan. 24, 2025. [Online]. Available: https://dronebotworkshop.com/ir-remotes/

[11] Avishay, avishorp/TM1637. (Jan. 11, 2025). C++. Accessed: Jan. 24, 2025. [Online]. Available: https://github.com/avishorp/TM1637

[12] I. Harvey, "bluepy documentation" Jan. 2025. Accessed: Jan. 25, 2025. [Online]. Available: https://ianharvey.github.io/bluepy-doc/

[13] DFRobot, *Bluno*. Accessed: Jan. 24, 2025. [Online]. Available: https://wiki.dfrobot.com/Bluno_SKU_DFR0267#target_6

[14] "AES — PyCryptodome 3.9.9 Documentation," Pycryptodome. Accessed: Jan. 25, 2025. [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html

[15] "Reverse SSH Tunneling: The Ultimate Guide," Qbee.io. Accessed: Jan. 25, 2025. [Online]. Available: https://qbee.io/misc/reverse-ssh-tunneling-the-ultimate-guide

[16] "Paho-Mqtt," PyPI. (Sept. 2, 2018). Accessed: Jan. 25, 2025. [Online]. Available: https://pypi.org/project/paho-mqtt

[17] The HiveMQ Team, "Getting Started with MQTT," HiveMQ. (Feb. 15, 2024). Accessed: Jan. 25, 2025. [Online]. Available: https://www.hivemq.com/blog/how-to-get-started-with-mqtt

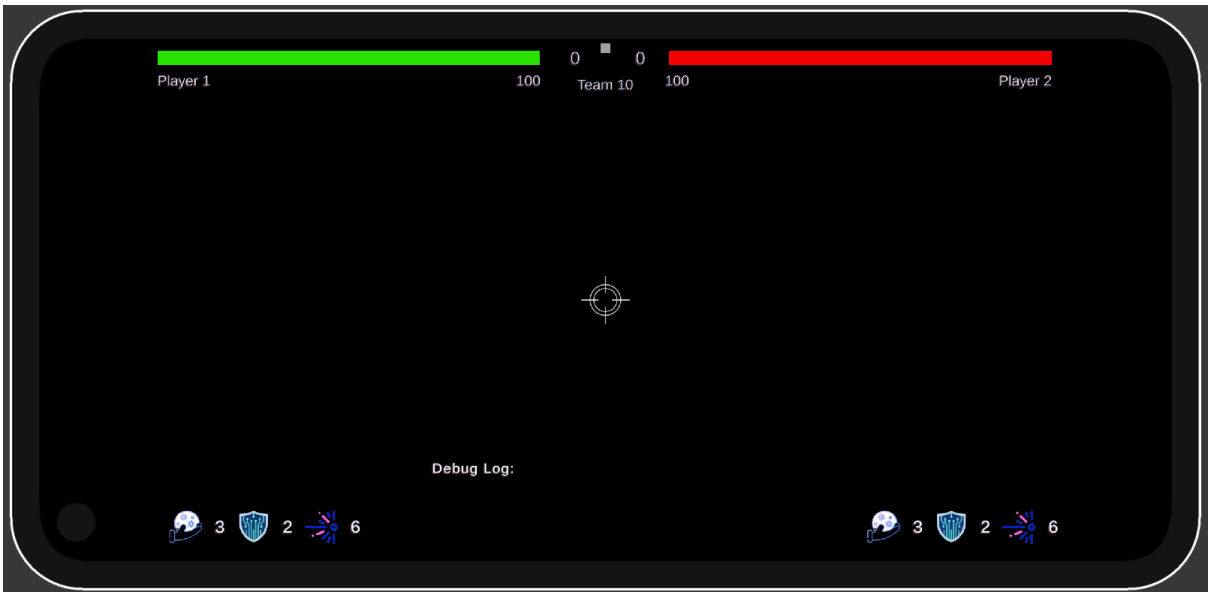
Images

Vest	https://shopee.sg/product/1017223840/23271419896?d_id=99dd9&uls_trackid=51qqp5p800rn
Forearm Guard	https://shopee.sg/product/293608406/25914736022?d_id=99dd9&uls_trackid=51qqr4g700rn
DFR0095 IR Transmitter	https://wiki.dfrobot.com/DIGITAL_IR_Transmitter_Module__SKU_DFR0095_
IMU MPU6050	https://www.electronicwings.com/arduino/mpu6050-interfacing-with-arduino-uno
Bluno Beetle	https://docs.cirkitdesigner.com/component/ee48dd5f-df70-466d-890b-4603c4d9f548/bluno-beetle
DFR0029-B Push Button	https://wiki.dfrobot.com/DFRobot_Digital_Push_Button_SKU_DFR0029
TM1637 7 Segment Display	https://lastminuteengineers.com/tm1637-arduino-tutorial/
Li-Po Battery	https://continental.sg/product/3-7v-rechargeable-1500mah-li-po-lithium-battery/
DFR0094 IR Receiver	https://wiki.dfrobot.com/Digital_IR_Receiver_Module__SKU_DFR0094_
TMB12A05 Buzzer	https://grabcad.com/library/tmb12a05-1

