2024

Disability Care Datalogger

FIRMWARE DOCUMENTATION

THOMAS

Table of Contents

Developer Guide

This section serves as a comprehensive guide for developers taking over or extending this project. It includes instructions on how to contribute to the codebase, modify existing functionalities, and expand the project with new features or sensors.

Project Overview

The project is built using a combination of Arduino firmware, open-source libraries, and Python-based machine learning scripts. It interfaces with MongoDB for data storage and processing, making it easy to customize and scale. The hardware includes a partially populated board with available I/O pins for additional sensors like MEMS microphones or addressable LEDs. The machine learning components are also open-source, allowing for model modifications and retraining based on new requirements.

The project can be modified using any IDE that supports .ino sketches and .cpp/.h files (e.g., Arduino IDE, VS Code with PlatformIO). Python scripts for AI processing and training are fully compatible with common data science environments such as Jupyter Notebook and PyCharm.

1. Project Structure and Key Files

    Firmware Directory (Firmware/):

        DCDLFirmwareFinalRev.ino: The primary sketch that initializes sensors, handles data acquisition, and communicates with MongoDB.
        HeartRate5.cpp and HeartRate5.h: Libraries for handling sensor inputs and pre-processing.
        Other Official open source libraries are also included and can be seen in the #Include declarations at the top of the DCDLFirmwareFinalRev.ino file.
        Note: All .cpp and .h files should follow a modular structure, making it easy to extend functionality without altering the main sketch.

    Machine Learning Directory (ML/):

        regressionmodel.py: Contains the script to train the RandomForestRegressor model. This script loads the training data (output.csv), preprocesses it, and trains the model. This model can be retrained with more info if required and will output a pkl for use with the main script.
        FinalizedModelFirmwareEnd.py: This script loads the trained model(pkl) and handles data processing and signal evaluation based on MongoDB entries.

RandomForrestModelFirmWareAutoTune.pkl: The pre-trained machine learning model saved using joblib. This file is loaded in MainProcessingModel.py to make predictions.
Note: Any modifications to the ML logic should be reflected in RandomForrestModelFirmWareAutoTune.py and the associated functions otherwise errors will arise with reading formatting etc.

Configuration Files:

Config.h: Contains configuration parameters like Wi-Fi credentials, MongoDB URI, and API keys.

2. Getting Started: Development Setup

Prerequisites:

Arduino IDE or VS Code with PlatformIO (for editing and uploading firmware).
Python 3.7+ (for executing the machine learning scripts).
MongoDB Atlas account and cluster set up (refer to the Cloud Integration Documentation for details).

Setting Up the Development Environment:

Arduino IDE:

- This is outlined in the **Installation Guide** on how to install the correct processor type and relative libraries. ○ Python Environment:
- Create a virtual environment and install the dependencies following the python documentation. Firmware Configuration:

Update the Config.h file with:

- **Wi-Fi Credentials**: SSID and PASSWORD.
- **MongoDB URI**: Connection string for the MongoDB cluster.
- **API Key**: For secure data transmission (if applicable).

3. Contributing to the Project

Understanding the Codebase:

Review the modular structure of .cpp and .h files. Each module (e.g., sensor handling, networking) is independent and can be modified without affecting other components. These are done in small modularized functions and thus as long as the main calls are still made and the functions are not changed then they will continue to work as intended.

Familiarize yourself with the DCDLFirmwareFinalRev.ino sketch for an overview of the initialization sequence and main loop.

Adding New Functionality:

Additional Sensors:

Connect new sensors to available I/O pins on the board.

Create a new .cpp and .h file for the sensor's library (e.g., new_sensor.cpp and new_sensor.h).

Initialize the sensor in the setup() function of DCDLFirmwareFinalRev.ino and add its reading logic in the loop() function.

Machine Learning Model:

Modify the training script (regressionmodel.py) with new features or datasets.

Retrain the model and save it as RandomForrestModelFirmWareAutoTune.pkl.

Load the new model in FinalizedModelFirmwareEnd.py and update the prediction logic if necessary.

Submitting Changes:

Create a new branch in the git repository with a descriptive name (e.g., feature-add-new-sensor).

Ensure the code compiles without errors and is tested on the hardware.

Open a pull request, providing detailed descriptions of the changes made.

Bug Fixes and Issue Reporting:

For each bug fix, include a description of the issue, steps to reproduce, and how it was resolved.

Use GitHub Issues or the project's issue tracking system to log bugs, feature requests, or documentation needs.

