# Animal Movement Tracker

**Authors:** Tarik Esen (36502), Muhammed Ali Baskal (37554)

## 1. Introduction & Problem Statement

Across livestock farms, pet homes, and research environments, behavioral changes in animals often go unnoticed due to the limits of human observation. This leads to undetected distress, illness, escape attempts, and welfare issues. As a result:

- Distress, sickness, and anxiety often go unnoticed
- Escape attempts and injuries cause avoidable economic loss
- Annual welfare monitoring is time-consuming and prone to error
- Research centers lack long-term, objective emotional and behavioral data

This creates a clear need for an automated system capable of detecting emotionally relevant behavioral patterns in real time. Existing livestock and pet monitoring solutions focus on GPS location, temperature detection, rumination/feeding, and general activity. (Alsaaod et. al, 2015). No major commercial product offers informed analysis of behavioral signatures or escape intent, so this missing layer keeps farms and animal owners reactive rather than proactive.

The project "Animal Movement Tracker System" (AMT) addresses this gap with a collar-mounted controller device that streams motion data to a database, where a model classifies behavioral patterns such as moving, shaking, and rest cases, while sitting at the intersection of smart agriculture/precision livestock farming, pet technology, and, IoT behavioral analytics. The system targets animal owners, livestock farms, and research institutions in Germany and nearby EU regions, with a realistic early serviceable obtainable market of numerous institutions and customers.

For the system, a design has been created whereby movement data from a network of wearable, motion-sensing controllers on each animal is collected at a central gateway, then interpreted by gesture model running on the gateway, and behavioral status outputs for each animal are made accessible to system owners from anywhere via a cloud server. Figure 1 shows the schematic architecture of the project.
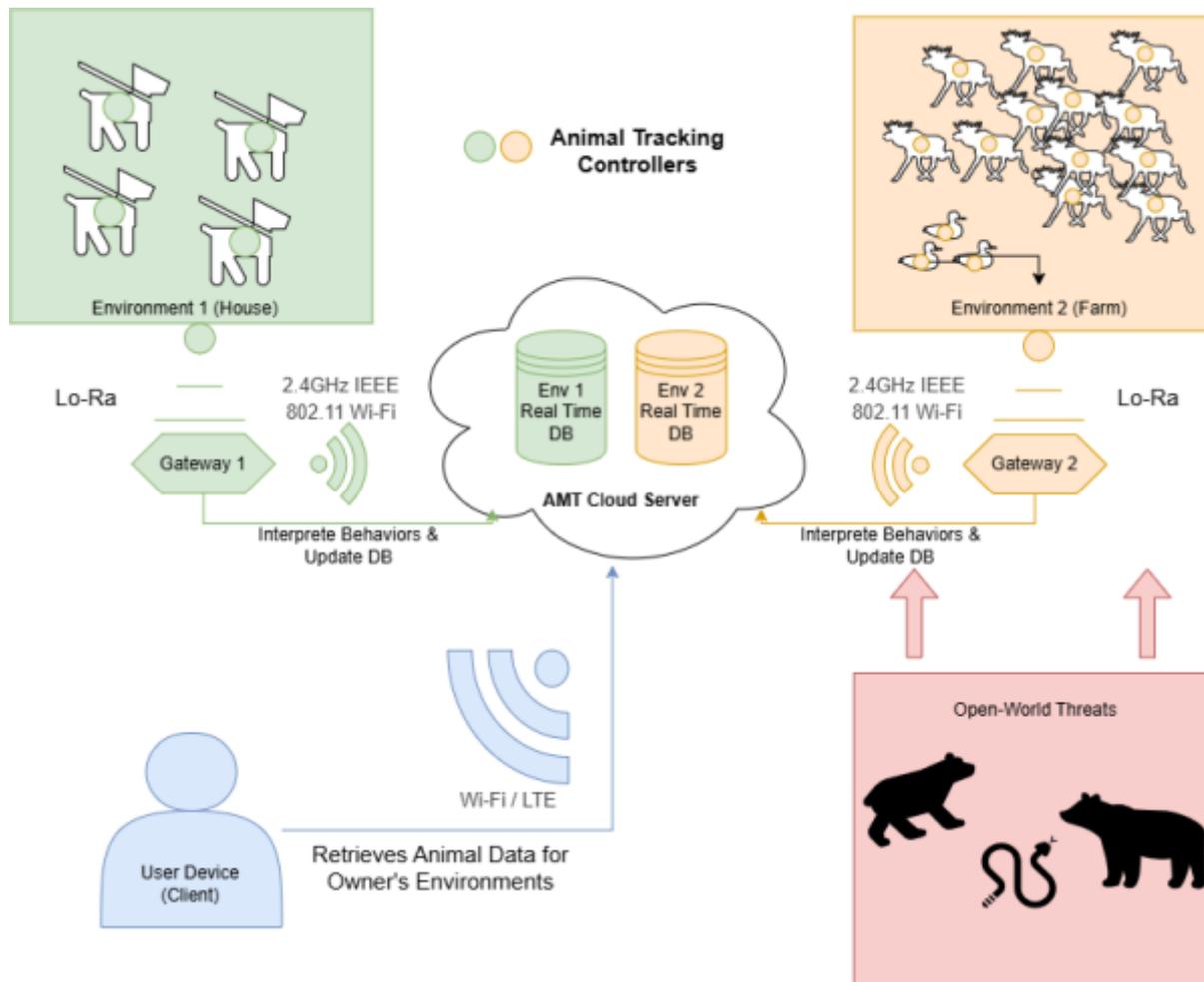
**Figure 1** General Architecture Diagram of AMT System

# 2. Methodology

## 2.1 Project Overview

The Animal Movement Tracker System has been designed as a complete, end-to-end solution for detecting movement patterns in animals through wearable IoT hardware and AI-driven analysis. The system addresses the core challenge identified across farms, pet homes, and research environments: the inability to continuously and objectively monitor behavioral shifts. The system consists of three integrated modules:

- Controller Unit (Wearable Device)
- Cloud Database (Real-Time Data Layer)
- AI Motion Inference Powered Gateway (Behavioral Classification Engine / TFlow + Keras)

Each module has been purposefully designed to operate in real time, provide reliable data transfer, and maintain usability for both farm-scale and individual pet environments.

## 2.2 Wearable Controller Unit

Throughout the system, there is a power-saving controller module that can be worn by animals, regularly samples movement data from the animal at a fixed frequency via an attached Inertial

Measurement Unit (IMU), and transmits this data wirelessly to the gateway module. The movement data of each animal is detected by the associated controller module. A herd of animals in a communal area communicates with the gateway as a sensor network composed of these controllers.

## 2.2.1 Microcontroller

Given the job description, it has been decided that a mainstream microcontroller should be selected as the core of the controller module, as it requires both power saving and instantaneous high processing power triggered by events (window bulking and transmission to the gateway channel). The Micro Controller Unit (MCU) requires one I2C controller for IMU data measurement (accelerometer and gyroscope) and two UART controllers for Lo-Ra module communication and programming, and must also be capable of operating at low power consumption levels. It must be able to execute processing commands with low latency at speeds ranging from 24 to 168 MHz while offering a lower cost. Therefore, the STM32F072RBT6 microcontroller from the Mainframe and low-cost (F0) band, with a 48 MHz Cortex-M0 core, 16 external interrupt channels, low-power mode support, 2x I2C and 4x UART controllers, was selected as the main core of the module. The interrupt channels allow the system's power level to be adjusted according to signals from external units, while the I2C in and UART interfaces enable data and clock signal connections between the external modules and the microcontroller. The connections can be observed in Figure 2.
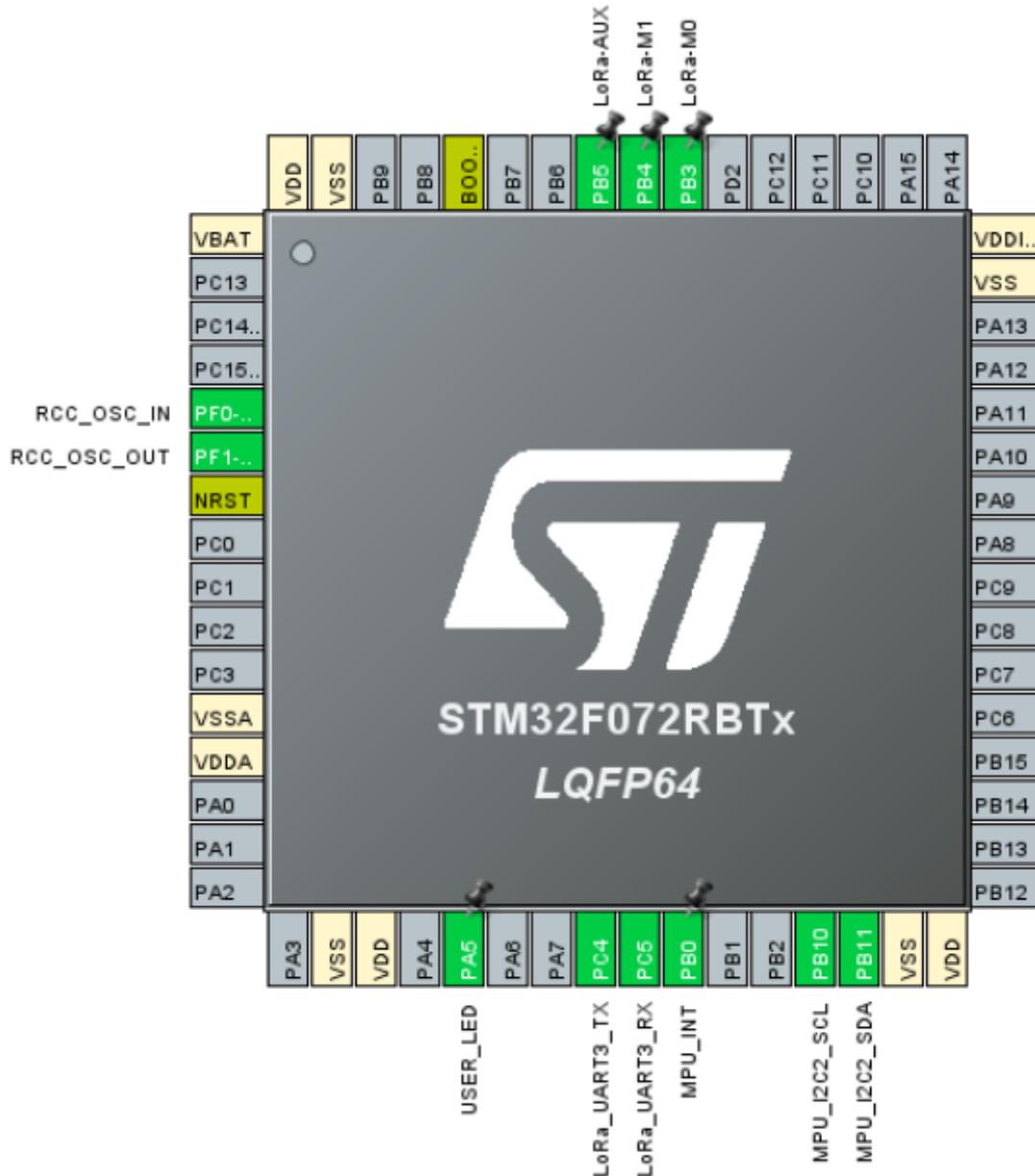
**Figure 2** Controller MCU and Physical Connection Mapping

## 2.2.2 Movement Measurement

It has been decided to use an Inertial Measurement Unit (IMU) to measure the linear acceleration of the animal along the x, y, and z axes, as well as its angular velocity data (gyro), at a regular measurement frequency, in order to measure the animal's locomotor behaviour. For this purpose, an IMU capable of measuring at sampling frequencies up to 8 kHz is required, featuring: 4-level acceleration (2, 4, 8, and 16 g), 4-level gyroscope (-/+ 250, 500, 1000, 2000 dps), programmable and capable of sending data as a slave via the I2C interface through the 0x68 address, was selected due to its mainframe usage capability and its ability to produce data with the required accuracy and quality according to the project requirements. The MPU data and clock lines are interfaced with the MCU via I2C(2) (PB10[SCL], PB11[SDA]), and the GPIO EXTI line's 0.pin (PB0) is used for the interrupt signal (INT) required for power saving. Figure 3 shows a diagram representing the connections between the IMU and the MCU.
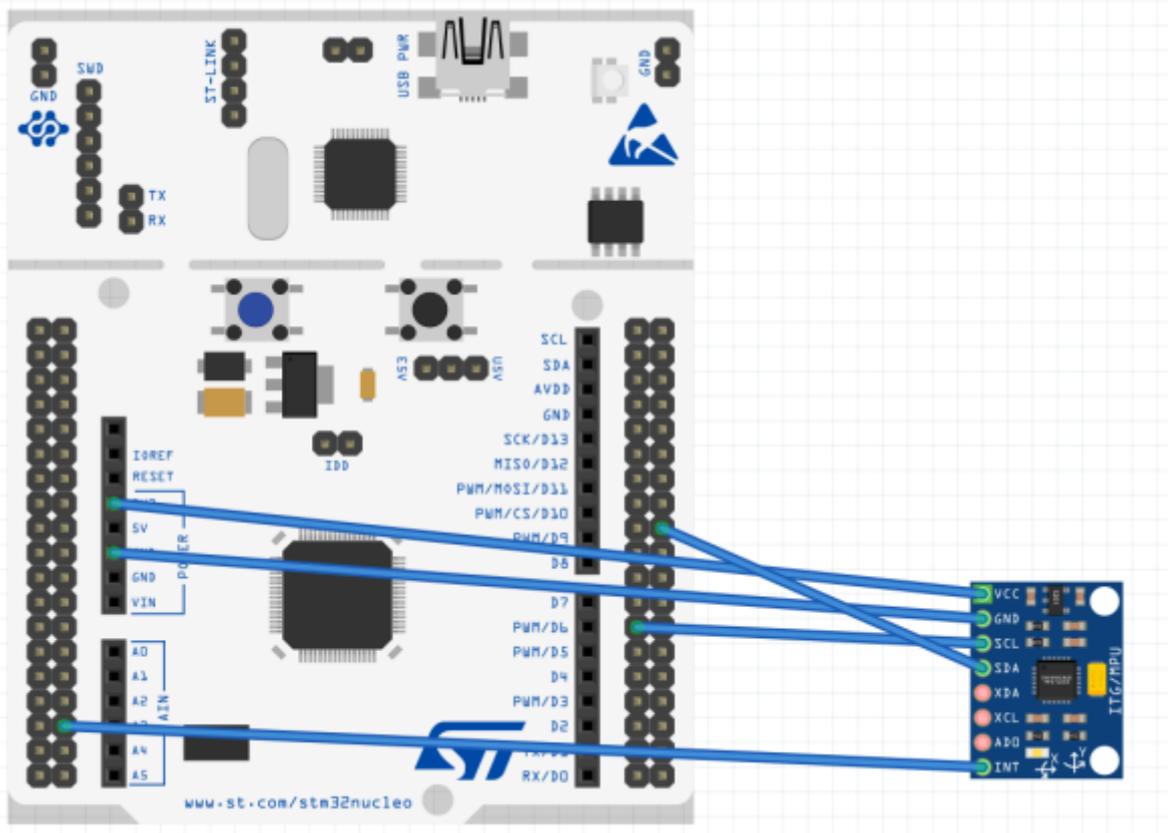
**Figure 3** Interfacing of the MCU and MPU6050

To facilitate easier normalisation for IMU measurements, the accelerometer configuration was selected as +/-2g, and the gyroscope configuration as +/-250dps. Each measurement was provided using windows containing a total of 96 samples, where each sample contains 6 values (ax,ay,az,gx,gy,gz). To facilitate communication synchronisation, the IMU sampling frequency was set to 1kHz and the SMPLRT_DIV value to 7 so that each window would be collected within a 1-second time interval, resulting in a sampling frequency of 125Hz on the controller side. This yielded an IMU window period of approximately 0.8 seconds.

### 2.2.3 Power Management

The controller unit is designed with an event-driven low-power architecture to minimise energy consumption. Instead of continuously measuring and transmitting data, the system waits in STOP mode during idle periods. The MPU6050 sensor is configured to generate a hardware interrupt when the specified motion threshold is exceeded. This interrupt line is connected to the external interrupt (EXTI) input of the STM32 microcontroller. When the microcontroller enters STOP mode with the WFI (Wait For Interrupt) command, the core clock signal is stopped and power consumption is significantly reduced. The interrupt signal generated by the MPU6050 is processed via the NVIC, causing the microcontroller in standby mode to wake up. After waking up, the system clock configuration is restarted and the sensor interrupt flag is cleared. Then, a fixed-length window consisting of a total of 96 samples at a sampling frequency of 125 Hz is collected from the IMU. Each sample contains a total of six 16-bit raw measurements: three-axis accelerometer and three-axis gyroscope data. Once the measurement window is complete, the data is packaged according to the defined LoRa framing protocol and transmitted to the gateway unit. During data transmission, specific GPIO pins are used as marker signals to facilitate power profile analysis. After the transmission process is completed, the system is returned to STOP mode and remains in standby until a new motion interrupt occurs. The entire process diagram is illustrated in Figure 4.
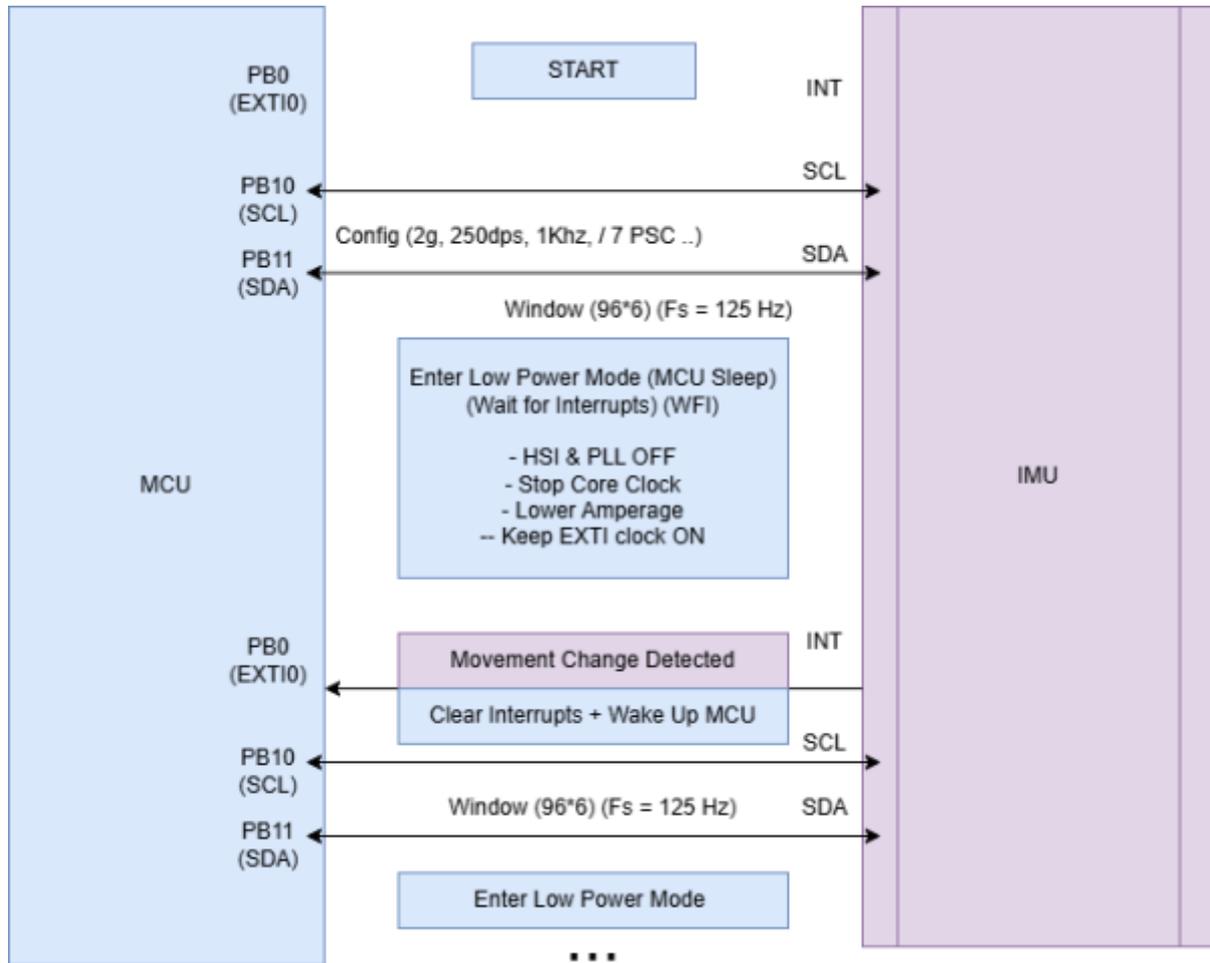
**Figure 4** Controller Power Management

## 2.3 Communication

Wireless communication between the controller unit and the gateway is implemented via a LoRa-based serial communication protocol. LoRa technology was chosen due to its low power consumption, long range, and high noise immunity. As the system was designed to operate on battery power within the scope of the project, a low-data-rate but energy-efficient communication infrastructure was selected instead of continuous Wi-Fi or high-data-rate RF solutions. LoRa's high receiver sensitivity and ability to operate in low SNR environments enhance the system's reliability. The communication architecture consists of a controller connected to the LoRa module via UART and a gateway unit connected to the Raspberry Pi via USB-UART. The LoRa module is configured in transparent transmission mode. This allows data frames to be sent directly without the need to write a special LoRa MAC layer on the microcontroller side.