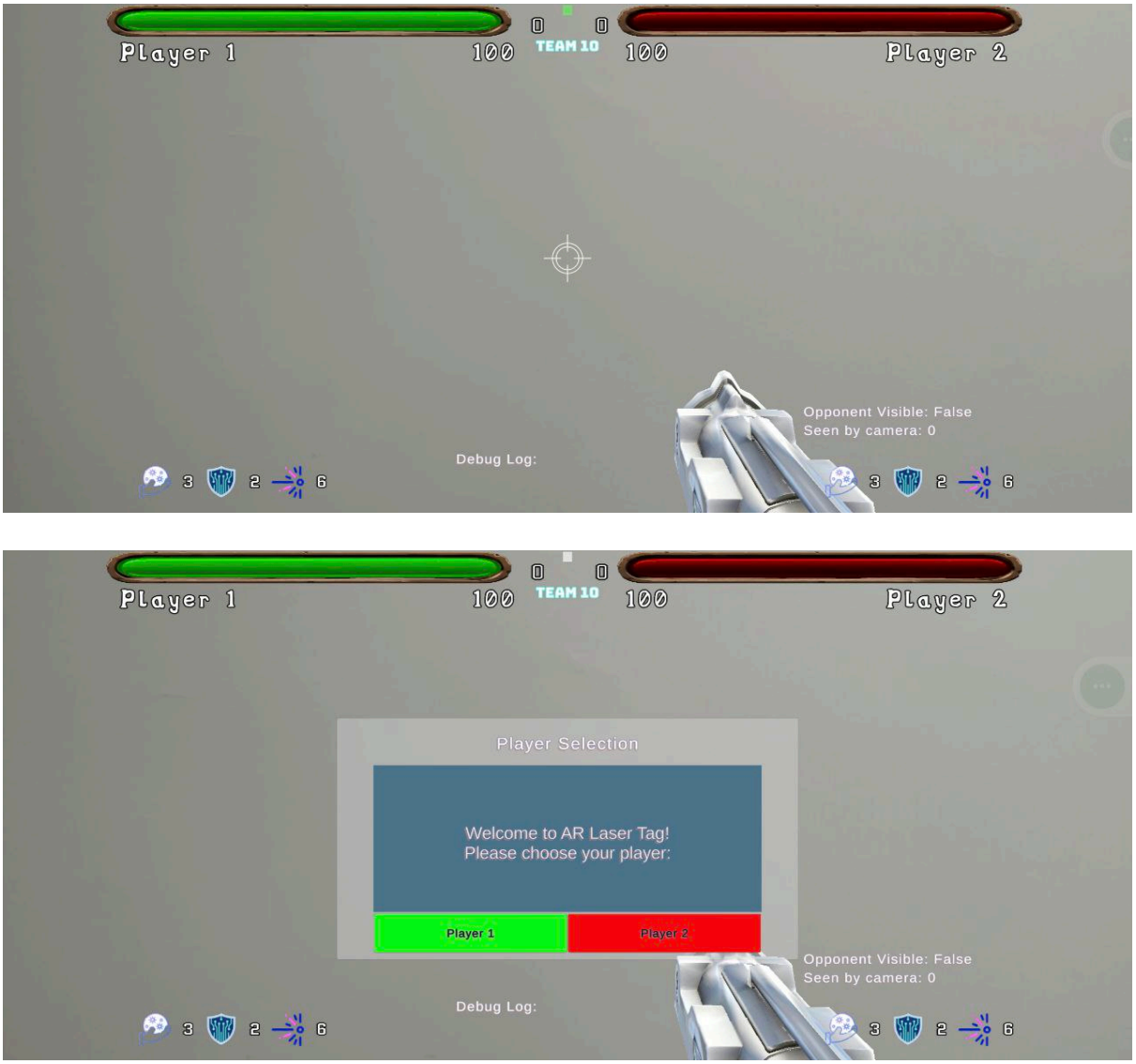
Appendix

1. **Visualizer**

Fig 7.1

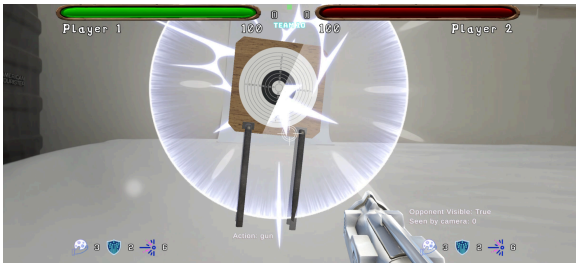
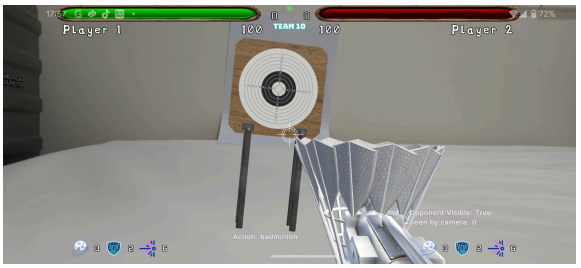


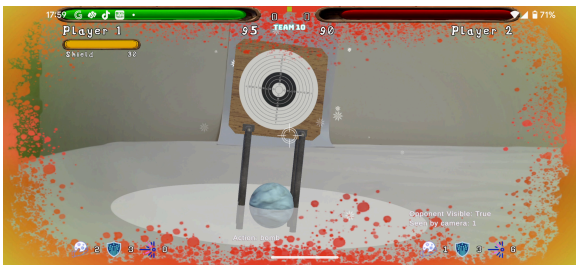
- Old GUI



- New GUI with player selection panel

Fig 7.2

Action	Visual Effect
Laser gun shot	
Golf	 
Badiminton	 
Boxing	

Fencing	
Snow Bomb (hitting vs. landing and snow fall)	 
Hit on opponent's shield	
Attack missed (with shield enabled)	
Take damage (with shield enabled)	

Reload (with shield enabled)	

Table 7.3

```
1 reference
private IEnumerator MoveLaser(GameObject projectile, Vector3 direction, Vector3 targetPoint)
{
    float distanceTraveled = 0f;

    if (projectile == null)
    {
        yield break;
    }

    while (projectile != null && distanceTraveled < maxDistance)
    {
        float step = projectileSpeed * Time.deltaTime;
        distanceTraveled += step;

        // Move the projectile
        projectile.transform.position = Vector3.MoveTowards(projectile.transform.position, targetPoint, step);

        // Perform RaycastAll to detect multiple hits (better for fast projectiles)
        RaycastHit[] hits = Physics.RaycastAll(projectile.transform.position, direction, projectileSpeed * Time.deltaTime);
        foreach (RaycastHit hit in hits)
        {
            if (hit.collider.CompareTag("Hitbox") || hit.transform.name == "Model" || hit.transform.name == "ARTarget")
            {
                Debug.Log("Laser hit: " + hit.collider.name);
                yield break; // Stop the coroutine