4. Extending the Project

Hardware Extensions:

The PCB has designated spots for MEMS microphones and addressable LEDs. You can populate these components and extend the functionality by updating the corresponding libraries.
Use free I/O pins for additional sensors like temperature sensors, gyroscopes, etc. Update the hardware initialization in DCDLFirmwareFinalRev.ino accordingly.

Software Extensions:

Modify the existing .cpp libraries or add new ones for complex data processing.

Expand the AI model to include new features or retrain the model to improve accuracy.

Integration with Other Services:

The project can be integrated with cloud services like AWS IoT, Google Cloud, or Microsoft Azure. Add new libraries for these services and create a new .cpp file to handle cloud communication as well as adding to the config.h for relative certificates etc. Then the DCDLFirmwareFinalRev.ino can be modified to include these changes.

5. Documentation and Knowledge Transfer

Updating Documentation:

All code changes should be accompanied by updates to the relevant documentation files.

Use the docs/ folder to create new documentation files if needed.

Machine Learning(Firmware interaction) documentation and breakdown:

ML Model Description

The machine learning model used in this project is a **RandomForestRegressor** from the sklearn library. The RandomForestRegressor is a versatile model that can handle regression tasks by building multiple decision trees and averaging their predictions to improve accuracy and control overfitting.

- **Model Objective:**
  The model is trained to evaluate the signal quality of readings from a pulse oximeter sensor. It predicts the expected heart rate based on the LED readings (e.g., green and IR LED values) collected from the sensor.

Features:

The model uses the following feature(s):

raw_values: A series of numerical values derived from the green LED sensor readings.

Expected Output:

The model outputs a continuous value representing the predicted heart rate raw values, which helps determine the quality of the signal. This output is used to guide the tuning process of the sensor parameters.

For more information on the RandomForestRegressor, refer to the sklearn RandomForest Documentation.

Training Data and Process

Training Data: The dataset used for training is a collection of sensor readings from pulse oximeters in an open source study done by Liang, Y., Chen, Z., Liu, G. *et al.* A new, short-recorded photoplethysmogram dataset for blood pressure monitoring in China. *Sci Data* **5**, 180020 (2018). https://doi.org/10.1038/sdata.2018.20 who took samples of PPG data from a number of participants and classified their risk of heart disease, this is stored in a CSV file (output.csv). Each row contains raw_values representing the green LED readings and a corresponding Heart Rate value that serves as the target variable.

Data Preprocessing:

- The raw values in the dataset are initially parsed and converted into numerical format using the parse_raw_values function.

    - Each list of raw values is exploded into individual rows to facilitate regression.
    - The data is then cleaned by dropping any rows with missing values (`NaN`), ensuring that only valid readings are used for training.
- **Splitting the Data:** The dataset is split into training and testing sets using an 80-20 split:
    - **Training Set:** 80% of the data, used to train the RandomForestRegressor.
    - **Testing Set:** 20% of the data, used to validate the model's performance.

- **Model Training:** The RandomForestRegressor is trained using the training set, with `raw_values` as the feature and `Heart Rate` as the target. After training, the model is saved as a `.pkl` file using the `joblib` library for future use.

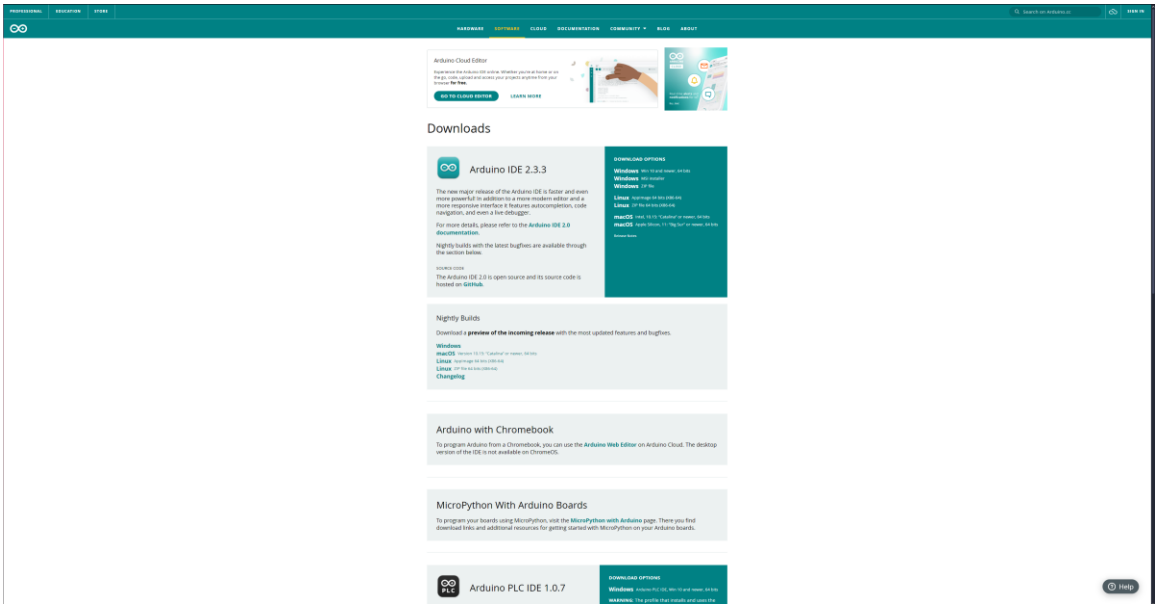
For details on the training script, refer to the code in the FinalizedModelFirmwareEnd.py and regressionmodel.py files included in the documentations folder.