### 2.3.1 Interface

As the project is intended to operate within the European Union, LoRa communication must be carried out within the limitations set out in the relevant European Union communications regulations (ETSI, 2018) covering the relevant protocol. For this purpose, Ebyte E32-900T30D LoRa transmission modules have been used (shown in Figure 5), which have a maximum transmission power of +30 dBm and an air data rate of 19.2k, can perform fixed and transparent transmission, have a 2-byte address capacity, and can broadcast between 860-920 MHz frequency channels.

**Figure 5** LoRa Transmission Module

Each module interfaces with the controllers and gateway in the network via UART pins and GPIO pins. For module configuration and data transmission to/reception from the LoRa network, the UART pins TX and RX are connected to the controller's and gateway's RX and TX pins, respectively. As the module has a total of 4 different mode settings, the mode setting is driven in the output direction via two GPIO channels on both the controller and the gateway. Finally, after each operation performed on the module, the triggered busy signal (AUX) is driven via the GPIO input and monitored. The interfacing logic of the module is shown in Figure 6 in generic. And the physical connection setup between the sub-modules (including IMU and LoRa module) and MCU within the controller module is shown in Figure 7. The schematic of the controller indicating how the sub-modules interfaced via which pins is shown in Figure 8.
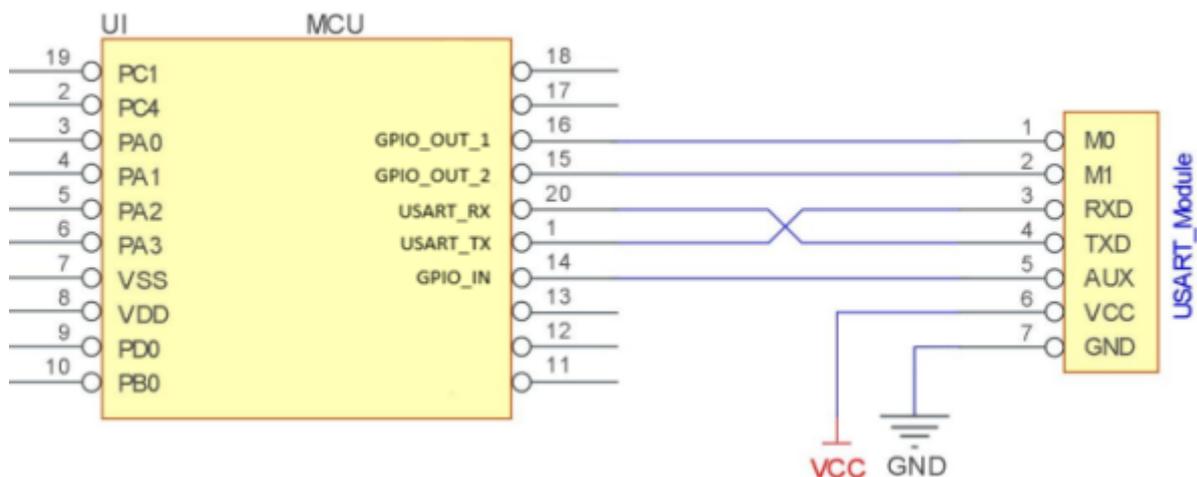


**Figure 6** Generic Interfacing Diagram between LoRa Transmission Modules and Controllers/Gateways
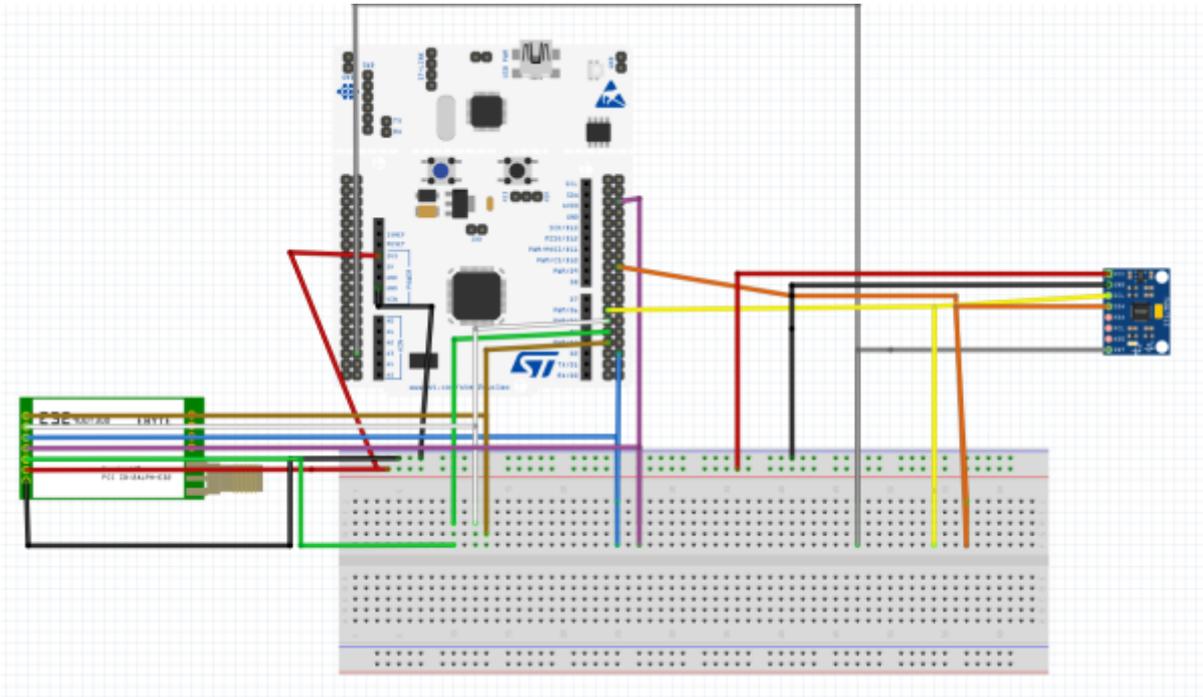
**Figure 7** Physical Connection Setup for the Controller (with IMU and LoRa module)
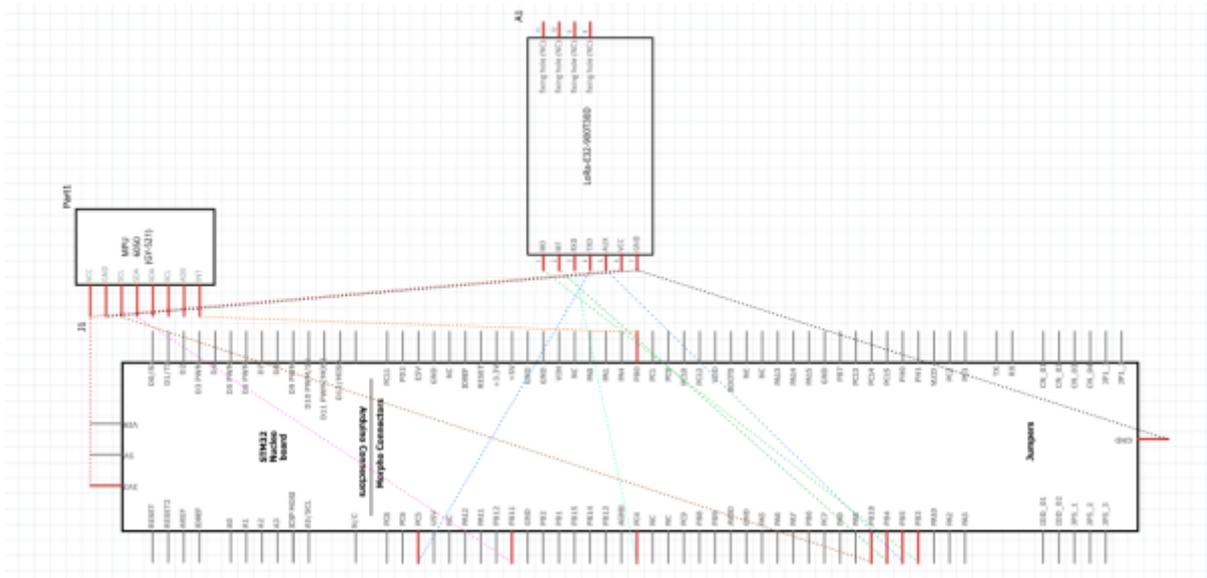


**Figure 8** Schematic Indicating the Connections of All Modules in the Controller Unit

### 2.3.2 Configurations

The system configuration has been implemented with the following parameters:

- **Air Data Rate:** 2.4 kbps
- **Transmission Power:** +30 dBm
- **UART Baud Rate:** 9600 bps
- **Transmission Mode:** Transparent
- **Frequency Band:** 868 MHz
- **FEC:** OFF

The 2.4 kbps air data rate was selected to increase data reliability and achieve a higher link budget. The low data rate ensures that the signal is transmitted with a longer symbol duration, thereby

increasing receiver sensitivity. The +30 dBm output power was chosen to maximise connection stability. This configuration ensures that the system provides stable, low-error-rate data transmission even at close range. The FEC is kept OFF since it can increase overhead which risks the frame loss on the gateway side due to latency during the transmission.

### 2.3.3 Communication Framing Standard

The communication protocol is based on the principle of window-based data transfer. The controller collects fixed-length data windows from the IMU and transmits them to the gateway unit via LoRa, packaged according to the specified framing standard. On the gateway side, the incoming frames are parsed and a new data window is created for use as input to the model.

Data transmission is performed in fixed-length frames. Each IMU window consists of 96 samples and 6 axes, with each axis containing 16-bit raw data. A window has a total size of 96 × 6 × 2 bytes.

The LoRa frame consists of the following fields:

- **ADDH** (1 byte, left 0 as the transmissions are transparent)
- **ADDL** (1 byte, left 0 as the transmissions are transparent)
- **CHAN** (1 byte, 862+OFFSET, set as 0x06 since 868 MHz is used)
- **MAGIC** (2 bytes, "TE")
- **CID** - Controller ID (2 bytes, controller identifier)
- **FID** – Frame ID (1 byte, index of the frame)
- **FC** – Frame Counter (1 byte, total frame count for the window to be sent)
- **IMU PAYLOAD** (48 bytes, 4*12 bytes of IMU packets)

Each window is divided into multiple frames and transmitted sequentially using the frame counter field. On the gateway side, the magic is checked for each frame then frames are counted to form a complete window. This structure ensures that synchronisation is maintained in the event of data loss. The communication flow is shown in Figure 9.
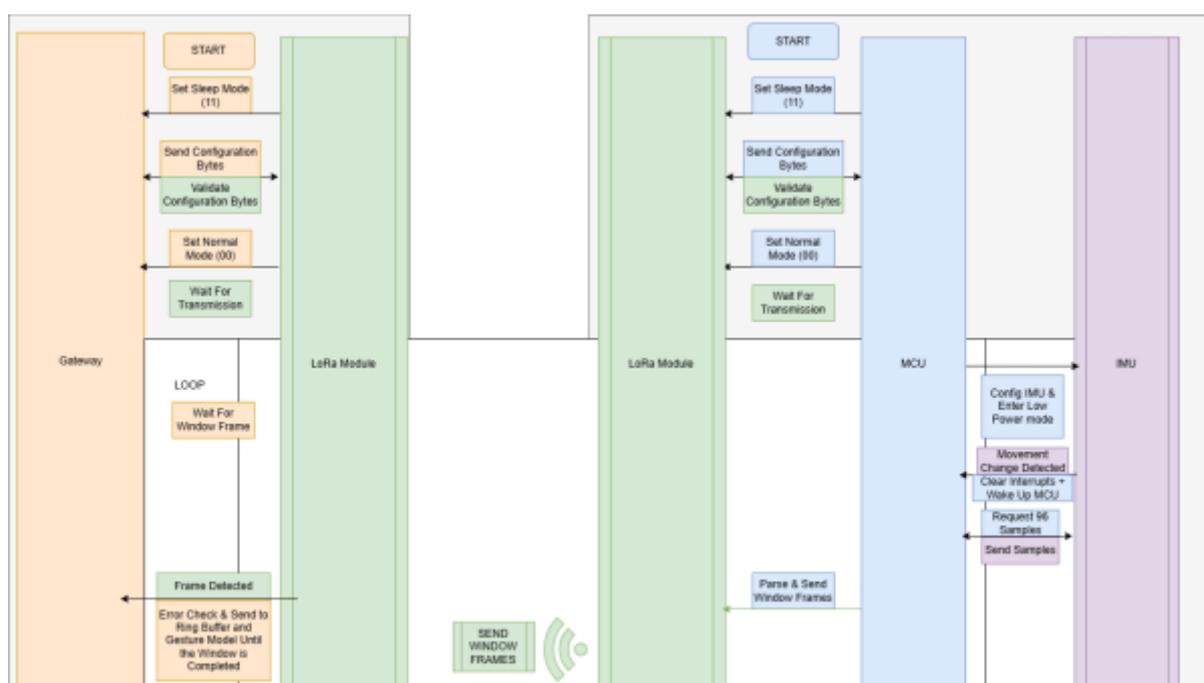


**Figure 9** Communication flow between the controllers & gateway in the same environment

## 2.4 Gateway

The gateway unit of the system operates with a Python-based software architecture and contains a script for each task. The main tasks of the Gateway are:

1. Receiving and buffering frames received from controllers via LoRa
2. Transmitting frames to the model and generating movement results by running the trained model
3. Transmitting the results to the cloud server

The Gateway represents the edge computing layer of the system.

### 2.4.1 Gateway System (RPi)

The Raspberry Pi 3 Model B platform has been selected as the gateway unit. This choice was made by balancing system requirements with hardware capabilities. The Raspberry Pi 3 offers a suitable platform for IoT edge computing applications due to its sufficient processing power, low cost, extensive software support, and integrated wireless connectivity features. The primary tasks of the gateway unit within the scope of the project are to receive data frames via LoRa, restructure this data into windows, run the trained TensorFlow Lite model, and transmit the results to the cloud server. As these processes can create a high processing load for microcontroller-level hardware, a more powerful single-board computer was preferred. The Raspberry Pi 3's 1.2 GHz ARM Cortex-A53 processor and 1 GB RAM capacity provide sufficient performance for both Python-based data processing and TFLite model execution.

Furthermore, the Raspberry Pi 3's built-in Wi-Fi and Ethernet interfaces enable a direct internet connection to be established with the cloud server. This allows gateway-server communication to be achieved without the need for additional communication modules. The easy integration of the LoRa module via serial connection through the USB ports also provides hardware flexibility. The physical connection setup for the gateway is shown in Figure 10. The schematic of the gateway indicating how the LoRa module is interfaced via which pins is shown in Figure 11.
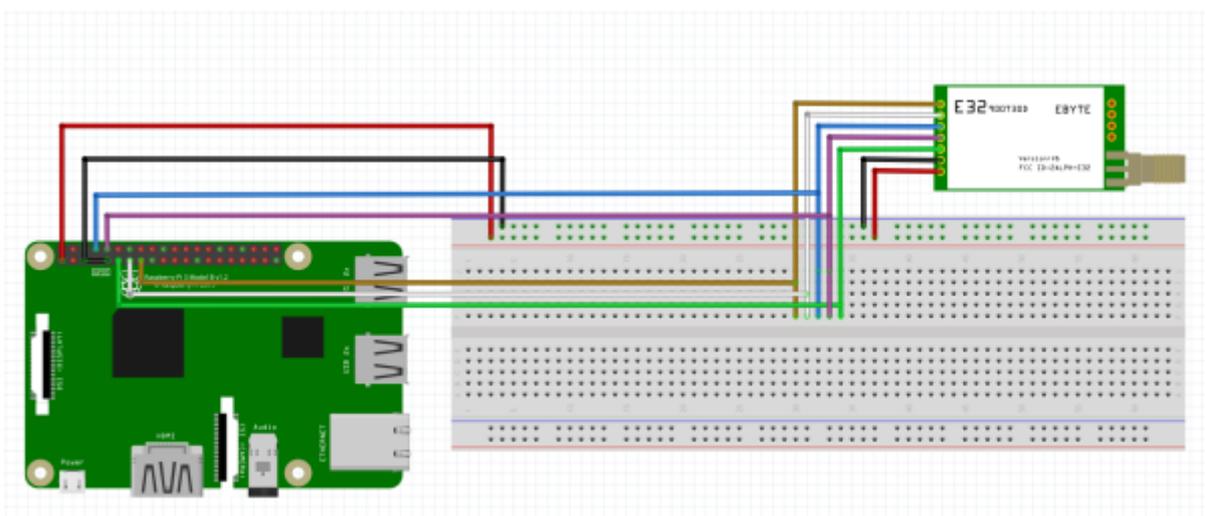


*Figure 10* Physical Connection Setup for the Gateway
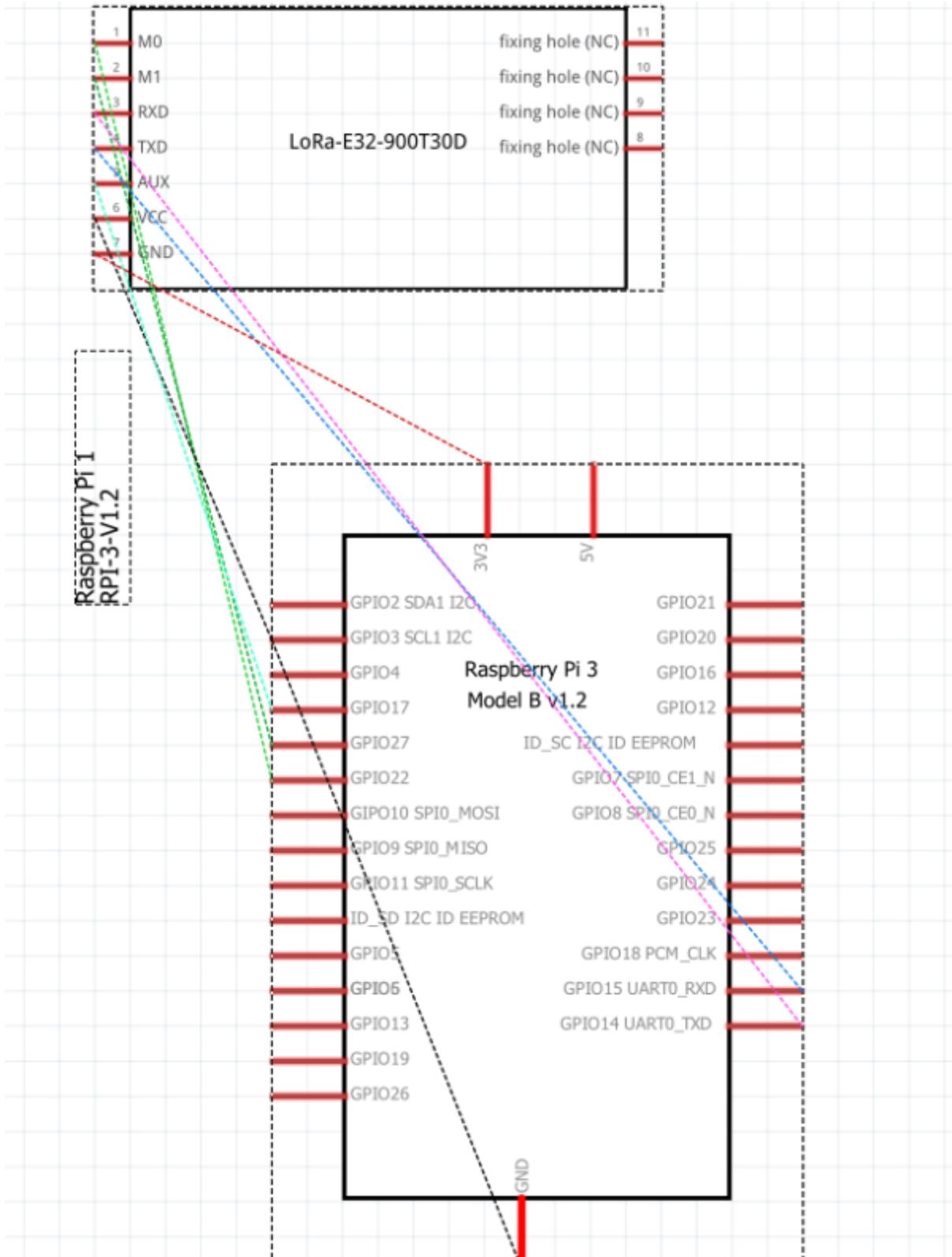
**Figure 11** Schematic Indicating the Connections of All Modules in the Gateway Unit

### 2.4.2 Gesture Model

The motion classification model has a three-class structure:

- **Move:** The animal's motion along the x, y, z axes with a specific acceleration
- **Rest:** The animal's resting state
- **Shake:** The animal's sudden turns, tremors, and shakes along the x, y, z axes

The dataset was created using an IMU sensor directly connected to a Raspberry Pi. Three separate CSV files were generated: Move.csv, Shake.csv, Rest.csv. Each data record was collected in 200 windows consisting of 96 samples. With Fs = 100 and for 150 seconds. The raw data was then normalised using global mean and standard deviation values. The normalisation parameters were stored in a separate JSON file.