            }
        }

        // Destroy if it reaches max distance
        if (projectile != null && distanceTraveled >= maxDistance)
        {
            Destroy(projectile);
            yield break;
        }

        yield return null; // Wait for the next frame
    }
}
```

Fig 7.4

```
4 references
protected override void OnDisconnected()
{
    base.OnDisconnected();
    isConnected = false;

    Debug.Log($"{DateTime.Now:yyyy-MM-dd HH:mm:ss.fff} - Disconnected.");
    connectionIcon.GetComponent<ConnectionUI>().ShowDisconnectedStatus();
    // try reconnecting
    if (!isConnected)
    {
        Connect();
        SubscribeTopics();
    }
}

4 references
protected override void OnConnectionLost()
{
    base.OnConnectionLost();
    isConnected = false;
    Debug.Log($"{DateTime.Now:yyyy-MM-dd HH:mm:ss.fff} - Connection Lost.");
    connectionIcon.GetComponent<ConnectionUI>().ShowConnectionLostStatus();
    // try reconnecting
    if (!isConnected)
    {
        Connect();
        SubscribeTopics();
    }
}
6 references
protected override void OnConnectionFailed(string errorMessage)
{
    base.OnConnectionFailed(errorMessage);
    isConnected = false;

    connectionIcon.GetComponent<ConnectionUI>().ShowDisconnectedStatus();
    InputHandler.Instance.AddUiMessage("Connection failed: " + errorMessage);
    if (!isConnected)
    {
        Connect();
        SubscribeTopics();
    }
}
```

Fig 7.5


two.png
twitter.jpg
@nuscomputing
7764.png
nature
SNOOZE REPORT
New Piskel.png
new_qr_code.png
one.png
simple_pat.png

Fig 7.6