Installation Guide

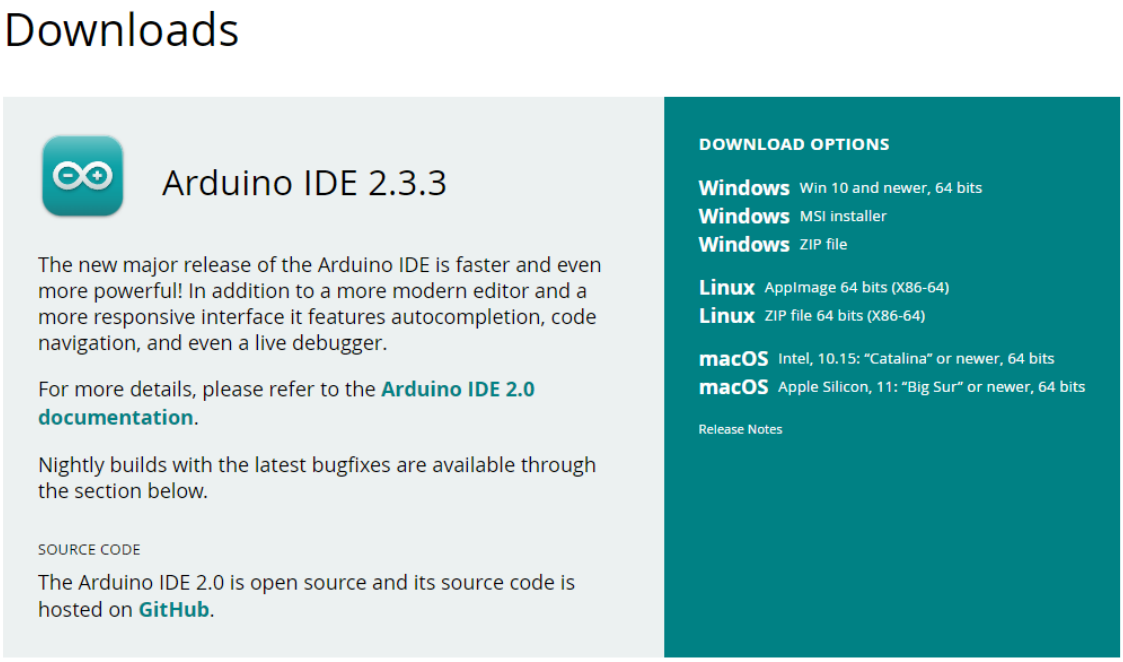
This section explains how to prepare the development environment for your project using the Arduino IDE.

1.1 Install the Arduino IDE

1. Visit the Arduino Software Page.



2. Download the appropriate version of the Arduino IDE based on your operating system (Windows, macOS, or Linux).



3. Follow the installation instructions based on your operating system:

Windows: Run the .exe file and follow the prompts to complete the installation.