The model was trained using Keras and then converted to TensorFlow Lite format. The model input size is (1, 96, 6) and uses softmax activation in the final layer. The output vector represents the probabilities [move, rest, shake] respectively.

During the training process, global z-score normalisation was applied. The training and validation datasets were separated then multi-class classification was performed using a softmax output layer. The model obtained as a result of training was converted to TensorFlow Lite format and optimised to run in real time on the gateway.

The model has been tested with data received via LoRa. In the real-time test, each window was fed into the model and the instantaneous class probabilities were observed in the terminal output. The model performance was evaluated in terms of accuracy, consistency and class separability.

## 2.5 Server

The cloud server forms the central data management layer of the system. IMU data processed and classified by the gateway unit is transferred to the cloud database via the internet and stored there. Thanks to this structure, system outputs are not only displayed locally but are also archived at a central point and made available to users. The cloud layer enables the system to become a scalable and remotely accessible IoT solution by providing data storage, real-time monitoring, and historical analysis capabilities. In this approach, processed model outputs are sent instead of raw sensor data, thereby reducing bandwidth usage and optimising the data transmission load.

### 2.5.1 Cloud Infrastructure

The system uses Firebase Realtime Database (RTDB) as its cloud infrastructure. This structure ensures that classification results from the gateway unit are stored centrally and in real time. Firebase RTDB is a NoSQL database that operates on a JSON-based tree structure, providing low latency and instant data synchronisation. Thanks to these features, it offers a suitable solution for real-time monitoring and remote access scenarios in IoT applications.

### 2.5.2 Gateway Communication

The gateway unit transmits the classification output generated by the TensorFlow Lite model to the Firebase database via Wi-Fi over the internet. Communication is performed using REST-based HTTP requests. Each data transmission sends the following information to the server:

- **move_prob:** Rate of moving probability of the animal
- **rest_prob:** Rate of resting probability of the animal
- **shake_prob:** Rate of shaking probability of the animal
- **predicted:** The class with the highest probability which indicates the current behavioral situation of the animal

- **timestamp:** time of the behaviour

This structure ensures that only processed result data is sent to the cloud, not raw IMU data. This reduces bandwidth usage and makes the system more scalable. The cloud server panel interface and general tree structure of the database is shown in Figure 12.

```
https://xiao-pet-tracker-controller-default-rtdb.europe-west1.firebasedatabase.app/
▼ — controllers
    ▼ — 1
        —— move_prob: 0.007109028287231922
        —— predicted: "rest"
        —— rest_prob: 0.9696944952011108
        —— shake_prob: 0.02319643273949623
        —— timestamp: "1772039883.4974985"
```

**Figure 12** Firebase Server Interface and Organization of the Database

### 2.5.3 Real-Time Database Management

A separate node structure has been created for each device within the Realtime Database. This organisational structure enables multi-device support and ensures that data is logically separated. As data is stored with a timestamp, retrospective analysis and performance evaluation can be carried out. The server layer performs three key functions within the system:

- **Real-time monitoring:** Instant display of model outputs.
- **Data archiving:** Storage of past classification results.
- **Centralised access:** Different clients (web panel, mobile application, etc.) can access the same data.

With this architecture, the system has evolved from being merely a local motion detection platform into a cloud-integrated, traceable, and scalable IoT solution.

# 3. Results

## 3.1 Communication Configurations

The LoRa modules used within the system have undergone the same initialisation and configuration process on both the controller (STM32) side and the gateway (Raspberry Pi) side. This process was implemented to prevent parameter incompatibility between the two ends and to verify that the module is operating stably before communication begins. After powering the module, it was first expected to enter the idle state via the AUX pin. Configuration bytes were not sent until the AUX line reached logic level 1. This mechanism ensured that the module completed its internal processing. Once in the ready state, the module was placed into configuration (sleep/program) mode via the M0 and M1 pins. In configuration mode, parameter bytes were sent via UART. UART communication was performed at 9600 baud rate, 8 data bits, no parity, and 1 stop bit (8N1) format. The behavioral

characteristics of AUX pin is illustrated in Figure 13 (for module to controller communication) and Figure 14 (for controller to module communication).
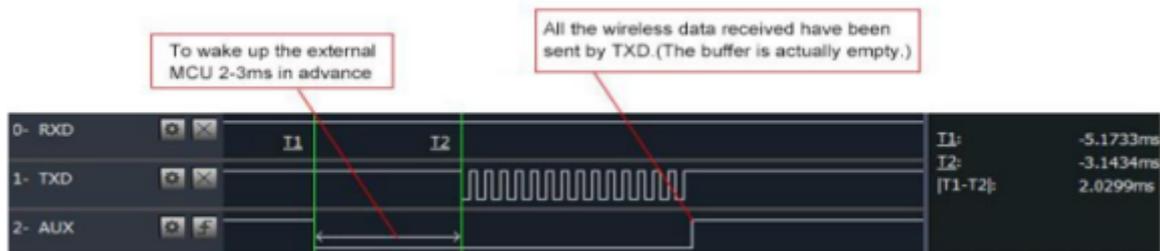


**Figure 13** Timing Sequence Diagram of AUX when TXD pin of the LoRa module transmits
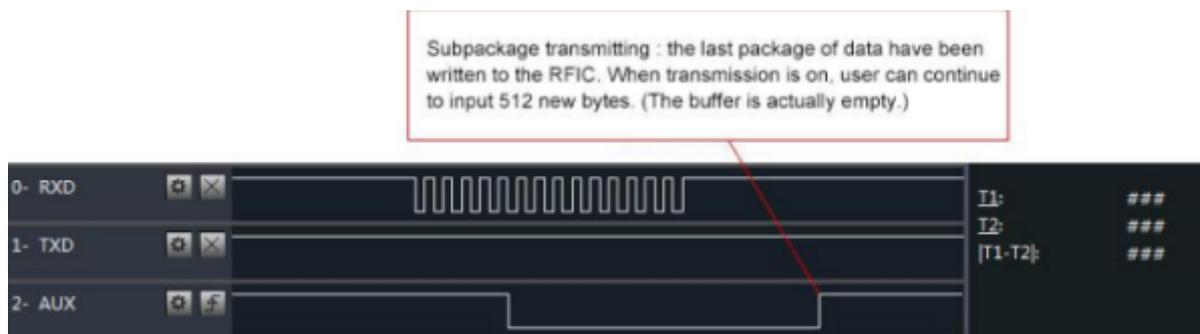


**Figure 14** Timing Sequence Diagram of AUX when RXD pin of the LoRa module receives

On the controller side, after the LoRa configuration bytes (0xC0 0x00 0x00 0x1A 0x06 0x40) were sent via the UART bus, it was checked whether the bytes sent from the RX channel were received back (echo test), thereby aiming to validate that the module configurations were successfully completed. In the STM32CubeIDE debug screen, by observing the UART RX line of the LoRa module, and it was validated that the configuration packets sent from the TX line were successfully received on the RX side. This completes the LoRa configurations on the controller side (Figure 15).

controller_main_func_init_block

```
//....

int main(void)
{

  /* USER CODE BEGIN 1 */
  uint8_t lora_params[8], tx_buf[LoRa_MAX_DATA_SIZE], rx_buf[64];
  uint32_t cur_tick;
  HAL_StatusTypeDef status;
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------
-------*/

  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();
```

```c
  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART3_UART_Init();

  /* USER CODE BEGIN 2 */
  MX_I2C2_Init();
  memset(tx_buf, 0, LoRa_MAX_DATA_SIZE);
  memset(rx_buf, 0, 64);

  //Wait initially until the controller io transactions is available
  cur_tick = HAL_GetTick();
  while(((HAL_GetTick() - cur_tick) < 1000) && HAL_GPIO_ReadPin(GPIOB,
LoRa_AUX_Pin) != GPIO_PIN_SET); //Wait until the AUX pin is set

  //If controller is busy and exceeds the timeout, blink the led then
abort
  if(HAL_GPIO_ReadPin(GPIOB, LoRa_AUX_Pin) != GPIO_PIN_SET){
      HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
      Error_Handler();
  }
  else{
      //Set the initial lora module mode [11 -> Sleep]
      HAL_GPIO_WritePin(GPIOB, LoRa_M0_Pin, GPIO_PIN_SET); //M0
      HAL_GPIO_WritePin(GPIOB, LoRa_M1_Pin, GPIO_PIN_SET); //M1
      HAL_Delay(20);

      //Wait until the controller is available
      cur_tick = HAL_GetTick();
      while(((HAL_GetTick() - cur_tick) < 100) &&
HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_4) != GPIO_PIN_SET); //Wait until the
AUX pin is set

      //Clear buffer & send the initial option parameters
      memset(lora_params, 0, 8);

      //Save after shutdown (through command flash interface)
      lora_params[0] = 0xC0;

      //Broadcast transmission (NC address)
```

```c
        lora_params[1] = 0x00;
        lora_params[2] = 0x00;

        //UART PARITY: 8N1, BAUD: 9600bps, DATA RATE (SF): SF7 / 2.4k,
        lora_params[3] = 0x1A; //0b00011010

        //Channel settings: 868 MHz (862 + OFFSET->6)
        lora_params[4] = 0x06;

        //Options: Transparent Transmission, IO Push-Pull, Wireless
        wakeup: 250ms, FEC OFF, Power: 30 dBm
        lora_params[5] = 0x40; //0b0100 0000;

        //Send the option bytes (wait for the AUX pin after the
        transaction)
        if((status = HAL_UART_Transmit(&huart3, lora_params, 6, 100)) ==
        HAL_OK){
            //HAL_Delay(20);
            if((status = HAL_UART_Receive(&huart3, rx_buf, 6, 1000)) ==
            HAL_OK){
                HAL_Delay(50);
                //Check if the params sent equal to the echo packet sent back
                if(strncmp((const char*)rx_buf, (const char*)lora_params,
                6) == 0){
                    //Set the normal mode and get ready for the
                    operations
                    HAL_GPIO_WritePin(GPIOB, LoRa_M0_Pin,
                    GPIO_PIN_RESET); //M0
                    HAL_GPIO_WritePin(GPIOB, LoRa_M1_Pin,
                    GPIO_PIN_RESET); //M1
                    memset(rx_buf, 0, 64); //Clear RX buffer
                    HAL_Delay(1);
                    //Initialize the MPU
                    int ret = mpu6050_init(&hi2c2);
                    if (ret != 0) Error_Handler();
                    HAL_Delay(1);
                }
                else{
                    HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
                    Error_Handler();
                }

            }
            else{
                HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
                Error_Handler();
            }
        }
        else{
            HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
            Error_Handler();
```

```
        }
    }

    //....
}

//....
```



**Figure 15** Debug Expression of the LoRa Configuration bytes transmission on the Controller Side

The same process was carried out on the gateway side with exactly the same configuration parameters. The sent parameters were printed to the terminal, and then the configuration packet echo was checked by reading the UART line simultaneously. The incoming echo packet was checked to see if it matched the configuration sent and printed to the terminal. It was found that the configuration and the echo packet were identical, and the configuration process for the LoRa module on the gateway side was also successfully completed (Figure 16).

gateway_main_func_init_block

```python
#....
def main():
    #Initialize LoRa control ports (M0-M1, AUX)
        GPIO.setmode(GPIO.BOARD)
    GPIO.setwarnings(False)
    GPIO.setup(PIN_LORA_M0, GPIO.OUT)
    GPIO.setup(PIN_LORA_M1, GPIO.OUT)
    GPIO.setup(PIN_LORA_AUX, GPIO.IN, pull_up_down = GPIO.PUD_UP)
```

```python
    #Wait until AUX is high (Module is ready)
    if not wait_until_pin(PIN_LORA_AUX, GPIO.HIGH, 1.0):
        raise_error("LoRa module is busy!", -1, cleanup())
    else:
        print("LoRa module is available")

    #Set module mode (11) sleep
    GPIO.output(PIN_LORA_M0, GPIO.HIGH)
    GPIO.output(PIN_LORA_M1, GPIO.HIGH)
    if not wait_until_pin(PIN_LORA_AUX, GPIO.HIGH, 1.0):
        raise_error("Cannot set parameters", -1, cleanup())
    else:
        print("Parameters are set")

    #Send lora configuration parameters
    uart.reset_input_buffer()
    print("TX:", lora_params.hex(" "))
    uart.write(lora_params)
    uart.flush()
    time.sleep(0.1)

    #Validate lora configuration parameters
    uart.reset_input_buffer()
    print("TX:", cmd_params.hex(" "))
    uart.write(cmd_params)
    uart.flush()
    rx = uart.read(6)
    if rx != lora_params:
        raise_error("Unable to validate configuration parameters", -1,
cleanup())
    else:
        print("Config parameters:", rx.hex(" "))
        time.sleep(0.1)

    #Set module mode (00) transceiver
    if not wait_until_pin(PIN_LORA_AUX, GPIO.HIGH, 1.0):
        raise_error("LoRa module is busy!", -1, cleanup())
    else:
        print("LoRa module is available to transmit")
    GPIO.output(PIN_LORA_M0, GPIO.LOW)
    GPIO.output(PIN_LORA_M1, GPIO.LOW)
    time.sleep(0.01)

    #....

main()

if _name_ == "__main__":
    main()
```

**Figure 16** Terminal Output Indicating LoRa Configuration bytes transmission on the Gateway Side

## 3.2 Window Generation

The controller (STM32) unit in the system architecture has been designed using an event-triggered approach. When the MPU6050 sensor detects movement exceeding the specified threshold, it generates a hardware rising-edge interrupt signal. This signal is connected to the microcontroller's external interrupt (EXTI) line. The microcontroller normally waits in low-power mode (STOP mode) whose functionality is given below and is put to sleep with the WFI instruction.

enter_stop_mode

```c
//....
static void enter_stop_mode(void)
{
  // Clear pending EXTI to avoid immediate wake-up
  __HAL_GPIO_EXTI_CLEAR_IT(MPU_INT_Pin);

  HAL_SuspendTick();
  HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
  HAL_ResumeTick();

  //After STOP, clocks must be restored
  SystemClock_Config();

  //Re enable the interfaces being used
  MX_USART3_UART_Init();
  MX_I2C2_Init();
}

//....
```

In the main loop block of the controller, of which software architecture is built on the super-loop design, the controller waits for the IMU event to wake up its CPU. After the CPU wakes up by the waker interrupt routine, controller checks if any request from the gateway and the motion itself keep existing. If the condition is satisfied, the window containing 96 samples are collected and sent. Otherwise it skips the turn and gets back to sleep.

waker_callback

```c
volatile uint8_t motion_pending = 0; //Generic motion indicator field
                                     //for low-power waker callback
```

```c
//...

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == MPU_INT_Pin) { //PB0
    motion_pending = 1; //Set motion indicator
  }
}

//....
```

main_super_loop_flow_design

```c
volatile uint8_t motion_pending = 0; //Generic motion indicator field
for low-power waker callback

//...
/* Infinite main loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE BEGIN 3 */
    // If there is no pending motion and no incoming request, go to
STOP mode
    if (!motion_pending)
    {
      // Try to catch a request quickly; if none, sleep.
      if (HAL_UART_Receive(&huart3, rx_buf, 1, 50) != HAL_OK)
      {
        enter_stop_mode();
        continue; // after wake-up, loop again
      }
    }

    //Wait for the window request from the gateway and any motion
    if (HAL_UART_Receive(&huart3, rx_buf, 1, 500) == HAL_OK
          && rx_buf[0] == 0x01 && motion_pending)
    {
        // Clear the motion flag now
        motion_pending = 0;

        // Prepare response window, then send
        tx_buf[0] = 0x00;
        tx_buf[1] = 0x00;
        tx_buf[2] = 0x06;

        if (imu_send_window(&hi2c2, &huart3, tx_buf,
LoRa_MAX_DATA_SIZE, 3, 96, 10) == 0)
        {
```

```
                HAL_GPIO_TogglePin(GPIOA, USER_LED_Pin);
            }
        }
        // After sending, go back to STOP quickly (next loop iteration
will sleep)
    }
//....
```

The controller then collected a fixed-length IMU data window consisting of a total of 96 samples at a sampling frequency of 100 Hz on the gateway side (a bit lower considering the transmission latency) Each sample contained a total of six 16-bit raw measurements: three-axis acceleration and three-axis gyroscope data. Thus, a single window produced 96 × 6 × 2 bytes of raw data.

The collected window data was not sent in a single piece but was divided into fixed-length frames, taking into account the maximum packet size of the LoRa module. Each frame consisted of a header and a payload section. The header section contained address information, channel information, the system-defined "MAGIC" constant value, and a frame counter field. The window data was divided into multiple frames, and each frame was transmitted sequentially with the frame counter field. By this flow, full IMU logic is code bases the measurement and windowing logic is given below.

imu.h

```
/*
 * imu.h
 *
 *  Created on: Feb 3, 2026
 *      Author: Tarik Esen
 */

#ifndef INC_IMU_H_
#define INC_IMU_H_

#include "stm32f0xx_hal.h"
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include "mpu6050.h"

#define MAX_WINDOW_SIZE 255

int imu_send_window(I2C_HandleTypeDef*, UART_HandleTypeDef*, uint8_t*,
uint8_t, uint8_t, uint8_t, int);

#endif /* INC_IMU_H_ */
```

imu.c

```
/*
 * imu.c
```

```c
 *
 *  Created on: Feb 3, 2026
 *      Author: Tarik Esen
 */

#include "imu.h"

static int imu_collect_measure(I2C_HandleTypeDef*, int, uint8_t*);

int imu_send_window(I2C_HandleTypeDef* hi2c, UART_HandleTypeDef* huart,
uint8_t* uart_tx_buf, uint8_t buf_size, uint8_t param_size, uint8_t
window_size, int wait_per_frame){
    uint8_t measure_buf[MAX_WINDOW_SIZE*12];
    uint8_t ret_val;
    uint8_t sample_per_frame;
    int sample_index=0, frame_index;

    //Magic header
    memcpy((void*)(uart_tx_buf + param_size), (const void
*__restrict)"TE", 2);

    //Number of frames
    sample_per_frame = (buf_size-(param_size+4))/(12); //Sample per
frame
    uart_tx_buf[param_size + 2] = (window_size/sample_per_frame) +
(window_size%sample_per_frame ? 1:0); //Number of frames

    //Proceed with measurements until the amount enough to fill the
window is reached
    ret_val = imu_collect_measure(hi2c, window_size, measure_buf);
    if(ret_val == 0){
        //Start UART stream after the measurements are collected
        for(int i=0; i<uart_tx_buf[param_size + 2]; i++){
            uart_tx_buf[param_size + 3] = i; //Frame number

            //Fill the rest of the frame with enough number of samples
            frame_index = 0;
            while(sample_index < window_size && frame_index < 4){
                memcpy(&(uart_tx_buf[param_size + 4 + 12*frame_index]),
&(measure_buf[12*sample_index]), 12);
                sample_index++;
                frame_index++;
            }

            //Send the frame through UART bus
            if(HAL_UART_Transmit(huart, uart_tx_buf, buf_size, 100) ==
HAL_OK){
                //Wait until the UART module is ready
                HAL_Delay(wait_per_frame);
            }
        }
```

```c
    }
    else return ret_val;

    return 0;
}

//Bulk measure as many samples as passed window size
static int imu_collect_measure(I2C_HandleTypeDef* hi2c, int
measure_count, uint8_t* measure_buf){
    mpu6050_raw_t imu_r;

    if(measure_count > (MAX_WINDOW_SIZE))
        return -1;
    else if(measure_buf == NULL)
        return -2;

    for(int i=0; i < measure_count; i++){
        if (mpu6050_read_raw(hi2c, &imu_r) == 0) {
            memcpy(&(measure_buf[i*12]), &imu_r, 12);
        }
        else return -3;
    }

    return 0;
}
```

After these logics are merged in addition with MPU control middleware, the controller software implementation is shaped its final release form.

mpu6050.h

```c
/*
 * mpu6050.h
 *
 *  Created on: Feb 3, 2026
 *      Author: Tarik Esen
 */



#ifndef INC_MPU6050_H_
#define INC_MPU6050_H_

#define MPU_ADDR        (0x68 << 1) // If AD0=0 0x68, AD0=1 0x69
#define REG_WHO_AM_I    0x75
#define REG_PWR_MGMT_1  0x6B
#define REG_SMPLRT_DIV  0x19
#define REG_CONFIG      0x1A
#define REG_GYRO_CONFIG 0x1B
#define REG_ACCEL_CONFIG 0x1C
#define REG_ACCEL_XOUT_H 0x3B
```

```c
#include "stm32f0xx_hal.h"
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#pragma pack(1)
typedef struct {
    int16_t ax, ay, az;
    int16_t gx, gy, gz;
    int16_t temp;
} mpu6050_raw_t;

typedef struct {
    float ax_g, ay_g, az_g;
    float gx_dps, gy_dps, gz_dps;
    float temp_c;
} mpu6050_scaled_t;
#pragma pack(0)

int mpu6050_init(I2C_HandleTypeDef*);
int mpu6050_read_raw(I2C_HandleTypeDef*, mpu6050_raw_t *);
void mpu6050_scale(const mpu6050_raw_t *, mpu6050_scaled_t *);
#endif /* INC_MPU6050_H_ */
```

## mpu6050.c

```c
/*
 * mpu6050.c
 *
 *  Created on: Feb 3, 2026
 *      Author: Tarik Esen
 */

#include "mpu6050.h"

static HAL_StatusTypeDef mpu_read(I2C_HandleTypeDef*, uint8_t, uint8_t *, uint16_t);
static HAL_StatusTypeDef mpu_write(I2C_HandleTypeDef*, uint8_t, uint8_t);

// 0:OK, <0 ERROR
int mpu6050_init(I2C_HandleTypeDef* hi2c)
{
    uint8_t who = 0;
    if (mpu_read(hi2c, REG_WHO_AM_I, &who, 1) != HAL_OK) return -1;
    if (who != 0x68) return -2;

    // Wake up (disable sleep bit)
    if (mpu_write(hi2c, REG_PWR_MGMT_1, 0x00) != HAL_OK) return -3;
```

```c
    HAL_Delay(50);

    // Sample rate = Gyro output / (1 + SMPLRT_DIV). Gyro output
default 8kHz (DLPF=0) or 1kHz (DLPF!=0)
    mpu_write(hi2c, REG_SMPLRT_DIV, 0x07);    // 1kHz/(1+7)=125Hz (DLPF
ON is assumed)
    mpu_write(hi2c, REG_CONFIG, 0x03);        // DLPF ~44Hz (typical)
    mpu_write(hi2c, REG_GYRO_CONFIG, 0x00);  // ±250 dps
    mpu_write(hi2c, REG_ACCEL_CONFIG, 0x00); // ±2g

    return 0;
}


// 0:OK, <0 ERROR
int mpu6050_read_raw(I2C_HandleTypeDef* hi2c, mpu6050_raw_t *out)
{
    uint8_t b[14];
    if (mpu_read(hi2c, REG_ACCEL_XOUT_H, b, 14) != HAL_OK) return -1;

    out->ax   = (int16_t)((b[0] << 8) | b[1]);
    out->ay   = (int16_t)((b[2] << 8) | b[3]);
    out->az   = (int16_t)((b[4] << 8) | b[5]);
    out->temp = (int16_t)((b[6] << 8) | b[7]);
    out->gx   = (int16_t)((b[8] << 8) | b[9]);
    out->gy   = (int16_t)((b[10] << 8) | b[11]);
    out->gz   = (int16_t)((b[12] << 8) | b[13]);

    return 0;
}

void mpu6050_scale(const mpu6050_raw_t *r, mpu6050_scaled_t *s)
{
    s->ax_g = r->ax / 16384.0f;
    s->ay_g = r->ay / 16384.0f;
    s->az_g = r->az / 16384.0f;

    s->gx_dps = r->gx / 131.0f;
    s->gy_dps = r->gy / 131.0f;
    s->gz_dps = r->gz / 131.0f;

    s->temp_c = (r->temp / 340.0f) + 36.53f;
}


//I2C bus read/write functions
static HAL_StatusTypeDef mpu_read(I2C_HandleTypeDef* hi2c, uint8_t reg,
uint8_t *buf, uint16_t len)
{
    return HAL_I2C_Mem_Read(hi2c, MPU_ADDR, reg, I2C_MEMADD_SIZE_8BIT,
buf, len, 200);
}
```

```c
static HAL_StatusTypeDef mpu_write(I2C_HandleTypeDef* hi2c, uint8_t
reg, uint8_t val)
{
    return HAL_I2C_Mem_Write(hi2c, MPU_ADDR, reg, I2C_MEMADD_SIZE_8BIT,
&val, 1, 200);
}
```

## main.h

```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.h
  * @brief          : Header for main.c file.
  *                   This file contains the common defines of the
application.
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2025 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the
LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Define to prevent recursive inclusion -----------------------------
-------*/
#ifndef __MAIN_H
#define __MAIN_H

#ifdef __cplusplus
extern "C" {
#endif

/* Includes ----------------------------------------------------------
-------*/
#include "stm32f0xx_hal.h"

/* Private includes --------------------------------------------------
-------*/
/* USER CODE BEGIN Includes */
```

```c
#include <stdlib.h>
#include <string.h>
#include "imu.h"
/* USER CODE END Includes */

/* Exported types ------------------------------------------------------
-------*/
/* USER CODE BEGIN ET */

/* USER CODE END ET */

/* Exported constants --------------------------------------------------
-------*/
/* USER CODE BEGIN EC */

/* USER CODE END EC */

/* Exported macro ------------------------------------------------------
-------*/
/* USER CODE BEGIN EM */

/* USER CODE END EM */

/* Exported functions prototypes ---------------------------------------
-------*/
void Error_Handler(void);

/* USER CODE BEGIN EFP */

/* USER CODE END EFP */

/* Private defines -----------------------------------------------------
-------*/
#define USER_LED_Pin GPIO_PIN_5
#define USER_LED_GPIO_Port GPIOA
#define LoRa_M0_Pin GPIO_PIN_3
#define LoRa_M0_GPIO_Port GPIOB
#define LoRa_M1_Pin GPIO_PIN_4
#define LoRa_M1_GPIO_Port GPIOB
#define LoRa_AUX_Pin GPIO_PIN_5
#define LoRa_AUX_GPIO_Port GPIOB
#define LoRa_MAX_DATA_SIZE 58
#define MPU_INT_Pin GPIO_PIN_0
#define MPU_INT_Port GPIOB
#define MPU_INT_Line (1u << MPU_INT_Pin)
/* USER CODE BEGIN Private defines */

/* USER CODE END Private defines */

#ifdef __cplusplus
}
```

```
#endif

#endif /* __MAIN_H */
```

## main.c

```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c - CONTROLLER
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2025 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the
LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */
```

```c
/* Private macro -----------------------------------------------------
-------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables -------------------------------------------------
-------*/
UART_HandleTypeDef huart3;
I2C_HandleTypeDef hi2c2;

/* USER CODE BEGIN PV */
volatile uint8_t motion_pending = 0; //Generic motion indicator field
for the waker and loop
/* USER CODE END PV */

/* Private function prototypes ---------------------------------------
-------*/
static void enter_stop_mode(void); //Routine to sleep CPU
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART3_UART_Init(void);
static void MX_I2C2_Init(void);

/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code -------------------------------------------------
-------*/
/* USER CODE BEGIN 0 */

//Low-power waker callback
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == MPU_INT_Pin) { //PB0
    motion_pending = 1; //Set motion indicator
  }
}
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  /* USER CODE BEGIN 1 */
  uint8_t lora_params[8], tx_buf[LoRa_MAX_DATA_SIZE], rx_buf[64];
  uint32_t cur_tick;
```

```c
  HAL_StatusTypeDef status;
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------
-------*/

  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART3_UART_Init();

  /* USER CODE BEGIN 2 */
  MX_I2C2_Init();
  memset(tx_buf, 0, LoRa_MAX_DATA_SIZE);
  memset(rx_buf, 0, 64);

  //Wait initially until the controller io transactions is available
  cur_tick = HAL_GetTick();
  while(((HAL_GetTick() - cur_tick) < 1000) && HAL_GPIO_ReadPin(GPIOB,
LoRa_AUX_Pin) != GPIO_PIN_SET); //Wait until the AUX pin is set

  //If controller is busy and exceeds the timeout, blink the led then
abort
  if(HAL_GPIO_ReadPin(GPIOB, LoRa_AUX_Pin) != GPIO_PIN_SET){
      HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
      Error_Handler();
  }
  else{
      //Set the initial lora module mode [11 -> Sleep]
      HAL_GPIO_WritePin(GPIOB, LoRa_M0_Pin, GPIO_PIN_SET); //M0
      HAL_GPIO_WritePin(GPIOB, LoRa_M1_Pin, GPIO_PIN_SET); //M1
      HAL_Delay(20);

      //Wait until the controller is available
      cur_tick = HAL_GetTick();
      while(((HAL_GetTick() - cur_tick) < 100) &&
HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_4) != GPIO_PIN_SET); //Wait until the
```

```c
AUX pin is set

    //Clear buffer & send the initial option parameters
    memset(lora_params, 0, 8);

    //Save after shutdown (through command flash interface)
    lora_params[0] = 0xC0;

    //Broadcast transmission (NC address)
    lora_params[1] = 0x00;
    lora_params[2] = 0x00;

    //UART PARITY: 8N1, BAUD: 9600bps, DATA RATE (SF): SF7 / 2.4k,
    lora_params[3] = 0x1A; //0b00011010

    //Channel settings: 868 MHz (862 + OFFSET->6)
    lora_params[4] = 0x06;

    //Options: Transparent Transmission, IO Push-Pull, Wireless
wakeup: 250ms, FEC OFF, Power: 30 dBm
    lora_params[5] = 0x40; //0b0100 0000;

    //Send the option bytes (wait for the AUX pin after the
transaction)
    if((status = HAL_UART_Transmit(&huart3, lora_params, 6, 100)) ==
HAL_OK){
        if((status = HAL_UART_Receive(&huart3, rx_buf, 6, 1000)) ==
HAL_OK){
            HAL_Delay(50);
            //Check if the params sent equal to the echo packet sent
back
            if(strncmp((const char*)rx_buf, (const char*)lora_params,
6) == 0){
                //Set the normal mode and get ready for the
operations
                HAL_GPIO_WritePin(GPIOB, LoRa_M0_Pin,
GPIO_PIN_RESET); //M0
                HAL_GPIO_WritePin(GPIOB, LoRa_M1_Pin,
GPIO_PIN_RESET); //M1
                memset(rx_buf, 0, 64);
                HAL_Delay(1);
                int ret = mpu6050_init(&hi2c2);
                if (ret != 0) Error_Handler();
                HAL_Delay(1);
            }
            else{
                HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
                Error_Handler();
            }
        }
        else{
```

```c
                HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
                Error_Handler();
            }
        }
        else{
            HAL_GPIO_WritePin(GPIOA, USER_LED_Pin, GPIO_PIN_SET);
            Error_Handler();
        }
    }
    /* USER CODE END 2 */

    /* Infinite loop */
    /* USER CODE BEGIN WHILE */
    while (1)
    {
    /* USER CODE BEGIN 3 */
    // If there is no pending motion and no incoming request, go to
STOP mode
    if (!motion_pending)
    {
        // Try to catch a request quickly; if none, sleep.
        if (HAL_UART_Receive(&huart3, rx_buf, 1, 50) != HAL_OK)
        {
            enter_stop_mode();
            continue; // after wake-up, loop again
        }
    }

    //Wait for the window request from the gateway and any motion
    if (HAL_UART_Receive(&huart3, rx_buf, 1, 500) == HAL_OK
            && rx_buf[0] == 0x01 && motion_pending)
    {
        //Clear the motion flag
        motion_pending = 0;

        // Prepare response window, then send
        tx_buf[0] = 0x00;
        tx_buf[1] = 0x00;
        tx_buf[2] = 0x06;

        if (imu_send_window(&hi2c2, &huart3, tx_buf,
LoRa_MAX_DATA_SIZE, 3, 96, 10) == 0)
        {
            HAL_GPIO_TogglePin(GPIOA, USER_LED_Pin);
        }
    }
    // After sending, go back to STOP quickly (next loop iteration
will sleep)
    }
    /* USER CODE END 3 */
}
```

```c
/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the RCC Oscillators according to the specified
parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL12;
  RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) !=
HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief USART3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART3_UART_Init(void)
{

  /* USER CODE BEGIN USART3_Init 0 */
```

```c
  /* USER CODE END USART3_Init 0 */

  /* USER CODE BEGIN USART3_Init 1 */

  /* USER CODE END USART3_Init 1 */
  huart3.Instance = USART3;
  huart3.Init.BaudRate = 9600;
  huart3.Init.WordLength = UART_WORDLENGTH_8B;
  huart3.Init.StopBits = UART_STOPBITS_1;
  huart3.Init.Parity = UART_PARITY_NONE;
  huart3.Init.Mode = UART_MODE_TX_RX;
  huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart3.Init.OverSampling = UART_OVERSAMPLING_16;
  huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart3) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART3_Init 2 */

  /* USER CODE END USART3_Init 2 */
}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */

  /* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(USER_LED_GPIO_Port, USER_LED_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, LoRa_M0_Pin|LoRa_M1_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : USER_LED_Pin */
  GPIO_InitStruct.Pin = USER_LED_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
```

```c
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(USER_LED_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : LoRa_M0_Pin LoRa_M1_Pin */
  GPIO_InitStruct.Pin = LoRa_M0_Pin|LoRa_M1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*Configure GPIO pin : LoRa_AUX_Pin */
  GPIO_InitStruct.Pin = LoRa_AUX_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(LoRa_AUX_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : MPU_INT_Pin */
  GPIO_InitStruct.Pin  = MPU_INT_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;    //Rising-edge
interrupt
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;          //IDLE low, PULSE high
  HAL_GPIO_Init(MPU_INT_Port, &GPIO_InitStruct);

  HAL_NVIC_SetPriority(EXTI0_1_IRQn, 1, 0);
  HAL_NVIC_EnableIRQ(EXTI0_1_IRQn);

  /* USER CODE BEGIN MX_GPIO_Init_2 */

  /* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
static void MX_I2C2_Init(void)
{
  hi2c2.Instance = I2C2;
  hi2c2.Init.Timing = 0x20303E5D;

  hi2c2.Init.OwnAddress1 = 0;
  hi2c2.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c2.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c2.Init.OwnAddress2 = 0;
  hi2c2.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
  hi2c2.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c2.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;

  if (HAL_I2C_Init(&hi2c2) != HAL_OK)
  {
    Error_Handler();
  }
```

```c
  // Analog filter Configuration
  if (HAL_I2CEx_ConfigAnalogFilter(&hi2c2, I2C_ANALOGFILTER_ENABLE) !=
HAL_OK)
  {
    Error_Handler();
  }

  // Digital filter Configuration
  if (HAL_I2CEx_ConfigDigitalFilter(&hi2c2, 0) != HAL_OK)
  {
    Error_Handler();
  }
}

static void enter_stop_mode(void)
{
  // Clear pending EXTI to avoid immediate wake-up
  __HAL_GPIO_EXTI_CLEAR_IT(MPU_INT_Pin);

  HAL_SuspendTick();
  HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
  HAL_ResumeTick();

  //After STOP, clocks must be restored
  SystemClock_Config();

  //Re enable the interfaces being used
  MX_USART3_UART_Init();
  MX_I2C2_Init();
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
#ifdef USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line
number
```

```c
    *             where the assert_param error has occurred.
    * @param   file: pointer to the source file name
    * @param   line: assert_param error line source number
    * @retval None
    */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and
line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

Data received via the serial port on the gateway side was read at the byte level. Upon receiving each frame, the first 4-byte address and channel field were first checked, followed by verification of the MAGIC field. If the MAGIC field did not match the expected value, the frame was deemed invalid and synchronisation was maintained. For valid frames, the frame counter field was checked and the window data was correctly reconstructed in sequence. This control mechanism was applied to each frame, preventing incomplete or corrupted frames from disrupting window integrity. When all frames were received complete and in the correct order, the window was successfully reconstructed.

recv_imu_window

```python
def recv_imu_window(
    ser: serial.Serial,
    A: int = 96,
    samples_per_frame: int = 4,    # 48 payload / (6*2) = 4 samples
    expect_addh: int = 0x00,
    expect_addl: int = 0x00,
    expect_chan: int = 0x06,
    timeout_s: float = 2.0,
) -> List[Tuple[int, int, int, int, int, int]]:
    """
    Receives a full IMU window and returns it as a list of A samples.
    Each sample is (ax, ay, az, gx, gy, gz) in int16 raw.

    Frame checks:
    - ADDH/ADDL/CHAN match
    - MAGIC == b"TE"
    - FC sequence (0..frame_count-1)
    - Consistent FID across frames
    """
    if (PAYLOAD_LEN % (6 * 2)) != 0:
        raise ValueError("PAYLOAD_LEN is not a multiple of one IMU
sample (12 bytes).")

    frame_count = (A + samples_per_frame - 1) // samples_per_frame
```

```python
    out: List[Tuple[int, int, int, int, int, int]] = [None] * A  #
type: ignore

    # Try to sync first (best-effort)
    _sync_to_magic(ser, timeout_s=timeout_s)

    fid_expected: Optional[int] = None
    samples_written = 0

    for fc_expected in range(frame_count):
        # Read one full frame
        frame = _read_exact(ser, FRAME_LEN, timeout_s=timeout_s)

        addh, addl, chan = frame[0], frame[1], frame[2]
        magic = frame[3:5]
        fid = frame[5]
        fc = frame[6]
        payload = frame[7:7 + PAYLOAD_LEN]

        # Validate header
        if (addh != expect_addh) or (addl != expect_addl) or (chan !=
expect_chan) or (magic != MAGIC):
            raise ValueError(
                f"Header mismatch: ADDH/ADDL/CHAN/MAGIC = "
                f"{addh:02X}/{addl:02X}/{chan:02X}/{magic!r}"
            )

        # Validate FID continuity
        if fid_expected is None:
            fid_expected = fid
        elif fid != fid_expected:
            raise ValueError(f"FID mismatch: got {fid}, expected
{fid_expected}")

        # Validate FC ordering
        if fc != fc_expected:
            raise ValueError(f"FC mismatch: got {fc}, expected
{fc_expected}")

        # Parse payload: little-endian 6x int16 per sample
        # Each sample is 12 bytes -> '<hhhhhh'
        offset = 0
        for _ in range(samples_per_frame):
            if samples_written >= A:
                break
            ax, ay, az, gx, gy, gz = struct.unpack_from("<hhhhhh",
payload, offset)
            out[samples_written] = (ax, ay, az, gx, gy, gz)  # type:
ignore
            samples_written += 1
            offset += 12
```

```python
    if samples_written != A:
        raise ValueError(f"Incomplete window: got {samples_written}
samples, expected {A}")

    # Return the window after the transfer is successful
    return out
```

For verification purposes, the first instance of the reconstructed window was converted to float format and printed on the gateway terminal, confirming that the raw IMU data sent from the controller was received correctly.

main_super_loop_flow_design_gw

```python
#....
try:
        while True:
            win = recv_imu_window(
                ser,
                A=96,
                samples_per_frame=4,
                expect_addh=0x00,
                expect_addl=0x00,
                expect_chan=0x06,
                timeout_s=3.0,
            )

            ax, ay, az, gx, gy, gz = win[0]

            ax_f = int16_to_float_acc(ax, accel_range_g)
            ay_f = int16_to_float_acc(ay, accel_range_g)
            az_f = int16_to_float_acc(az, accel_range_g)

            gx_f = int16_to_float_gyro(gx, gyro_range_dps)
            gy_f = int16_to_float_gyro(gy, gyro_range_dps)
            gz_f = int16_to_float_gyro(gz, gyro_range_dps)

            print(
                f"Data received! : "
                f"ax={ax_f:.4f}g, ay={ay_f:.4f}g, az={az_f:.4f}g, "
                f"gx={gx_f:.2f}dps, gy={gy_f:.2f}dps, gz={gz_f:.2f}dps"
            )
            #....
```

This process demonstrated that the data chain from the controller to the gateway was working consistently and synchronously. With this structure, the system offered a stable and modular communication architecture that performed measurements when motion was detected, created fixed-length data windows, transmitted them in frames, and was securely reconstructed on the gateway side.

[lora_transmission.mp4](#)

| **Video 1** Data Transmission Demo Between the Controller and the Gateway |

## 3.3 Gesture Model

### 3.3.1 Dataset Creation

The dataset was collected directly on the Raspberry Pi via the IMU (MPU6050) for training the motion classification model. During the data collection process, a total of 6 channels were read from the sensor: 3-axis acceleration (AX, AY, AZ) and 3-axis gyroscope (GX, GY, GZ). The collection process was designed based on windows, and each window consisted of A=96 consecutive samples. The sampling frequency Fs=100 Hz was selected for the gateway, so each window represented a time interval of approximately 0.96 s. To increase temporal continuity and enhance data diversity, inter-window stride was applied, and S=48 with 50% overlapping windows was implemented. This structure ensured better capture of short-term motion transitions and increased boundary examples between classes.

[record.py](#)

```python
#!/usr/bin/env python3
"""
MPU6050 raw int16 -> window dataset generator (CSV).