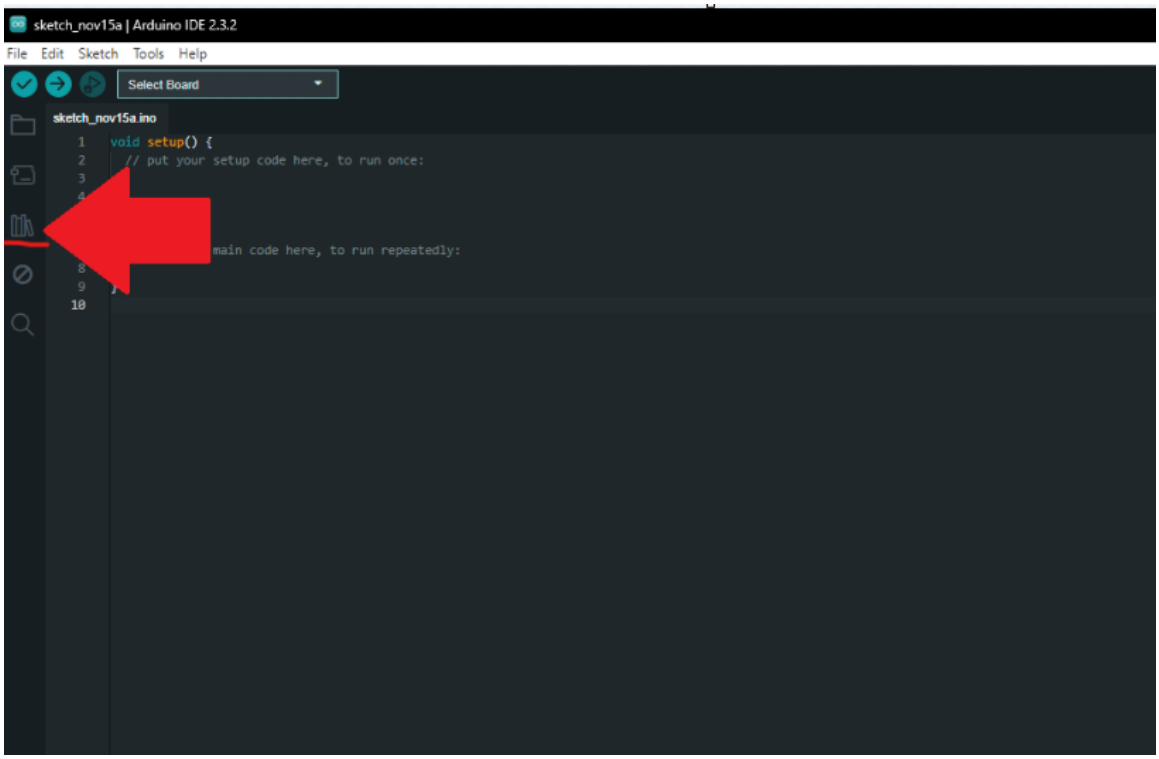
macOS: Open the .dmg file and drag the Arduino IDE to your Applications folder.

Linux: Extract the downloaded tarball and run the install.sh script.

For more detailed instructions, refer to the official Arduino Installation Guide.

1.2 Install Required Libraries

1. Open the Arduino IDE.
2. Navigate to Sketch → Include Library → Manage Libraries…



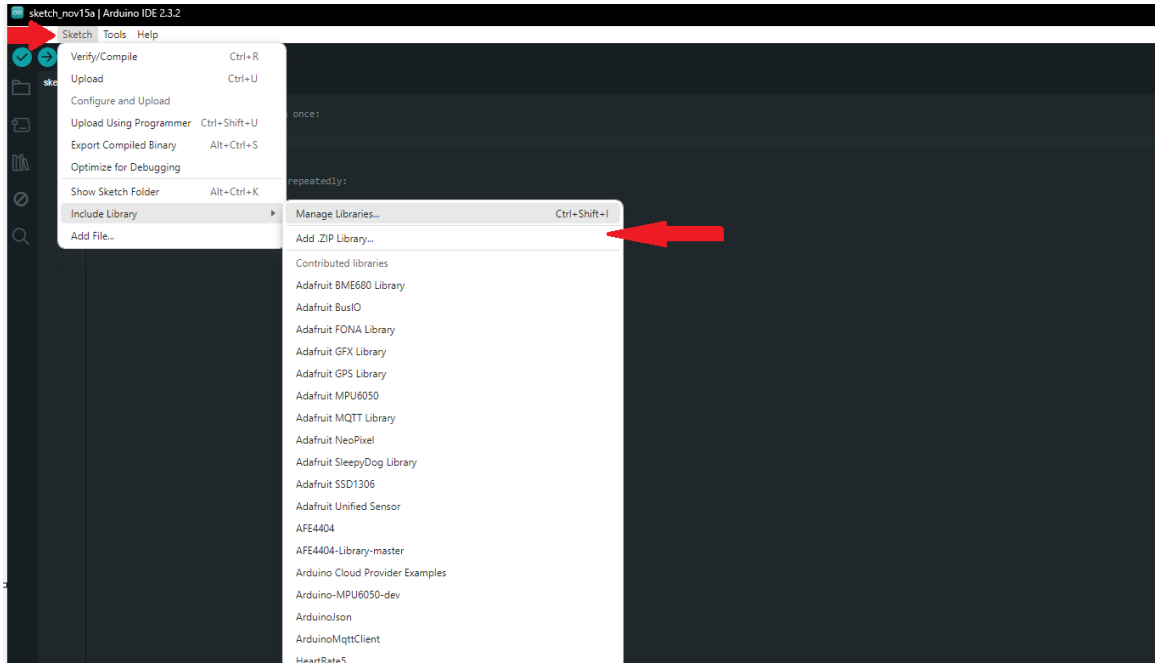
3. In the Library Manager, search for the required libraries and click "Install".
   Here's a list of the necessary libraries for this project:

```
<Wire.h>
<WiFi.h>
<HTTPClient.h>
<ArduinoJson.h>
"HeartRate5.h"
<Adafruit_MPU6050.h>
<Adafruit_Sensor.h>
<MCP79412RTC.h>
<Adafruit_NeoPixel.h>
<TimeLib.h>
```