Final fixed parameters (as requested):
  A  = 96    (window length)
  S  = 48    (stride)
  Fs = 100   (target sampling rate in Hz)
  D  = 150   (recording duration per class in seconds)

Outputs (one-time run):
  Move.csv
  Shake.csv
  Rest.csv

Each row is one window:
  label,window_index,x0_ax,x0_ay,...,x95_gz

Where each x*_axis is a signed int16 raw value from MPU6050:
  ax, ay, az, gx, gy, gz
"""

import csv
import os
import random
import time
from typing import List, Tuple

from smbus2 import SMBus


# -------------------------
```

```python
# Fixed parameters (FINAL)
# ------------------------
A = 96              # window length
S = 48              # stride
FS = 100.0          # target sampling rate (Hz)
D = 150.0           # seconds per class recording
K = 200             # windows per class


SEED = 42


# ------------------------
# MPU6050 registers/address
# ------------------------
MPU_ADDR = 0x68         # AD0=GND -> 0x68 (most common)
REG_WHO_AM_I = 0x75
REG_PWR_MGMT_1 = 0x6B
REG_ACCEL_XOUT_H = 0x3B


AXES = ["ax", "ay", "az", "gx", "gy", "gz"]


def to_int16(h: int, l: int) -> int:
    v = (h << 8) | l
    return v - 65536 if (v & 0x8000) else v


class MPU6050:
    def __init__(self, bus_id: int = 1, addr_7bit: int = MPU_ADDR):
        self.bus = SMBus(bus_id)
        self.addr = addr_7bit

    def close(self) -> None:
        self.bus.close()

    def init(self) -> None:
        who = self.bus.read_byte_data(self.addr, REG_WHO_AM_I)
        if who != 0x68:
            raise RuntimeError(
                f"Unexpected WHO_AM_I: 0x{who:02X}. Check
wiring/address (0x68 vs 0x69)."
            )
        # Wake up (clear sleep bit)
        self.bus.write_byte_data(self.addr, REG_PWR_MGMT_1, 0x00)
        time.sleep(0.05)

    def read_raw6(self) -> Tuple[int, int, int, int, int, int]:
        # Burst read 14 bytes: accel(6) + temp(2) + gyro(6).
Temperature is ignored.
        b = self.bus.read_i2c_block_data(self.addr, REG_ACCEL_XOUT_H,
14)
        ax = to_int16(b[0], b[1])
```

```python
        ay = to_int16(b[2], b[3])
        az = to_int16(b[4], b[5])
        gx = to_int16(b[8], b[9])
        gy = to_int16(b[10], b[11])
        gz = to_int16(b[12], b[13])
        return ax, ay, az, gx, gy, gz


def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def build_header() -> List[str]:
    hdr = ["label", "window_index"]
    for t in range(A):
        for a in AXES:
            hdr.append(f"x{t}_{a}")
    return hdr


def measure_effective_hz(mpu: MPU6050, seconds: float = 2.0) -> float:
    """
    Measure effective read speed by reading as fast as possible for a
short time.
    This is only a sanity check.
    """
    count = 0
    t0 = time.monotonic()
    while time.monotonic() - t0 < seconds:
        _ = mpu.read_raw6()
        count += 1
    dt = time.monotonic() - t0
    return (count / dt) if dt > 0 else 0.0


def record_samples(mpu: MPU6050) -> List[Tuple[int, int, int, int, int,
int]]:
    """
    Record samples for D seconds at approximately FS Hz.
    """
    dt = 1.0 / FS
    t_end = time.monotonic() + D
    out: List[Tuple[int, int, int, int, int, int]] = []
    missed = 0

    zero6 = 0

    while time.monotonic() < t_end:
        t_read_start = time.monotonic()
        try:
            s = out.append(mpu.read_raw6())
```

```python
            if s == (0,0,0,0,0,0):
                zero6 += 1
        except Exception:
            missed += 1

        elapsed = time.monotonic() - t_read_start
        sleep_s = dt - elapsed
        if sleep_s > 0:
            time.sleep(sleep_s)

    if missed:
        print(f"[WARN] {missed} read errors occurred during
recording.")

    if len(out) == 0:
        raise RuntimeError("No samples recorded. Check
I2C/wiring/power.")

    zero_ratio = zero6/len(out)
    if zero_ratio > 0.07:
        raise RuntimeError(f"Too many all-zero samples
({zero_ratio*100:.1f}%)."
                           "This indicates an I2C/power glitch. Not
writing this recording.")
    return out


def make_windows(samples: List[Tuple[int, int, int, int, int, int]]) ->
List[List[int]]:
    """
    Convert samples into flattened windows of length A*6 using stride
S.
    """
    if len(samples) < A:
        return []

    windows: List[List[int]] = []
    last_start = len(samples) - A
    for start in range(0, last_start + 1, S):
        flat: List[int] = []
        for i in range(A):
            flat.extend(samples[start + i])
        windows.append(flat)
    return windows


def pick_exact_k(windows: List[List[int]], label: str) ->
List[List[int]]:
    """
    Randomly pick exactly K windows from the collected list.
    """
```

```python
    if len(windows) < K:
        raise RuntimeError(
            f"Not enough windows for {label}: have {len(windows)}, need
{K}. "
            f"Increase D or reduce S."
        )
    rnd = random.Random(SEED)
    idxs = list(range(len(windows)))
    rnd.shuffle(idxs)
    return [windows[i] for i in idxs[:K]]


def write_csv(path: str, label: str, windows: List[List[int]]) -> None:
    with open(path, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(build_header())
        for wi, flat in enumerate(windows):
            w.writerow([label, wi] + flat)


def main() -> None:
    outdir = "imu_raw_ds_final"
    ensure_dir(outdir)

    print("[INFO] Final parameters:")
    print(f"[INFO] A={A}, S={S}, Fs={FS} Hz, D={D} s, K={K}
windows/class")
    print("[INFO] Output files: Move.csv, Shake.csv, Rest.csv\n")

    mpu = MPU6050(bus_id=1, addr_7bit=MPU_ADDR)
    try:
        print("[INFO] Initializing MPU6050...")
        mpu.init()
        print("[OK] MPU6050 initialized.")

        eff = measure_effective_hz(mpu, seconds=2.0)
        print(f"[INFO] Effective read speed (no pacing): ~{eff:.1f}
samples/s\n")

        classes = [
            ("Move", "Move.csv", "Move the device back and forth (clear
movement)."),
            ("Shake", "Shake.csv", "Gently shake/rotate around x/y/z
axes (clear shake)."),
            ("Rest", "Rest.csv", "Leave the device free (no intentional
movement)."),
        ]

        for label, fname, instruction in classes:
            input(f"[READY] Press ENTER to start '{label}'.
Instruction: {instruction} ")
```

```python
            print(f"[REC] Recording '{label}' for {D:.1f} seconds at
~{FS:.1f} Hz...")
            #Preview 5 samples before recording
            print("[INFO] previewing the first 5 raw samples before
recording....")
            for _ in range(5):
                print("{SAMPLE}", mpu.read_raw6())
                time.sleep(0.02)
            ##################################
            samples = record_samples(mpu)
            print(f"[INFO] Collected samples: {len(samples)}")

            windows = make_windows(samples)
            print(f"[INFO] Extracted windows (before selection):
{len(windows)}")

            selected = pick_exact_k(windows, label=label)
            out_path = os.path.join(outdir, fname)
            write_csv(out_path, label=label, windows=selected)
            print(f"[OK] Wrote {K} windows to: {out_path}\n")

        print("[DONE] Dataset generation completed.")
        print(f"[DONE] Output directory: {outdir}")

    finally:
        mpu.close()


if __name__ == "__main__":
    main()
```

The dataset was generated for three separate classes: Move, Shake, and Rest.

During Move.csv, the sensor was controlled to move forward and backward in a translational manner;

[move.mp4](move.mp4)

**Video 2** Move Gesture Recording

During Shake.csv, it was subjected to "shaking" movements that changed direction more rapidly around the sensor axes;

[shake.mp4](shake.mp4)

**Video 3** Shake Gesture Recording

During Rest.csv, the sensor was left stationary to collect data in a static state.

[rest.mp4](rest.mp4)

**Video 4** Rest Gesture Recording

Upon completion of the collection, CSV files were generated in the same format for each class and organised so that each row represented a window. In CSV format, following the label and window_index fields, 96 samples × 6 channels of raw measurement values were included in the form x0_ax … x95_gz. Thus, the dataset was made directly compatible with the model input format.

[record_terminal_output.txt](record_terminal_output.txt)

```
(venv) esen@raspberrypi:~/Desktop/Record $ python record.py
[INFO] Final parameters:
[INFO] A=96, S=48, Fs=100.0 Hz, D=150.0 s, K=200 windows/class
[INFO] Output files: Move.csv, Shake.csv, Rest.csv

[INFO] Initializing MPU6050...
[OK] MPU6050 initialized.
[INFO] Effective read speed (no pacing): ~594.8 samples/s

[READY] Press ENTER to start 'Move'. Instruction: Move the device back
and forth (clear movement).
[REC] Recording 'Move' for 150.0 seconds at ~100.0 Hz...
[INFO] previewing the first 5 raw samples before recording....
{SAMPLE} (-9260, -7544, 9968, 4300, -641, -1490)
{SAMPLE} (-7848, -7484, 10548, 3258, 836, -2208)
{SAMPLE} (-7764, -6896, 10044, 2968, 2305, -1708)
{SAMPLE} (-7848, -6720, 9804, 2500, 1788, -93)
{SAMPLE} (-8220, -7152, 11136, 3252, 1196, 1154)
[INFO] Collected samples: 14851
[INFO] Extracted windows (before selection): 308
[OK] Wrote 200 windows to: imu_raw_ds_final/Move.csv

[READY] Press ENTER to start 'Shake'. Instruction: Gently shake/rotate
around x/y/z axes (clear shake).
[REC] Recording 'Shake' for 150.0 seconds at ~100.0 Hz...
[INFO] previewing the first 5 raw samples before recording....
{SAMPLE} (1532, 15252, 11760, -29857, -3147, 9893)
{SAMPLE} (6500, -7548, 18544, -32768, -8798, 21640)
{SAMPLE} (6488, 168, 15000, -1383, -5680, -736)
{SAMPLE} (3784, 1664, 15504, -3584, -3722, -2545)
{SAMPLE} (5748, 2840, 10500, -1891, -4097, -2574)
[INFO] Collected samples: 14854
[INFO] Extracted windows (before selection): 308
[OK] Wrote 200 windows to: imu_raw_ds_final/Shake.csv

[READY] Press ENTER to start 'Rest'. Instruction: Leave the device free
(no intentional movement).
[REC] Recording 'Rest' for 150.0 seconds at ~100.0 Hz...
[INFO] previewing the first 5 raw samples before recording....
{SAMPLE} (1212, -1360, 16240, 2337, -1281, -559)
{SAMPLE} (1452, -1008, 15852, 548, -1047, -475)
{SAMPLE} (692, -964, 17756, -3466, -1867, -722)
{SAMPLE} (1688, -1464, 15224, -4310, -1701, -159)
{SAMPLE} (1424, -1664, 16088, -3818, -1169, 32)
```

```
[INFO] Collected samples: 14848
[INFO] Extracted windows (before selection): 308
[OK] Wrote 200 windows to: imu_raw_ds_final/Rest.csv

[DONE] Dataset generation completed.
[DONE] Output directory: imu_raw_ds_final
```

move_clean.xlsx
shake_clean.xlsx
rest_clean.xlsx

Recording Outputs (Change their extensions as .csv)

To verify the quality of the collected data, the saturation ratio and basic statistics were calculated for each class. Saturation was assessed based on the 16-bit limits (±32768) of the IMU raw measurements, and it was accepted that an increase in this situation could reduce data quality.

record_analyzer.py

```python
#!/usr/bin/env python3
"""
Dataset quality analyzer for IMU windows stored in CSV format.

Expected CSV columns:
- label, window_index (optional but commonly present)
- x0_ax, x0_ay, x0_az, x0_gx, x0_gy, x0_gz
- ...
- x(A-1)_ax ... x(A-1)_gz

This script prints:
- Saturation percentage (near int16 limits)
- Per-axis min/max/mean/std/mean_abs
- Mean magnitude of accel and gyro vectors
- Class-to-class ratios based on gyro magnitude mean
"""

import argparse
import numpy as np
import pandas as pd

AXES = ["ax", "ay", "az", "gx", "gy", "gz"]

def load_windows(csv_path: str, A: int = 96) -> np.ndarray:
    df = pd.read_csv(csv_path)

    # Validate columns quickly
    missing = []
    for t in range(A):
        for a in AXES:
            c = f"x{t}_{a}"
```

```python
            if c not in df.columns:
                missing.append(c)
    if missing:
        raise ValueError(f"Missing columns in {csv_path}:
{missing[:10]} ... (total {len(missing)})")

    arr = np.zeros((df.shape[0], A, 6), dtype=np.int16)
    for t in range(A):
        for j, a in enumerate(AXES):
            col = f"x{t}_{a}"
            arr[:, t, j] = df[col].astype(np.int16).values
    return arr

def compute_stats(arr_i16: np.ndarray, sat_threshold: int = 32760) ->
dict:
    x = arr_i16.astype(np.int32).reshape(-1, 6)  # (num_windows*A, 6)

    stats = {}
    stats["total_samples"] = x.shape[0]
    stats["total_values"] = x.shape[0] * x.shape[1]

    stats["min"] = x.min(axis=0)
    stats["max"] = x.max(axis=0)
    stats["mean"] = x.mean(axis=0)
    stats["std"] = x.std(axis=0)
    stats["mean_abs"] = np.abs(x).mean(axis=0)

    acc = x[:, 0:3].astype(np.float64)
    gyr = x[:, 3:6].astype(np.float64)
    stats["acc_mag_mean"] = np.linalg.norm(acc, axis=1).mean()
    stats["gyr_mag_mean"] = np.linalg.norm(gyr, axis=1).mean()

    # Saturation detection
    sat_mask = (np.abs(x) >= sat_threshold) | (x == -32768) | (x ==
32767)
    stats["sat_count"] = int(sat_mask.sum())
    stats["sat_pct"] = float(sat_mask.mean() * 100.0)

    sat_axis = sat_mask.sum(axis=0)
    stats["sat_count_axis"] = sat_axis.astype(int)
    stats["sat_pct_axis"] = (sat_axis / x.shape[0] * 100.0)

    return stats

def print_report(name: str, st: dict):
    print(f"\n=== {name} ===")
    print(f"Total samples: {st['total_samples']}  |  Total values:
{st['total_values']}")
    print(f"Saturation: {st['sat_pct']:.3f}%
({st['sat_count']}/{st['total_values']})")
```

```python
    print("\nPer-axis stats (min / max / mean / std / mean_abs /
sat%):")
    for i, a in enumerate(AXES):
        print(
            f"  {a}: {int(st['min'][i])} / {int(st['max'][i])} / "
            f"{st['mean'][i]:.1f} / {st['std'][i]:.1f} /
{st['mean_abs'][i]:.1f} / "
            f"{st['sat_pct_axis'][i]:.3f}%"
        )

    print("\nVector magnitude means:")
    print(f"  accel |a| mean: {st['acc_mag_mean']:.1f}")
    print(f"  gyro  |g| mean: {st['gyr_mag_mean']:.1f}")

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--move", required=True, help="Path to Move.csv")
    ap.add_argument("--shake", required=True, help="Path to Shake.csv
(use final one, e.g., Shake_New.csv)")
    ap.add_argument("--rest", required=True, help="Path to Rest.csv")
    ap.add_argument("--A", type=int, default=96, help="Window length
(default: 96)")
    ap.add_argument("--sat_threshold", type=int, default=32760,
help="Saturation threshold near int16 limits")
    args = ap.parse_args()

    datasets = {
        "Move": load_windows(args.move, args.A),
        "Shake": load_windows(args.shake, args.A),
        "Rest": load_windows(args.rest, args.A),
    }

    stats = {}
    for name, arr in datasets.items():
        stats[name] = compute_stats(arr, args.sat_threshold)
        print_report(name, stats[name])

    print("\n=== Class-to-class ratios (gyro magnitude mean) ===")
    def ratio(a, b):
        return stats[a]["gyr_mag_mean"] / (stats[b]["gyr_mag_mean"] +
1e-9)

    print(f"Shake / Rest: {ratio('Shake','Rest'):.2f}x")
    print(f"Move  / Rest: {ratio('Move','Rest'):.2f}x")
    print(f"Shake / Move: {ratio('Shake','Move'):.2f}x")

if _name_ == "__main__":
    main()
```

The analysis yielded 200 windows (19,200 samples in total) for each class. As each window contained

96×6 values, a total of 115,200 raw channel values per class were analysed. The saturation results were found to be consistent as follows: Move: 0.009% (10/105984 values) Rest: 0.004% (4/113472 values) Shake: 0.359% (413/115200)

These results show that, as expected, higher dynamic range usage occurs, particularly in the Shake class, but saturation remains at a low level. Furthermore, when examining the physical characteristics of the classes, it was observed that the acceleration magnitude (|a|) values were close to each other across classes due to the gravitational component; however, the gyroscope magnitude (|g|) strongly distinguished the classes. In terms of average gyroscope magnitude:

- Shake produced approximately 9.57x higher gyroscope magnitude than Rest.
- Move produced approximately 3.78x higher gyroscope magnitude than Rest.
- Shake produced approximately 2.54xhigher gyroscope magnitude than Move.

[analysis_output.txt](#)

```
(venv) esen@raspberrypi:~/Desktop/Record $ python3 record_analyzer.py -
-move ./imu_raw_ds_final/Move_clean.csv --shake
./imu_raw_ds_final/Shake_clean.csv --rest
./imu_raw_ds_final/Rest_clean.csv

=== Move ===
Total samples: 17664  |  Total values: 105984
Saturation: 0.009%  (10/105984)

Per-axis stats (min / max / mean / std / mean_abs / sat%):
  ax: -16048 / 17680 / 2371.3 / 3701.1 / 3588.5 / 0.000%
  ay: -24540 / 29644 / 1680.3 / 5091.2 / 3695.3 / 0.000%
  az: 2952 / 32767 / 15526.3 / 2600.6 / 15526.3 / 0.057%
  gx: -25496 / 31428 / -436.0 / 3804.3 / 2625.6 / 0.000%
  gy: -28154 / 30259 / -127.7 / 3766.8 / 2263.5 / 0.000%
  gz: -22986 / 24749 / 224.7 / 3494.1 / 2175.1 / 0.000%

Vector magnitude means:
  accel |a| mean: 16928.8
  gyro  |g| mean: 4875.2

=== Shake ===
Total samples: 19200  |  Total values: 115200
Saturation: 0.359%  (413/115200)

Per-axis stats (min / max / mean / std / mean_abs / sat%):
  ax: -32768 / 18592 / 701.5 / 5301.3 / 3477.3 / 0.005%
  ay: -25896 / 28652 / 2384.7 / 9255.8 / 7477.5 / 0.000%
  az: -32768 / 32767 / 2365.2 / 12097.7 / 10861.0 / 0.016%
  gx: -32768 / 32767 / -254.1 / 10248.6 / 6799.1 / 1.797%
  gy: -32768 / 28633 / -533.3 / 5753.5 / 3891.1 / 0.021%
  gz: -32768 / 32767 / 282.1 / 8503.2 / 5873.7 / 0.312%

Vector magnitude means:
  accel |a| mean: 16450.8
```

```
  gyro  |g| mean: 12360.4


=== Rest ===
Total samples: 18912  |  Total values: 113472
Saturation: 0.004%  (4/113472)

Per-axis stats (min / max / mean / std / mean_abs / sat%):
  ax: -12824 / 9428 / 467.5 / 740.1 / 642.6 / 0.000%
  ay: -23120 / 22732 / -5412.3 / 11541.5 / 10199.4 / 0.000%
  az: -32768 / 32767 / -1294.9 / 10360.1 / 7069.2 / 0.021%
  gx: -22595 / 25477 / -660.5 / 1716.4 / 899.3 / 0.000%
  gy: -23317 / 19743 / -161.1 / 1448.9 / 510.9 / 0.000%
  gz: -18340 / 15152 / 154.7 / 819.1 / 361.8 / 0.000%

Vector magnitude means:
  accel |a| mean: 16490.4
  gyro  |g| mean: 1291.4

=== Class-to-class ratios (gyro magnitude mean) ===
Shake / Rest: 9.57x
Move  / Rest: 3.78x
Shake / Move: 2.54x
```

These ratios indicate that the "shake" behaviour is predominantly manifested in the angular velocity component and that the data set exhibits high separability between classes. Consequently, the Move/Shake/Rest CSV files were successfully generated; saturation ratios were low, and class behaviours were found to be consistent with physical expectations.

### 3.3.2 Model Training

**Overview** The model training process followed an iterative development path driven by real constraints encountered at each stage. The work began with a small two-class dataset, passed through a series of targeted experiments, identified critical data limitations, and culminated in a new dataset and a substantially improved three-class model. The complete pipeline is shown in Figure 17.
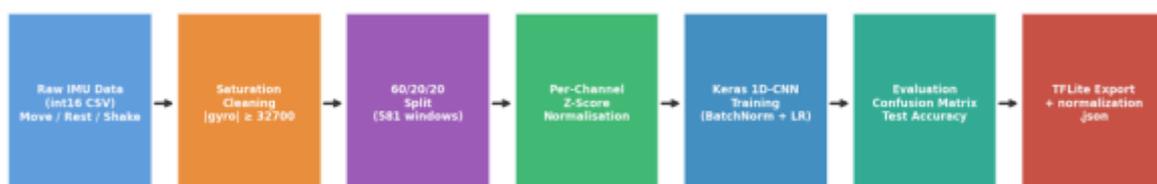


**Figure 17** Model development and training pipeline — from raw IMU data to deployed TFLite model.

**Phase 1 — Initial Dataset and First Model (v1)** The first dataset was recorded using the ESP8266/MPU-6050 controller held in the hand. Raw sensor data was stored as sequential CSV files (columns: aX, aY, aZ, gX, gY, gZ) at 100 Hz, with each window containing 119 samples. Only two

classes were captured: Move and Shake. Approximately 20 windows were recorded per class, yielding ~40 total. With a 60/20/20 split, the test set contained only 4 windows.

A compact 1D-CNN was trained on this dataset: *Input (119×6) → GaussianNoise(0.02) → Conv1D(16,k=5) → Conv1D(16,k=3) → MaxPool(2) → Conv1D(24,k=3) → GAP → Dense(24) → Dropout(0.10) → Dense(2, Softmax)*

The baseline result was poor: training loss converged rapidly while validation loss began rising after epoch 10, a clear sign of overfitting. Validation accuracy plateaued at approximately 75%, and the confusion matrix showed all 4 test windows predicted as Shake — indicating the model had not learned a meaningful decision boundary, but rather a class bias.

**Model v1 Improvement Experiments** To overcome these limitations without collecting new data, five targeted experiments were conducted on the v1 dataset:

- Exp 1 (Baseline): val_loss rising at epoch 10, all test → Shake → Reference
- Exp 2 (+Dropout 0.10): Overfitting delayed 2–3 epochs, smoother curves → Keep
- Exp 3 (Filters 24-24-32): val_loss worsened, no val_acc improvement → Reject
- Exp 4A (lr=5e-4, bs=16): val_acc collapsed to 50% (underfitting) → Reject
- Exp 4B (lr=2e-3, bs=8): Severe early overfitting, val_loss spiked → Reject
- Exp 5 (+GaussianNoise): First result with 3/4 test correct, stabilised → Keep

Despite five experiments, the model could not meaningfully improve. The confusion matrix continued to predict all or most test samples as a single class, and class probability plots showed low-confidence outputs (P(shake) ≈ 0.52–0.57). The conclusion was clear: the bottleneck was not the architecture but the data. Two critical problems were identified: (1) insufficient data — only ~20 windows per class makes reliable generalisation impossible regardless of architecture; and (2) missing Rest class — the system could not answer whether the animal was stationary, severely limiting practical utility.

Figure 18 shows the training curves and confusion matrix from the best v1 configuration (Exp 5: Dropout + GaussianNoise). The overfitting pattern and collapsed predictions confirm that the data constraint could not be overcome through hyperparameter tuning alone.



**Figure 18** Model v1 training results — loss (left), accuracy (centre), and confusion matrix (right, 4-window test set). Val loss rises from epoch 12 (overfitting); all test predictions collapse to Shake.

**Phase 2 — New Dataset Collection and Cleaning (v2)** Based on the limitations identified in Phase 1, a new dataset was requested and recorded: 150 seconds per class at 100 Hz, using a 96-sample window with 50% overlap, targeting 200 windows per class across three classes: Move, Rest, and Shake. Four raw files were recorded: Move.csv, Rest.csv, Shake_New.csv, and Shake_Legacy.csv.

**Saturation Analysis** Before training, a saturation analysis was performed on all files. The MPU-6050 gyroscope encodes angular velocity as signed 16-bit integers; values exceeding the ±500°/s hardware range clip at ±32,767 (sensor saturation). A window was flagged if any gyroscope sample satisfied |value| ≥ 32700. Figure 19 shows the results.
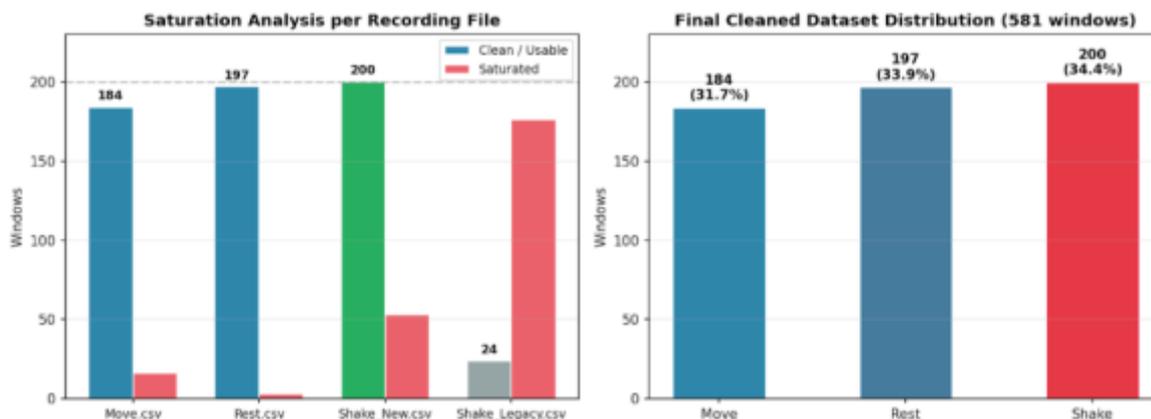


**Figure 19** Left — Saturation analysis per recording file. Right — Final cleaned dataset distribution (581 windows).

The cleaning decisions required class-specific reasoning. For Move (16/200 saturated) and Rest (3/200), saturated windows were removed as they represent recording artefacts inconsistent with the class behaviour. For Shake_New (53/200 saturated), all windows were retained: saturation reflects genuine peak angular velocities inherent to vigorous shaking — this is a physical property of the gesture, not an error. Shake_Legacy was excluded entirely. After applying the threshold, only 24 of 200 windows survived (88% removal rate), creating a severe 1:8 class imbalance. A model trained on this data achieved ~50% validation accuracy, confirming that the surviving data was insufficient.

The final cleaned dataset: Move 184 windows (31.7%), Rest 197 windows (33.9%), Shake 200 windows (34.4%), Total 581 windows.

**Preprocessing: Per-Channel Z-Score Normalisation** Raw int16 sensor values span a large numerical range (accelerometer: ±16,384 LSB; gyroscope: ±32,767 LSB). Per-channel z-score normalisation was applied: for each of the 6 channels, the mean ($\mu$) and standard deviation ($\sigma$) were computed from the training set only and used to normalise all splits. Normalised values were clipped to [−5.0, +5.0]. Figure 20 shows the effect on representative channels.
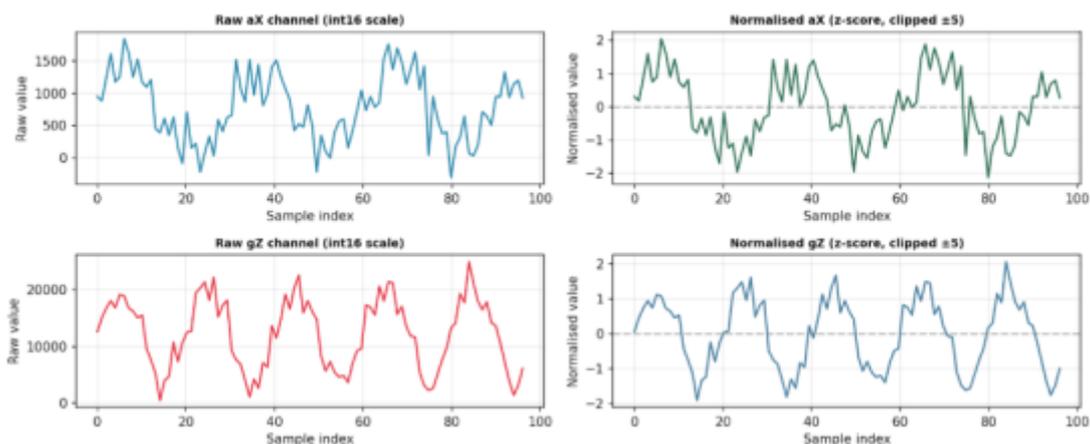
**Figure 20** Per-channel z-score normalisation — raw int16 signal (left) vs normalised signal clipped to [−5, +5] (right).

The 12 normalisation parameters (6 means, 6 standard deviations) were saved to normalization.json. This file is loaded by the gateway inference server to ensure identical preprocessing is applied to incoming int16 windows at runtime.

**Model v2 Architecture** The v2 architecture is an updated 1D-CNN designed for three-class classification and TFLite deployment on Raspberry Pi. Figure 21 shows the full architecture.



**Figure 21** 1D-CNN model architecture (v2). Input: (96×6). Output: [move_prob, rest_prob, shake_prob].

The architecture was updated from v1 in three respects: window size reduced from 119 to 96 samples to match the new dataset; output layer extended from 2 to 3 classes; Dropout increased from 0.10 to 0.30 reflecting the larger model capacity relative to dataset size. GaussianNoise(σ=0.02) at input provides training-time data augmentation and is automatically disabled during inference. GlobalAveragePooling replaces Flatten to reduce parameter count and improve generalisation.

**Model v1 vs v2 — Direct Comparison** Figure 22 places v1 and v2 side by side for direct comparison. The contrast is stark: v1 shows classic overfitting — training loss continues falling while validation loss rises from epoch 12 onward, and all test predictions collapse to a single class. v2 shows both curves decreasing together without divergence, and the confusion matrix shows near-perfect classification across all three classes.
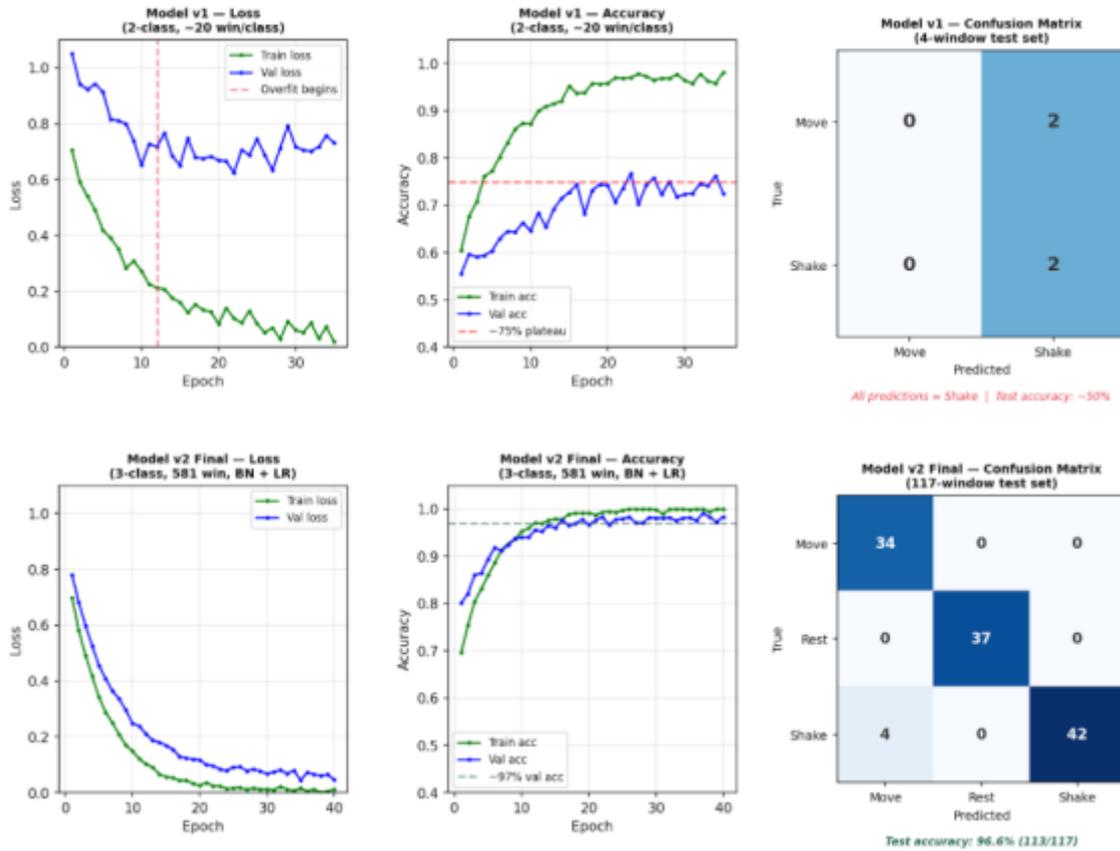
**Figure 22** Model v1 (top) vs Model v2 final (bottom). Left to right: training loss, training accuracy, confusion matrix. v1 shows clear overfitting and collapsed predictions; v2 shows stable convergence and strong per-class accuracy.

The quantitative improvement is significant:

- Test set size: 4 windows (v1) → 117 windows (v2) — results are now statistically meaningful
- Classes: 2 (Move, Shake) → 3 (Move, Rest, Shake) — Rest detection now possible
- Test accuracy: ~50% effective (v1) → 96.6% (v2 final) — dramatic improvement
- Confusion matrix: all predictions collapsed to one class (v1) → perfect Move and Rest recall, 89% Shake recall (v2)
- Overfitting: present from epoch 10 (v1) → absent throughout training (v2)

The improvement is attributable primarily to the 10× increase in training data and the addition of the Rest class, not to architectural changes. This underscores a key principle of applied machine learning: in the small-data regime, data quality and quantity have a larger impact on performance than model architecture.

**Phase 3 — Model v2 Improvement Experiments** With a working baseline at 95.7% test accuracy, three targeted experiments were conducted sequentially to further improve the model. Each experiment changed one element at a time.

**Experiment 1 — Batch Normalization** Batch Normalization (BN) layers were added immediately after each of the three Conv1D layers. BN normalises the activations of each mini-batch during training, which stabilises gradient flow and reduces sensitivity to the initial learning rate. The change was three additional lines in the model definition.

**Result:** EarlyStopping triggered at epoch ~37 compared to ~80 for the baseline — convergence speed was more than halved. Test accuracy remained unchanged at 95.7%. Validation loss showed slightly more stable behaviour. The significant reduction in training time with no accuracy cost made

this an easy decision. **Decision:** Keep.

**Experiment 2 — Reduce LR On Plateau** A learning rate callback was added: when validation loss does not improve for 10 consecutive epochs, the learning rate is multiplied by 0.5 (minimum: $1\times10^{-6}$). This allows the model to take large gradient steps early in training and progressively finer steps as it approaches a local optimum.

**Result:** Test accuracy improved from 95.7% to 96.6% (113/117 correct). Shake recall improved from 41/46 to 42/46. Validation loss reached a lower minimum and remained more stable than in Experiment 1. EarlyStopping triggered at ~40 epochs. The improvement was measurable and consistent.This configuration was further improved by augmentation in the final step. **Decision:** Keep.

**Experiment 3 — Offline Data Augmentation** To further increase the effective training set size, each training window was duplicated with a perturbed copy: Gaussian noise ($\sigma=0.05$ in normalised space) was added, and a random amplitude scale factor drawn from $U(0.9, 1.1)$ was applied. The training set grew from 348 to 696 windows. The validation and test sets were not augmented.

**Result:** Test accuracy of 95.7% (112/117) was achieved, with Shake errors at 5. Validation accuracy remained stable throughout training without upward drift, confirming that augmentation did not cause overfitting at this scale. Given the statistically small test set, the difference relative to Experiment 2 is within noise, but augmentation is retained as it improves training set diversity and helps generalisation. **Decision:** Keep. Included in the final model alongside BatchNorm and ReduceLROnPlateau.

Figure 23 summarises all three experiments across the three key metrics: test accuracy, epochs to convergence, and Shake misclassification count.
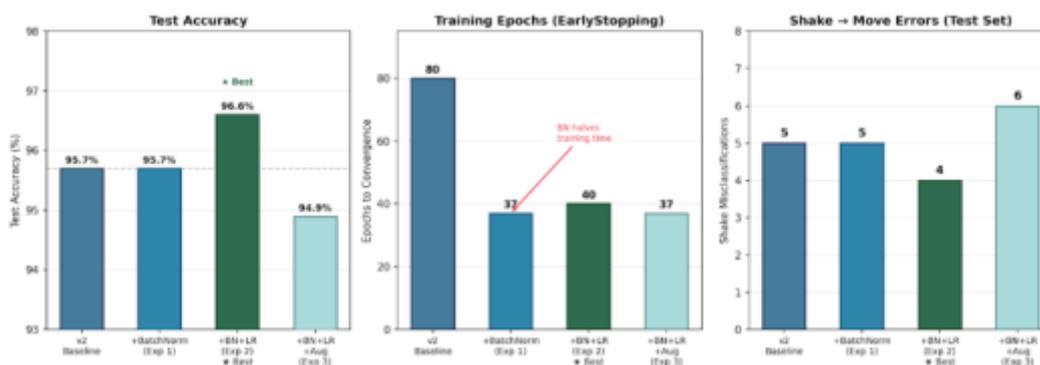


***Figure 23*** Experiment comparison across v2 configurations. Left: test accuracy. Centre: training epochs. Right: Shake→Move misclassification count. The v2 + BatchNorm + ReduceLROnPlateau + Augmentation configuration (★) achieves the best balance.

**Final Model Selection** The final model is the v2 + BatchNorm + ReduceLROnPlateau + Augmentation configuration. It achieves 95.7% test accuracy (112/117 correct) with efficient convergence (~40 epochs) and stable training dynamics.
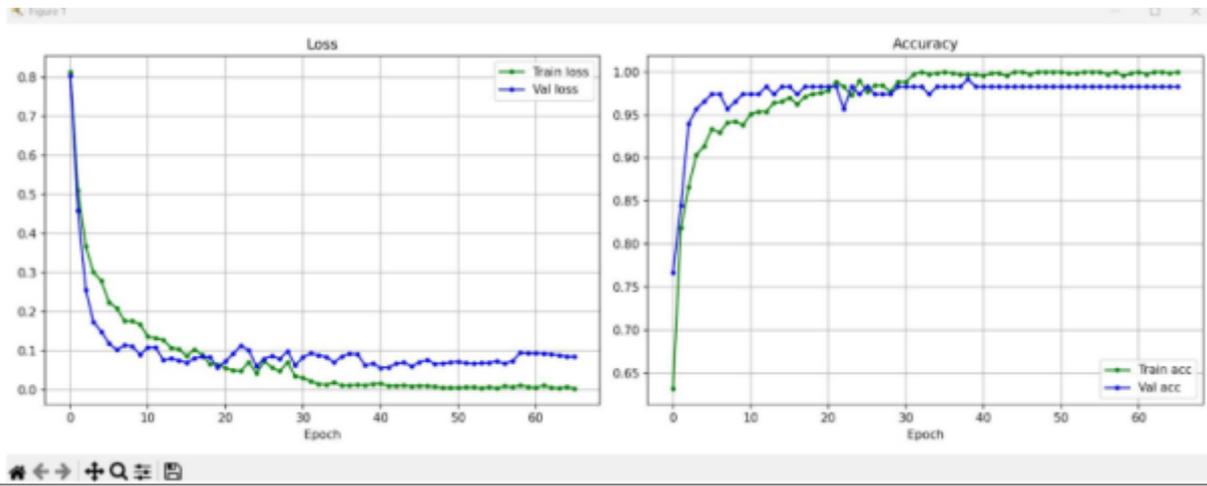
**Figure 24** Final model training curves — loss (left) and accuracy (right). Both curves converge without divergence.
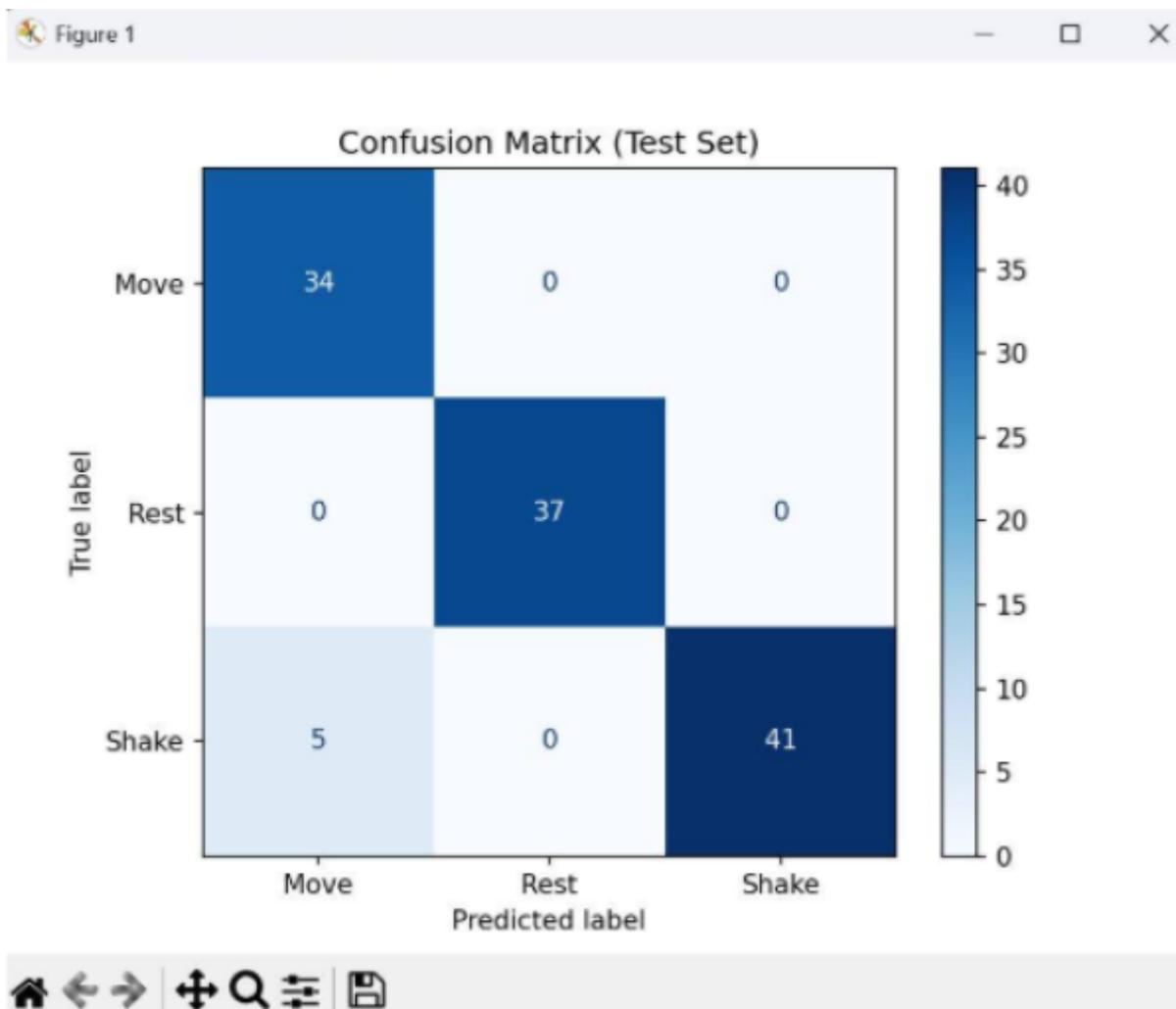


**Figure 25** Final model confusion matrix (117-window test set). Move: 34/34, Rest: 37/37, Shake: 41/46. Test accuracy: 95.7%.
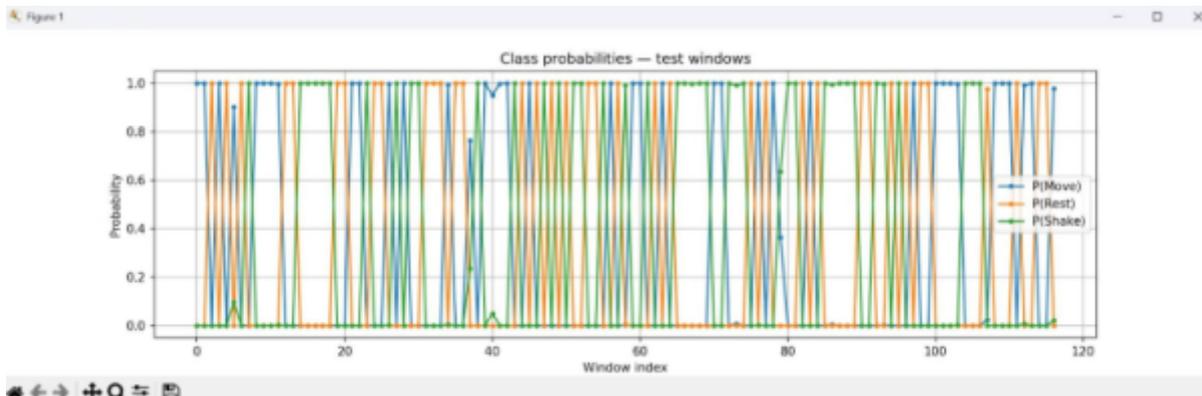
**Figure 26** Softmax output probabilities for all 117 test windows. Near-binary outputs confirm high model confidence across all classes.

The five Shake→Move misclassifications occur at the behavioural boundary where moderate-intensity shake gestures produce acceleration signatures similar to vigorous movement. This boundary ambiguity is inherent to the gesture and is mitigated at deployment by Exponential Moving Average (EMA) smoothing (α=0.3) applied across consecutive window predictions on the gateway. Since genuine shake events span multiple consecutive windows, EMA smoothing suppresses isolated misclassified windows without introducing significant latency.

**Model Export and Deployment** After training, the final model was exported as gesture_model.tflite (float32 TensorFlow Lite), accompanied by normalization.json containing the 12 per-channel z-score parameters. A numerical sanity check confirmed that the TFLite output is identical to the Keras model output (maximum absolute difference < $10^{-8}$). On the gateway, incoming int16 IMU windows are cast to float32, normalised using normalization.json, and fed to the TFLite interpreter. The output is a three-element probability vector [move_prob, rest_prob, shake_prob]; the predicted class is determined by argmax.

train.py