HeartRate5 is a custom library and will need to be installed using the method below.

For custom or third-party libraries:

- Download the library as a .zip file from its source repository.
- Navigate to Sketch → Include Library → Add .ZIP Library….



- Select the .zip file and click "Open" to add it to your Arduino library.

- For Ease of use there is an included Zip file with all currently used Libraries for the firmware.

For additional details on adding libraries, refer to Adding Libraries to the Arduino IDE.

1.3 Manually Adding Libraries

Download the .zip library files from the respective repositories.

Locate your Arduino Libraries folder:

- **Windows**: Documents/Arduino/libraries
- **macOS**: ~/Documents/Arduino/libraries
- **Linux**: ~/Arduino/libraries

Extract the contents of the .zip file and copy the library folder (e.g., ArduinoJson) into the libraries folder.
Restart the Arduino IDE to ensure the libraries are loaded correctly.
For more information, visit Arduino Library Guide.

1.4 Connecting the Microcontroller and Configuring the Board

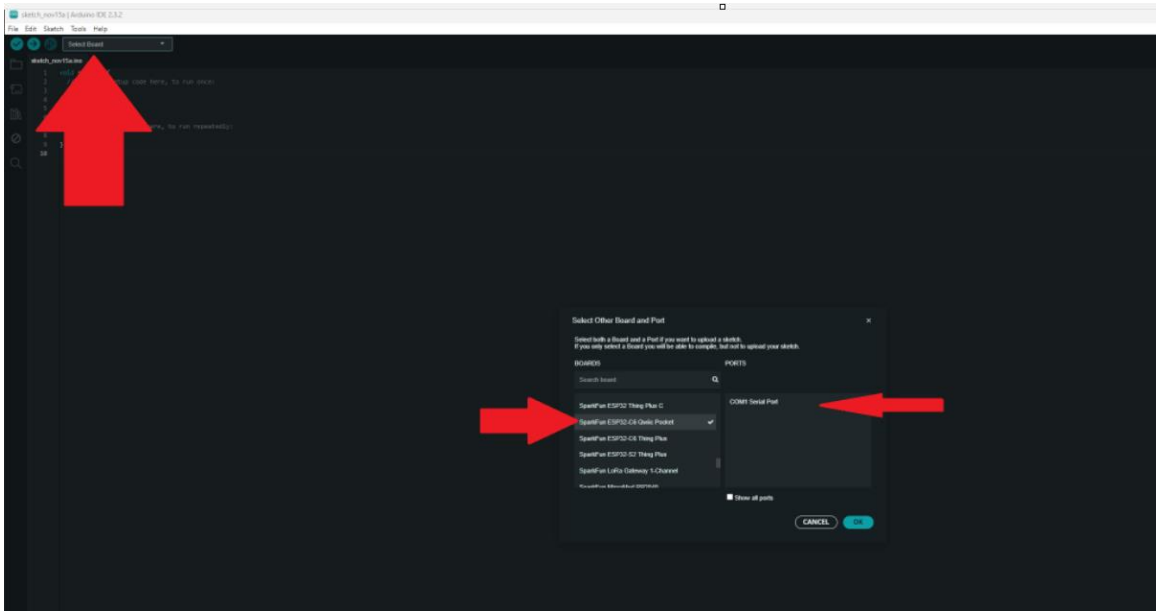
Connecting the Microcontroller and Configuring the Board

Plug your ESP32 or Arduino microcontroller into your computer using a USB cable.

As this is a custom board you will need to import the board config for the board the instructions on doing so are outlined within the ESPRESSIF documentation found at their link: https://docs.espressif.com/projects/arduinoesp32/en/latest/installing.html.

Once completed in the Arduino IDE, go to Tools → Board and select the appropriate board (Sparkfun ESP32 C6 QWIIC) :

- **ESP32**: Choose SparkFun ESP32 QWIIC board.



For detailed instructions on setting up the ESP32, refer to the ESP32 Board Installation Guide.

1.5 Upload Code to the Microcontroller

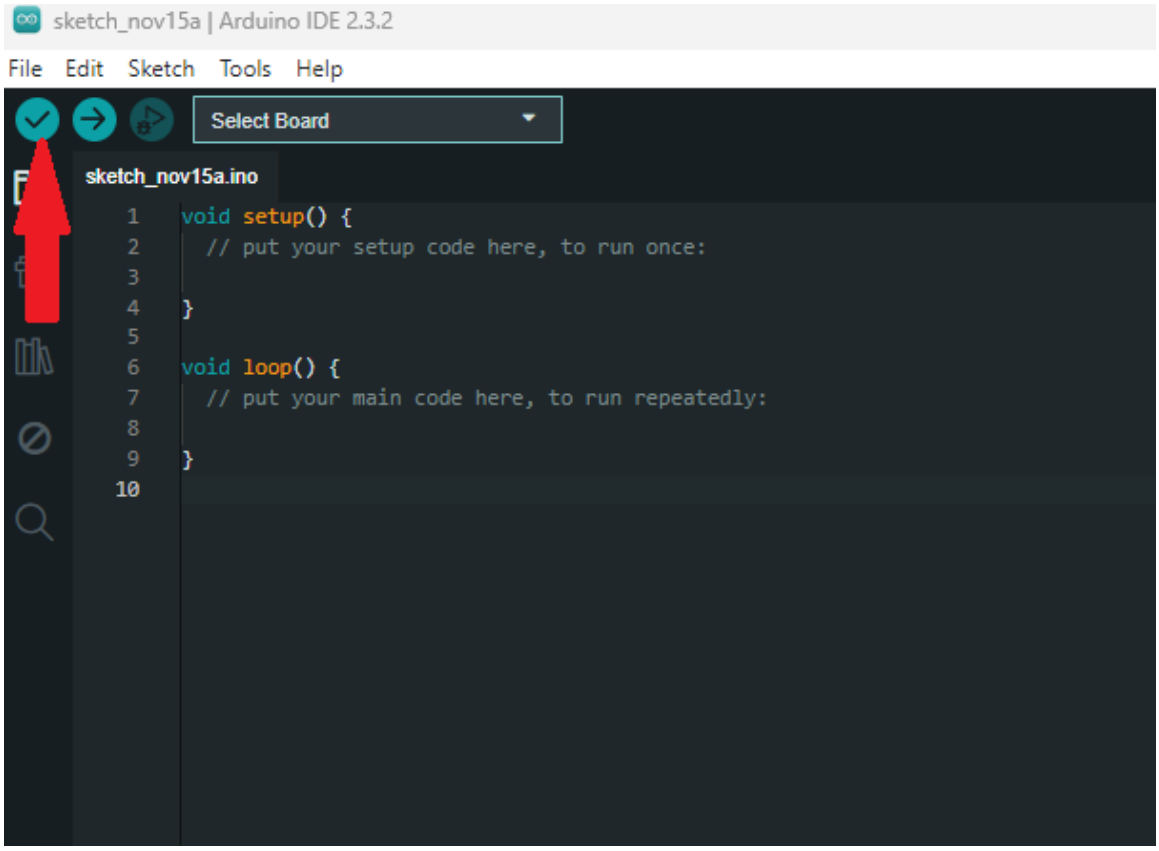
Open your project code in the Arduino IDE.

Verify the code:

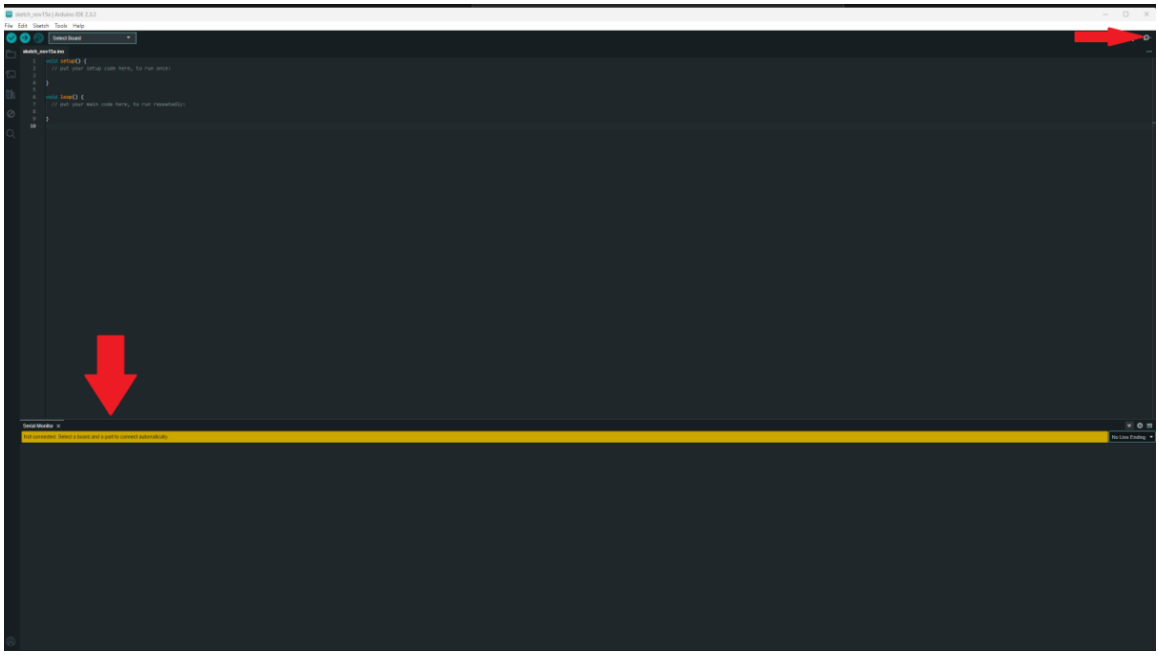
Click the checkmark icon (✓) in the top-left corner of the IDE. This will compile the code to ensure there are no errors.

Upload the code:

Click the right-arrow icon (→) next to the checkmark. The IDE will compile and upload the code to your microcontroller.



Monitor the serial output:
Open the Serial Monitor by navigating to Tools → Serial Monitor to view the realtime data being output by the microcontroller.



For more details on uploading code, refer to Uploading Code to Arduino.

1.6 Troubleshooting Common Issues

If you encounter any issues with library inclusion or code compilation, ensure that:

All libraries are installed correctly.

You have selected the correct board and port.
Your code has no syntax errors.
For additional troubleshooting, refer to Arduino Troubleshooting Guide and the ESPRESSIF troubleshooting guide.

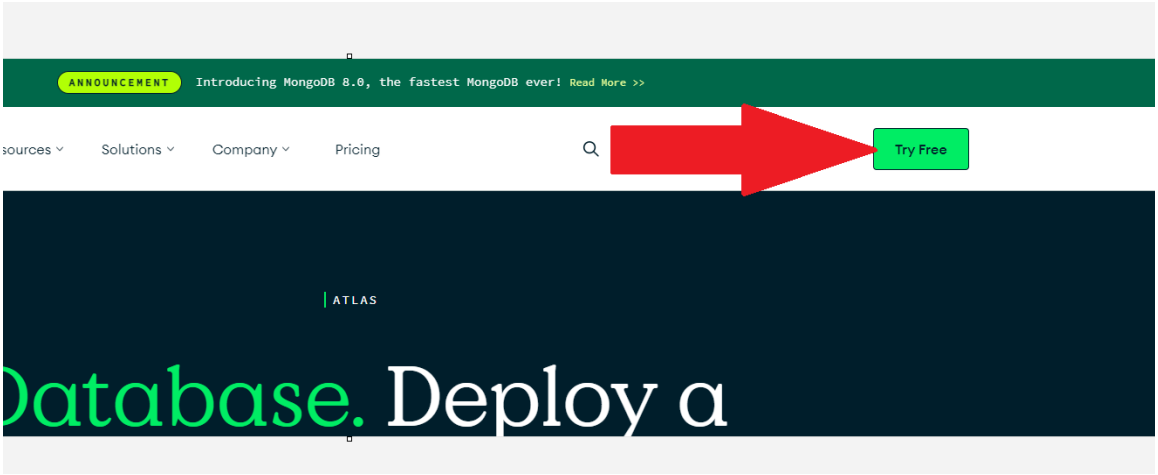
Cloud Integration Documentation

This section covers how to integrate MongoDB for data management.

MongoDB Configuration

Set Up a MongoDB Cluster.