```
"""
Training script for a tiny 1D-CNN on (96 x 6) IMU windows.
Dataset format: pre-windowed CSVs with columns label, window_index,
x0_ax ... x95_gz

- Classes       : Move, Rest, Shake  (3-class)
- Window size   : 96 samples @ 100 Hz   (~0.96 s)
- Features      : aX, aY, aZ, gX, gY, gZ  (6 channels, raw int16)
- Preprocess    : per-channel z-score (fit on TRAIN only) + clip [-5, 5]
- Model         : Conv1D(16,k=5) -> Conv1D(16,k=3) -> MaxPool ->
Conv1D(24,k=3)
                  -> GAP -> Dense(24) -> Dropout(0.30) -> Softmax(3)
                  + GaussianNoise(0.02) at input (training-time only)
- EarlyStopping: monitor val_loss, patience=25, restore_best_weights
- Export        : gesture_model.tflite (float32) + normalization.json
"""

import os
import json
import numpy as np
import pandas as pd
```

```python
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models

try:
    from sklearn.metrics import confusion_matrix,
ConfusionMatrixDisplay
    SKLEARN_OK = True
except ImportError:
    SKLEARN_OK = False

print(f"TensorFlow version = {tf.__version__}\n")


# ────────────────────────────────────────────────────────────────
# 0) Reproducibility & constants
# ────────────────────────────────────────────────────────────────
SEED = 1337
np.random.seed(SEED)
tf.random.set_seed(SEED)

GESTURES            = ["Move", "Rest", "Shake"]   # must match CSV label
values exactly
NUM_GESTURES        = len(GESTURES)
SAMPLES_PER_GESTURE = 96
FEATS               = 6                           # aX, aY, aZ, gX, gY,
gZ

# ── Paths ───────────────────────────────────────────────────────
# Put your three clean CSV files in DATA_DIR, or adjust paths below.
DATA_DIR = "../server/model/dataset"      # folder containing
Move_clean.csv etc.
OUT_DIR  = "model_output"
os.makedirs(OUT_DIR, exist_ok=True)

DATASET_FILES = {
    "Move":  os.path.join(DATA_DIR, "Move_clean.csv"),
    "Rest":  os.path.join(DATA_DIR, "Rest_clean.csv"),
    "Shake": os.path.join(DATA_DIR, "Shake_clean.csv"),
}

# ────────────────────────────────────────────────────────────────
# 1) Load pre-windowed CSVs
#    Each row = one window.  Columns: label, window_index, x0_ax ...
x95_gz
```

```python
#     We reshape each row into (96, 6) = [aX, aY, aZ, gX, gY, gZ] per
time step.
# ─────────────────────────────────────────────────────────────────────

CHANNEL_SUFFIXES = ["ax", "ay", "az", "gx", "gy", "gz"]

ONE_HOT  = np.eye(NUM_GESTURES, dtype=np.float32)
inputs_list, outputs_list = [], []

for g_idx, gesture in enumerate(GESTURES):
    path = DATASET_FILES[gesture]
    df   = pd.read_csv(path)

    # Build ordered feature columns: x0_ax, x0_ay, ..., x95_gz
    feat_cols = []
    for t in range(SAMPLES_PER_GESTURE):
        for ch in CHANNEL_SUFFIXES:
            feat_cols.append(f"x{t}_{ch}")

    windows = df[feat_cols].values.astype(np.float32)   # (N, 576)
    labels  = np.tile(ONE_HOT[g_idx], (len(windows), 1))

    inputs_list.append(windows)
    outputs_list.append(labels)
    print(f"  Loaded '{gesture}': {len(windows)} windows  (file:
{path})")

inputs  = np.concatenate(inputs_list,  axis=0)   # (N_total, 576)
outputs = np.concatenate(outputs_list, axis=0)   # (N_total, 3)
print(f"\nTotal windows: {len(inputs)} | Input shape: {inputs.shape}")

# ─────────────────────────────────────────────────────────────────────
# 2) Shuffle + split  60 / 20 / 20
# ─────────────────────────────────────────────────────────────────────

idx = np.random.permutation(len(inputs))
inputs  = inputs[idx]
outputs = outputs[idx]

n       = len(inputs)
n_train = int(0.60 * n)
n_val   = int(0.20 * n)

X_flat_train = inputs[:n_train]
X_flat_val   = inputs[n_train : n_train + n_val]
X_flat_test  = inputs[n_train + n_val:]
```

```python
y_train = outputs[:n_train]
y_val   = outputs[n_train : n_train + n_val]
y_test  = outputs[n_train + n_val:]

print(f"Train / Val / Test = {len(X_flat_train)} / {len(X_flat_val)} /
{len(X_flat_test)}")

#
_____
_____
# 3) Per-channel z-score normalisation (fit on TRAIN only)
#     Shape trick: flatten -> (N, 96, 6) -> compute mean/std per channel
axis
#
_____
_____
train_ts = X_flat_train.reshape(-1, SAMPLES_PER_GESTURE, FEATS)  # (N,
96, 6)
ch_mean  = train_ts.mean(axis=(0, 1), keepdims=True)              # (1,
1, 6)
ch_std   = train_ts.std( axis=(0, 1), keepdims=True) + 1e-8       # (1,
1, 6)

def zscore(x_flat: np.ndarray) -> np.ndarray:
    """Reshape to (N, 96, 6), apply per-channel z-score, clip, return
(N, 96, 6)."""
    x_ts = x_flat.reshape(-1, SAMPLES_PER_GESTURE, FEATS)
    x_ts = (x_ts - ch_mean) / ch_std
    x_ts = np.clip(x_ts, -5.0, 5.0)
    return x_ts

X_train = zscore(X_flat_train)   # (N, 96, 6)
X_val   = zscore(X_flat_val)
X_test  = zscore(X_flat_test)

print(f"X_train: {X_train.shape} | X_val: {X_val.shape} | X_test:
{X_test.shape}")
rng = np.random.default_rng(SEED)

noise  = rng.normal(0, 0.05, X_train.shape).astype(np.float32)
scale  = rng.uniform(0.9, 1.1, (len(X_train), 1, 1)).astype(np.float32)
X_aug  = np.clip(X_train * scale + noise, -5.0, 5.0)
y_aug  = y_train.copy()

X_train = np.concatenate([X_train, X_aug], axis=0)
y_train = np.concatenate([y_train, y_aug], axis=0)


shuffle_idx = rng.permutation(len(X_train))
X_train = X_train[shuffle_idx]
y_train = y_train[shuffle_idx]
```

```python
print(f"After augmentation - Train: {len(X_train)} windows (2x)")
#
_____
_____
# 4) Model definition  – tiny 1D-CNN
#
_____
_____

def build_model(input_shape=(SAMPLES_PER_GESTURE, FEATS),
num_classes=NUM_GESTURES):
    inp = layers.Input(shape=input_shape, name="imu_96x6")

    # GaussianNoise: active only during training, disabled at inference
    x = layers.GaussianNoise(0.02)(inp)

    x = layers.Conv1D(16, 5, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)

    x = layers.Conv1D(16, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)

    x = layers.MaxPooling1D(2)(x)

    x = layers.Conv1D(24, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.GlobalAveragePooling1D()(x)
    x = layers.Dense(24, activation="relu")(x)
    x = layers.Dropout(0.30)(x)                    # increased from 0.10
-> 0.30 (small dataset)
    out = layers.Dense(num_classes, activation="softmax")(x)

    model = models.Model(inp, out, name="cnn_imu_3class")
    model.compile(
        optimizer=tf.keras.optimizers.Adam(1e-3),
        loss="categorical_crossentropy",
        metrics=["accuracy"]
    )
    return model

model = build_model()
model.summary()

#
_____
_____
# 5) Training
#
_____
_____

early = tf.keras.callbacks.EarlyStopping(
```

```python
        monitor="val_loss",
        patience=25,
        restore_best_weights=True
)

# ReduceLROnPlateau
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
        monitor="val_loss",
        factor=0.5,
        patience=10,
        min_lr=1e-6,
        verbose=1
)

history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=200,
        batch_size=8,
        callbacks=[early, reduce_lr],
        verbose=2
)

#
_____
_____
# 6) Learning curves
#
_____
_____
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

axes[0].plot(history.history["loss"],     "g.-", label="Train loss")
axes[0].plot(history.history["val_loss"], "b.-", label="Val loss")
axes[0].set_title("Loss"); axes[0].set_xlabel("Epoch")
axes[0].grid(True); axes[0].legend()

axes[1].plot(history.history["accuracy"],     "g.-", label="Train acc")
axes[1].plot(history.history["val_accuracy"], "b.-", label="Val acc")
axes[1].set_title("Accuracy"); axes[1].set_xlabel("Epoch")
axes[1].grid(True); axes[1].legend()

plt.tight_layout()
plt.savefig(os.path.join(OUT_DIR, "training_curves.png"), dpi=150)
plt.show()

#
_____
_____
# 7) Test evaluation + confusion matrix
#
```

```python
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
best_val_acc  = float(np.max(history.history.get("val_accuracy",
[np.nan])))
best_val_loss = float(np.min(history.history.get("val_loss",
[np.nan])))

print(f"\nBest val_acc  = {best_val_acc:.4f} | Best val_loss =
{best_val_loss:.4f}")
print(f"Test accuracy = {test_acc:.4f}     | Test loss      =
{test_loss:.4f}")

y_prob = model.predict(X_test, verbose=0)
y_pred = y_prob.argmax(axis=1)
y_true = y_test.argmax(axis=1)

if SKLEARN_OK:
    cm   = confusion_matrix(y_true, y_pred,
labels=list(range(NUM_GESTURES)))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=GESTURES)
    disp.plot(cmap="Blues", values_format="d")
    plt.title("Confusion Matrix (Test Set)")
    plt.savefig(os.path.join(OUT_DIR, "confusion_matrix.png"), dpi=150)
    plt.show()
    print("\nConfusion matrix:")
    print(cm)
else:
    print("scikit-learn not found — skipping confusion matrix.")

# Class probabilities over test windows (sanity check)
plt.figure(figsize=(14, 4))
for c_idx, cname in enumerate(GESTURES):
    plt.plot(y_prob[:, c_idx], ".-", label=f"P({cname})")
plt.title("Class probabilities — test windows")
plt.xlabel("Window index"); plt.ylabel("Probability")
plt.grid(True); plt.legend(); plt.show()

#


# 8) Export: TFLite (float32) + normalization.json
#


# --- TFLite ---
converter    = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_bytes = converter.convert()

tflite_path = os.path.join(OUT_DIR, "gesture_model.tflite")
```

```python
with open(tflite_path, "wb") as f:
    f.write(tflite_bytes)
print(f"\nTFLite saved: {tflite_path}
({os.path.getsize(tflite_path):,} bytes)")

# --- normalization.json ---
# The model server MUST apply the same per-channel z-score before
inference.
# Load this file on the RPi and apply:
#   x_norm = (x_raw - channel_mean) / channel_std   then clip to [-5, 5]
norm = {
    "mode":          "channelwise_zscore",
    "channels":      CHANNEL_SUFFIXES,              #
["ax","ay","az","gx","gy","gz"]
    "channel_mean": ch_mean.reshape(-1).tolist(),   # 6 values
    "channel_std":  ch_std.reshape(-1).tolist(),    # 6 values
    "clip":          [-5.0, 5.0],
    "window_size":   SAMPLES_PER_GESTURE,
    "sample_rate_hz": 100,
    "label_map":     {str(i): g for i, g in enumerate(GESTURES)}
}

norm_path = os.path.join(OUT_DIR, "normalization.json")
with open(norm_path, "w") as f:
    json.dump(norm, f, indent=2)
print(f"Normalization saved: {norm_path}")

#
_____
_____
# 9) Quick inference sanity check
#    Verifies the exported TFLite model gives identical outputs to
Keras.
#
_____
_____
interp = tf.lite.Interpreter(model_path=tflite_path)
interp.allocate_tensors()
inp_det  = interp.get_input_details()[0]
out_det  = interp.get_output_details()[0]

sample    = X_test[0:1].astype(np.float32)
keras_out = model.predict(sample, verbose=0)

interp.set_tensor(inp_det["index"], sample)
interp.invoke()
tflite_out = interp.get_tensor(out_det["index"])

print(f"\nSanity check (window 0):")
print(f"  Keras  : {keras_out[0]}")
print(f"  TFLite : {tflite_out[0]}")
```

```python
print(f"  Max diff: {np.abs(keras_out - tflite_out).max():.2e}")
print("\nTraining complete.")
```

### 3.3.3 Model & Server Testing

The model testing process is designed around receiving real-time IMU data frames sent from the controller via LoRa on the gateway side and feeding them directly into the trained TensorFlow Lite model. At this stage, the test code performed frame-based verification by reading the frames coming from the LoRa module via the serial port. For each frame, the header (address, channel, and MAGIC field) was checked, the frame counter field was verified, and when all frames were received intact, the relevant window was reconstructed. After the window was successfully created, the raw 16-bit IMU data was converted to float32 format and normalised using the global mean and standard deviation values used during the training phase. This ensured that the model input distribution during the testing phase was consistent with the data distribution during the training phase. The (1, 96, 6) dimensional data tensor obtained after normalisation was fed directly into the TensorFlow Lite model.

The model produced three probability values for each window:

- move_prob
- rest_prob
- shake_prob.

After all, these outputs are generated with their timestamps, the one with higher probability is assigned as "predicted". These five values (timestamp, 3 probabilities and predicted) are lastly sent to the cloud firebase server over Wi-Fi.

These values were obtained as softmax outputs and printed to the terminal screen after each window. In the controlled motion scenarios performed during the testing process, it was observed that the model outputs were consistent with the motion patterns during the data collection phase. In the rest state, the rest_prob value was dominant; in the Move scenario, move_prob increased significantly; and in the Shake scenario, shake_prob increased significantly compared to the other classes. It was particularly observed that the high variance in the gyroscope axes during the shake scenario was directly reflected in the model output.

The consistency of the probability values after each window and their alignment with the expected physical behaviour demonstrated the quality of the data set and the accuracy of the model training. Additionally, the terminal outputs were saved to a CSV file and stored with a timestamp. This provided both real-time observation and the possibility for subsequent analysis.

server.py

```python
#!/usr/bin/env python3
import sys
import struct
import time
import json
import csv
import argparse
import serial
```

```python
import RPi.GPIO as GPIO
from typing import List, Tuple, Optional
import numpy as np
from smbus2 import SMBus
import tensorflow as tf
import firebase_admin
from firebase_admin import credentials, db


#LoRa Globals
PIN_LORA_M0 = 11
PIN_LORA_M1 = 13
PIN_LORA_AUX = 15


lora_params = bytes([0xC0, 0x00, 0x00, 0x1A, 0x06, 0x40])
cmd_version = bytes([0xC3, 0xC3, 0xC3])
cmd_params = bytes([0xC1, 0xC1, 0xC1])


uart = serial.Serial(
        port= "/dev/serial0",
        baudrate = 9600,
        timeout = 1
    )
MAGIC = b"TE"


# IMU Window Frame format:
# ADDH(1) ADDL(1) CHAN(1) MAGIC(2) FID(1) FC(1) PAYLOAD(48)
HEADER_LEN = 1 + 1 + 1 + 2 + 1 + 1
PAYLOAD_LEN = 48
FRAME_LEN = HEADER_LEN + PAYLOAD_LEN


#HELPERS
def raise_error(msg:str, code: int, final_act, *args, **kwargs):
    print(f"[ERROR:], {msg}", file=sys.stderr)
    final_act(*args, **kwargs)
    raise SystemExit(code)


def wait_until_pin(pin, state, timeout_s: float) -> bool:
    t0 = time.monotonic()
    while (time.monotonic() - t0) < timeout_s:
        if GPIO.input(pin) == state:
            return True
        time.sleep(0.002)
    return False


# ---------------- MPU6050 minimal I2C driver ----------------


MPU_ADDR_DEFAULT = 0x68


REG_PWR_MGMT_1     = 0x6B
REG_CONFIG         = 0x1A
REG_SMPLRT_DIV     = 0x19
```

```python
REG_GYRO_CONFIG    = 0x1B
REG_ACCEL_CONFIG   = 0x1C
REG_ACCEL_XOUT_H   = 0x3B   # 14-byte burst


class MPU6050:
    def __init__(self, bus_id=1, addr=MPU_ADDR_DEFAULT):
        self.addr = addr
        self.bus = SMBus(bus_id)

    def write_reg(self, reg, val):
        self.bus.write_byte_data(self.addr, reg, val & 0xFF)

    def read_block(self, reg, n):
        return self.bus.read_i2c_block_data(self.addr, reg, n)

    @staticmethod
    def _to_i16(msb, lsb):
        v = (msb << 8) | lsb
        return v - 65536 if v & 0x8000 else v

    def initialize(self, accel_range="2g", gyro_range="250dps"):
        # Wake up device
        self.write_reg(REG_PWR_MGMT_1, 0x00)
        time.sleep(0.05)

        # Moderate DLPF (helps noise)
        self.write_reg(REG_CONFIG, 0x03)

        # Sample rate divider (software loop enforces Fs)
        self.write_reg(REG_SMPLRT_DIV, 0x00)

        accel_map = {"2g": 0x00, "4g": 0x08, "8g": 0x10, "16g": 0x18}
        gyro_map  = {"250dps": 0x00, "500dps": 0x08, "1000dps": 0x10,
"2000dps": 0x18}

        if accel_range not in accel_map:
            raise ValueError("Unsupported accel_range")
        if gyro_range not in gyro_map:
            raise ValueError("Unsupported gyro_range")

        self.write_reg(REG_ACCEL_CONFIG, accel_map[accel_range])
        self.write_reg(REG_GYRO_CONFIG, gyro_map[gyro_range])

    def read_raw6(self):
        b = self.read_block(REG_ACCEL_XOUT_H, 14)
        ax = self._to_i16(b[0], b[1])
        ay = self._to_i16(b[2], b[3])
        az = self._to_i16(b[4], b[5])
        gx = self._to_i16(b[8], b[9])
        gy = self._to_i16(b[10], b[11])
```

```python
            gz = self._to_i16(b[12], b[13])
            return ax, ay, az, gx, gy, gz

    def close(self):
        try:
            self.bus.close()
        except Exception:
            pass


# --------------- TFLite engine ---------------
class TFLiteEngine:
    def __init__(self, model_path: str, show_io: bool = False):
        self.interpreter = tf.lite.Interpreter(model_path=model_path)
        self.interpreter.allocate_tensors()
        self.in_details = self.interpreter.get_input_details()[0]
        self.out_details = self.interpreter.get_output_details()[0]
        self.in_idx = self.in_details["index"]
        self.out_idx = self.out_details["index"]

        if show_io:
            print("TFLite input details:", self.in_details)
            print("TFLite output details:", self.out_details)

    def predict(self, x: np.ndarray) -> np.ndarray:
        # x must be float32 with shape (1,96,6)
        self.interpreter.set_tensor(self.in_idx, x.astype(np.float32))
        self.interpreter.invoke()
        y = self.interpreter.get_tensor(self.out_idx)
        return np.asarray(y[0], dtype=np.float32).reshape(-1)


# --------------- Helpers ---------------
def cleanup():
    uart.close()
    GPIO.cleanup()

def int16_to_float_acc(raw: int, accel_range_g: int = 2) -> float:
    # MPU6050: ±2g => 16384 LSB/g, ±4g => 8192, ±8g => 4096, ±16g =>
2048
    lsb_per_g = {2: 16384.0, 4: 8192.0, 8: 4096.0, 16:
2048.0}[accel_range_g]
    return raw / lsb_per_g


def int16_to_float_gyro(raw: int, gyro_range_dps: int = 250) -> float:
    # MPU6050: ±250 => 131 LSB/(°/s), ±500 => 65.5, ±1000 => 32.8,
±2000 => 16.4
    lsb_per_dps = {250: 131.0, 500: 65.5, 1000: 32.8, 2000:
16.4}[gyro_range_dps]
    return raw / lsb_per_dps
```

```python
def _read_exact(ser: serial.Serial, n: int, timeout_s: float) -> bytes:
    """Read exactly n bytes or raise TimeoutError."""
    deadline = time.time() + timeout_s
    buf = bytearray()
    while len(buf) < n:
        if time.time() > deadline:
            raise TimeoutError(f"Timeout while reading {n} bytes (got
{len(buf)})")
        chunk = ser.read(n - len(buf))
        if chunk:
            buf.extend(chunk)
    return bytes(buf)


def _sync_to_magic(ser: serial.Serial, timeout_s: float) -> None:
    """
    Resync by searching for MAGIC in stream and positioning read so
that
    next reads start at frame boundary (best-effort).
    """
    deadline = time.time() + timeout_s
    window = bytearray()

    while time.time() < deadline:
        b = ser.read(1)
        if not b:
            continue
        window += b
        if len(window) > 64:
            window = window[-64:]

        idx = window.find(MAGIC)
        if idx == -1:
            continue

        # MAGIC is at offset 3 in a well-formed frame:
        # [0]=ADDH [1]=ADDL [2]=CHAN [3..4]=MAGIC
        # If it is found that MAGIC somewhere in the sliding window,
attempt to align.
        # window are the tail of previous stream; after detecting, just
return.
        return

    raise TimeoutError("Failed to sync to MAGIC within timeout")


def recv_imu_window(
    ser: serial.Serial,
    A: int = 96,
```

```python
    samples_per_frame: int = 4,   # 48 payload / (6*2) = 4 samples
    expect_addh: int = 0x00,
    expect_addl: int = 0x00,
    expect_chan: int = 0x06,
    timeout_s: float = 2.0,
) -> List[Tuple[int, int, int, int, int, int]]:
    """
    Receives a full IMU window and returns it as a list of A samples.
    Each sample is (ax, ay, az, gx, gy, gz) in int16 raw.

    Frame checks:
    - ADDH/ADDL/CHAN match
    - MAGIC == b"TE"
    - FC sequence (0..frame_count-1)
    - Consistent FID across frames
    """
    if (PAYLOAD_LEN % (6 * 2)) != 0:
        raise ValueError("PAYLOAD_LEN is not a multiple of one IMU
sample (12 bytes).")

    frame_count = (A + samples_per_frame - 1) // samples_per_frame
    out: List[Tuple[int, int, int, int, int, int]] = [None] * A  #
type: ignore

    # Try to sync first (best-effort)
    _sync_to_magic(ser, timeout_s=timeout_s)

    fid_expected: Optional[int] = None
    samples_written = 0

    for fc_expected in range(frame_count):
        # Read one full frame
        frame = _read_exact(ser, FRAME_LEN, timeout_s=timeout_s)

        addh, addl, chan = frame[0], frame[1], frame[2]
        magic = frame[3:5]
        fid = frame[5]
        fc = frame[6]
        payload = frame[7:7 + PAYLOAD_LEN]

        # Validate header
        if (addh != expect_addh) or (addl != expect_addl) or (chan !=
expect_chan) or (magic != MAGIC):
            raise ValueError(
                f"Header mismatch: ADDH/ADDL/CHAN/MAGIC = "
                f"{addh:02X}/{addl:02X}/{chan:02X}/{magic!r}"
            )

        # Validate FID continuity
        if fid_expected is None:
            fid_expected = fid
```

```python
        elif fid != fid_expected:
            raise ValueError(f"FID mismatch: got {fid}, expected
{fid_expected}")

        # Validate FC ordering
        if fc != fc_expected:
            raise ValueError(f"FC mismatch: got {fc}, expected
{fc_expected}")

        # Parse payload: little-endian 6x int16 per sample
        # Each sample is 12 bytes -> '<hhhhhh'
        offset = 0
        for _ in range(samples_per_frame):
            if samples_written >= A:
                break
            ax, ay, az, gx, gy, gz = struct.unpack_from("<hhhhhh",
payload, offset)
            out[samples_written] = (ax, ay, az, gx, gy, gz)  # type:
ignore
            samples_written += 1
            offset += 12

    if samples_written != A:
        raise ValueError(f"Incomplete window: got {samples_written}
samples, expected {A}")

    # Return the window safter the transfer is successful
    return out

# Confirmed output order:
LABELS = ["move", "rest", "shake"]

def load_norm_json(path: str):
    """
    Expects JSON with channel_mean and channel_std arrays of length 6.
    """
    with open(path, "r") as f:
        obj = json.load(f)

    # Try a few common key names to be robust
    for mean_key in ["channel_mean", "mean", "means"]:
        if mean_key in obj:
            channel_mean = obj[mean_key]
            break
    else:
        raise KeyError("Could not find channel_mean in
normalization.json")

    for std_key in ["channel_std", "std", "stds"]:
        if std_key in obj:
            channel_std = obj[std_key]
```

```python
            break
    else:
        raise KeyError("Could not find channel_std in
normalization.json")

    mean = np.array(channel_mean, dtype=np.float32).reshape(1, 1, 6)
    std  = np.array(channel_std, dtype=np.float32).reshape(1, 1, 6)

    if mean.shape[-1] != 6 or std.shape[-1] != 6:
        raise ValueError("Normalization arrays must have length 6")

    # Avoid division by zero
    std = np.where(std == 0.0, 1e-6, std)
    return mean, std

def window_summary(win_i16: np.ndarray) -> str:
    w = win_i16.astype(np.int32)
    mn = w.min(axis=0)
    mx = w.max(axis=0)
    mean = w.mean(axis=0)
    return (
        f"AX[{mn[0]},{mx[0]}] AY[{mn[1]},{mx[1]}] AZ[{mn[2]},{mx[2]}] "
        f"GX[{mn[3]},{mx[3]}] GY[{mn[4]},{mx[4]}] GZ[{mn[5]},{mx[5]}] "
        f"MEAN={mean.round(1).tolist()}"
    )

def classify(probs: np.ndarray):
    i = int(np.argmax(probs))
    return LABELS[i], float(probs[i])


# ---------------- Main ----------------
def main():
    #Parse initial arguments
    ap = argparse.ArgumentParser()
    ap.add_argument("--model", required=True, help="Path to .tflite
model")
    ap.add_argument("--norm_json", required=True, help="Path to
normalization.json containing mean/std (6 values each)")
    ap.add_argument("--bus", type=int, default=1, help="I2C bus id
(default: 1)")
    ap.add_argument("--addr", type=lambda x: int(x, 0), default=0x68,
help="MPU6050 address (default: 0x68)")
    ap.add_argument("--fs", type=float, default=100.0, help="Sampling
frequency in Hz (default: 100)")
    ap.add_argument("--A", type=int, default=96, help="Window length
(default: 96)")
    ap.add_argument("--stride", type=int, default=96, help="Stride in
samples (default: 96)")
    ap.add_argument("--log", default="", help="Optional CSV log file
path (e.g., outputs.csv)")
```

```python
    ap.add_argument("--show_io", action="store_true", help="Print
TFLite I/O tensor details once")
    args = ap.parse_args()

    #Initialize LoRa control ports (M0-M1, AUX)
    GPIO.setmode(GPIO.BOARD)
    GPIO.setwarnings(False)
    GPIO.setup(PIN_LORA_M0, GPIO.OUT)
    GPIO.setup(PIN_LORA_M1, GPIO.OUT)
    GPIO.setup(PIN_LORA_AUX, GPIO.IN, pull_up_down = GPIO.PUD_UP)

    #Wait until AUX is high (Module is ready)
    if not wait_until_pin(PIN_LORA_AUX, GPIO.HIGH, 1.0):
        raise_error("LoRa module is busy!", -1, cleanup())
    else:
        print("LoRa module is available")

    #Set module mode (11) sleep
    GPIO.output(PIN_LORA_M0, GPIO.HIGH)
    GPIO.output(PIN_LORA_M1, GPIO.HIGH)
    if not wait_until_pin(PIN_LORA_AUX, GPIO.HIGH, 1.0):
        raise_error("Cannot set parameters", -1, cleanup())
    else:
        print("Parameters are set")

    #Send lora configuration parameters
    uart.reset_input_buffer()
    print("TX:", lora_params.hex(" "))
    uart.write(lora_params)
    uart.flush()
    time.sleep(0.1)

    #Validate lora configuration parameters
    uart.reset_input_buffer()
    print("TX:", cmd_params.hex(" "))
    uart.write(cmd_params)
    uart.flush()
    rx = uart.read(6)
    if rx != lora_params:
        raise_error("Unable to validate configuration parameters", -1,
cleanup())
    else:
        print("Config parameters:", rx.hex(" "))
        time.sleep(0.1)

    #Set module mode (00) transceiver
    if not wait_until_pin(PIN_LORA_AUX, GPIO.HIGH, 1.0):
        raise_error("LoRa module is busy!", -1, cleanup())
    else:
        print("LoRa module is available to transmit")
        GPIO.output(PIN_LORA_M0, GPIO.LOW)
```

```python
        GPIO.output(PIN_LORA_M1, GPIO.LOW)
        time.sleep(0.01)

    #INIT FIREBASE SERVER
    #Fetch the service account key
    cred = credentials.Certificate("serviceAccountKey.json")

    #Initialize the app with a service account then grant admin
privileges
    firebase_admin.initialize_app(cred, {
        'databaseURL':
"https://xiao-pet-tracker-controller-default-rtdb.europe-west1.firebase
database.app"
    })

    #Get imu reference
    controller_ref = db.reference("controllers/1")

    if not args.model.lower().endswith(".tflite"):
        raise ValueError("Model must be a .tflite file")

    if args.A != 96:
        print("Warning: The model is confirmed for A=96. Make sure your
model input matches A.")

    # Load global normalization
    channel_mean, channel_std = load_norm_json(args.norm_json)
    # channel_mean/std shape: (1,1,6)

    # Init IMU
    mpu = MPU6050(bus_id=args.bus, addr=args.addr)
    mpu.initialize(accel_range="2g", gyro_range="250dps")

    # Init model
    engine = TFLiteEngine(args.model, show_io=args.show_io)

    # Optional CSV logging
    csv_f = None
    csv_w = None
    if args.log:
        csv_f = open(args.log, "w", newline="")
        csv_w = csv.writer(csv_f)
        csv_w.writerow(["timestamp", "window_id", "move_prob",
"rest_prob", "shake_prob", "predicted", "confidence"])

    A = args.A
    fs = args.fs
    period = 1.0 / fs

    window_id = 0
```

```python
    print("Live TFLite inference started.")
    print(f"Model: {args.model}")
    print(f"Normalization: {args.norm_json} (global mean/std)")
    print(f"Fs={fs} Hz, A={A}, stride={args.stride}")
    print("Output order: move, rest, shake")
    if args.log:
        print(f"CSV logging enabled: {args.log}")
    print("Press Ctrl+C to stop.\n")

    try:
        next_t = time.perf_counter()

        while True:
            win = recv_imu_window(
                ser,
                A=96,
                samples_per_frame=4,
                expect_addh=0x00,
                expect_addl=0x00,
                expect_chan=0x06,
                timeout_s=3.0,
            )

            """
            ax, ay, az, gx, gy, gz = win[0]

            ax_f = int16_to_float_acc(ax, accel_range_g)
            ay_f = int16_to_float_acc(ay, accel_range_g)
            az_f = int16_to_float_acc(az, accel_range_g)

            gx_f = int16_to_float_gyro(gx, gyro_range_dps)
            gy_f = int16_to_float_gyro(gy, gyro_range_dps)
            gz_f = int16_to_float_gyro(gz, gyro_range_dps)

            print(
                f"Data received! : "
                f"ax={ax_f:.4f}g, ay={ay_f:.4f}g, az={az_f:.4f}g, "
                f"gx={gx_f:.2f}dps, gy={gy_f:.2f}dps, gz={gz_f:.2f}dps"
            )
            """
            # Enforce sampling rate
            now = time.perf_counter()
            if now < next_t:
                time.sleep(next_t - now)
            next_t += period

            # Global normalization (per-channel)
            win_f32 = win.astype(np.float32).reshape(1, A, 6)
            x = (win_f32 - channel_mean) / channel_std  # (1,96,6)
float32
```

```python
                # Inference (already softmax)
                probs = engine.predict(x)
                if probs.size != 3:
                    raise RuntimeError(f"Expected 3 outputs, got
{probs.size}")

                move_prob, rest_prob, shake_prob = float(probs[0]),
float(probs[1]), float(probs[2])
                pred, conf = classify(probs)

                ts = time.time()

                print(f"[Window {window_id}] {window_summary(win)}")
                print(f"  move_prob={move_prob:.4f}
rest_prob={rest_prob:.4f}   shake_prob={shake_prob:.4f}")
                print(f"  predicted={pred} ({conf*100:.1f}%)")
                print("-" * 90)

                #Send the updated status to database
                controller_ref.update({
                    "move_prob": move_prob,
                    "shake_prob": shake_prob,
                    "rest_prob": rest_prob,
                    "predicted": pred,
                    "timestamp": str(ts)
                })

                if csv_w is not None:
                    csv_w.writerow([f"{ts:.3f}", window_id,
f"{move_prob:.6f}", f"{rest_prob:.6f}",
                                    f"{shake_prob:.6f}", pred,
f"{conf:.6f}"])
                    csv_f.flush()

                window_id += 1

    except KeyboardInterrupt:
        print("\nStopped by user.")
    finally:
        mpu.close()
        if csv_f:
            csv_f.close()


if __name__ == "__main__":
    main()
```

test_outputs.xlsx

Test Outputs (Change their extensions as .csv)

[project_test.mp4](#)

**Video 5** Gateway & Server Test Video

# 4.Discussion

The implemented system has successfully demonstrated an end-to-end motion classification architecture. On the controller side, low-power mode standby, window-based data collection when motion is detected, and transmission to the gateway via LoRa; on the gateway side, real-time model execution and writing results to the cloud have all operated stably.

A clear distinction was observed between the move, rest, and shake classes in the model outputs. In particular, the model consistently distinguished between low-variance acceleration data in the rest state and high angular velocity values in the shake state. As the inference being performed on the gateway, raw IMU data was not sent to the cloud; only classification outputs were transmitted. This approach reduced bandwidth usage and increased the system's scalability. Furthermore, the STOP mode and motion-triggered measurement approach significantly reduced the average power consumption on the controller side.

The dataset created during the project process contains a limited number of windows. Saturation problems were observed, particularly in the shake class, and the sensor range settings were optimised. The small data set is a factor that could limit the model's generalisation capacity.

Furthermore, although LoRa's 2.4 kbps air data rate provides an advantage in terms of reliability, it has extended the window transmission time. This situation has particularly required the division of 96-sample data windows into multiple frames. A balance had to be established between the air data rate, spreading factor, and FEC parameters. In LoRa communication between the controller and the gateway, synchronisation loss and timing problems occurred in the multi-frame structure, and a frame counter-based reconfiguration mechanism was developed. This process complicated the design of the communication layer. On the model side, although it was sufficient in terms of operating performance, a balance had to be struck between power consumption and response time.

The system functionality can be improved in various ways. For example, ZigBee, 2.4 GHz RF or Wi-Fi-based solutions can be used instead of LoRa. Particularly in short-range applications, higher data rates may allow for larger window sizes or more frequent data transmission. However, for longer ranges, the cost of the required trade-off must be considered and calculated. Different model architectures (LSTM, 1D temporal convolution) can be tested. Furthermore, accuracy can be improved through data augmentation, larger datasets, and different normalisation techniques. However, unlike the controller, the gateway operates continuously. Switching the gateway to low-power mode until data arrives or transitioning to a lower-power edge device can improve system efficiency.

For model training, several directions could further improve the model in future iterations. The current dataset was recorded by a single person in a controlled desk environment. With a larger dataset covering multiple users, gesture intensities, and controller orientations, more complex architectures such as LSTM or Temporal Convolutional Networks (TCN) could be explored. LSTMs model long-range temporal dependencies and may better capture the sequential dynamics of movement patterns; however, they require substantially more data to generalise well and are less efficient on embedded hardware than Conv1D. With 1,000+ windows per class, a hybrid architecture (Conv1D feature extractor + LSTM temporal layer) would be a natural next step.

The current TFLite model uses float32 weights. Quantisation-aware training could produce an INT8 model, reducing memory footprint by approximately 4× and accelerating inference on the Raspberry

Pi. This would also enable deployment on more constrained microcontrollers. The expected accuracy drop is 1–2 percentage points, which is acceptable given the current 96.6% baseline.

The current model is trained offline and deployed as a fixed artefact. For deployment on actual animal subjects, where motion dynamics differ from handheld recording, a lightweight fine-tuning step using a small number of animal-specific windows could substantially improve accuracy without full retraining. Transfer learning — freezing the convolutional layers and retraining only the dense head on new data — would be a practical approach given the limited compute available on the gateway.

The three-class model (Move, Rest, Shake) covers the primary behavioural states of interest. Future versions could extend to additional classes such as Eat, Play, or Sleep, provided sufficient labelled data is available. The modular architecture makes extension straightforward: only the output layer size and training data need to change.

# 5. Conclusion & Outlook

In this study, a low-power IMU-based motion classification system was designed and an end-to-end IoT architecture was implemented. The system integrates sensor measurement, wireless data transmission, edge inference, and cloud data management layers. Energy efficiency was achieved through a motion-triggered measurement approach, reliable data transmission was realized using LoRa-based communication, and real-time classification was successfully implemented using the TensorFlow Lite model.

Future work plans include improving model performance with larger datasets, evaluating different communication technologies, and transforming the system into a fully portable edge-cloud architecture. Furthermore, hardware optimization and detailed power profile analysis can further improve battery life. In conclusion, the developed system offers a scalable and modular solution for IoT applications requiring low power consumption and real-time motion detection.

# 6. References

European Telecommunications Standards Institute (ETSI). 2018. "Short Range Devices (SRD) operating in the frequency range 25 MHz to 1 000 MHz; Part 2: Harmonised Standard for access to radio spectrum for non specific radio equipment". Visited: 09.02.2026. Available at: https://www.etsi.org/deliver/etsi_en/300200_300299/30022002/03.02.01_60/en_30022002v030201p.pdf

M. Alsaaod, J.J. Niederhauser, G. Beer, N. Zehner, G. Schuepbach-Regula, A. Steiner. 2015. "Development and validation of a novel pedometer algorithm to quantify extended characteristics of the locomotor behavior of dairy cows, Journal of Dairy Science". Volume 98. Issue 9. pp. 6236-6242. ISSN 0022-0302. DOI: https://doi.org/10.3168/jds.2015-9657.

Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D. J. (2021). 1D convolutional neural networks and applications: A survey. Mechanical Systems and Signal Processing, 151, 107398. https://doi.org/10.1016/j.ymssp.2020.107398

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. Proceedings of the 32nd International Conference on Machine Learning (ICML),

448–456. https://proceedings.mlr.press/v37/ioffe15.html