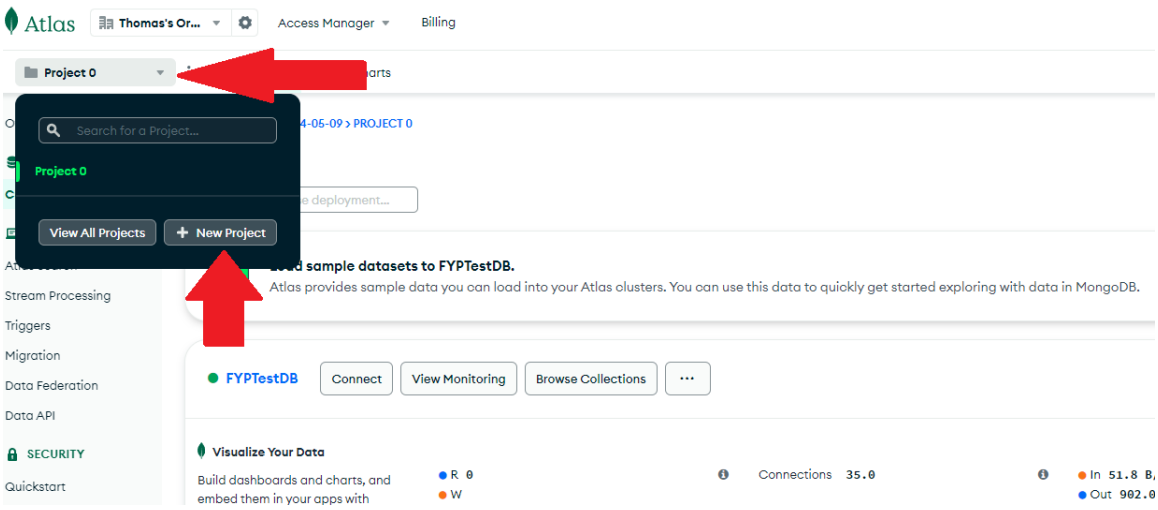
Visit the MongoDB Atlas Website and sign up for a free account.



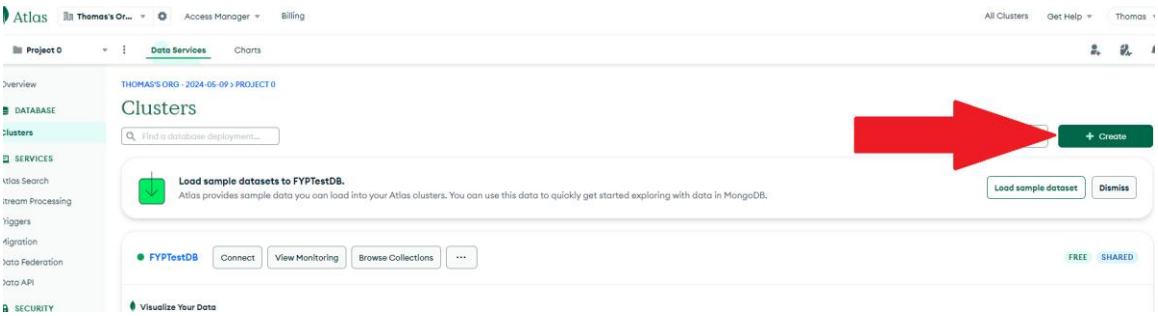
After logging in, create a new project:

Click on the "New Project" button and provide a name for your project.

Click "Create Project" to proceed.

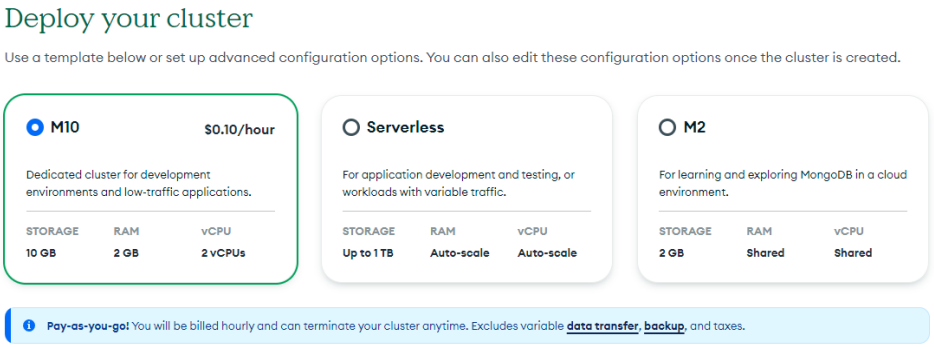


Create a new cluster:

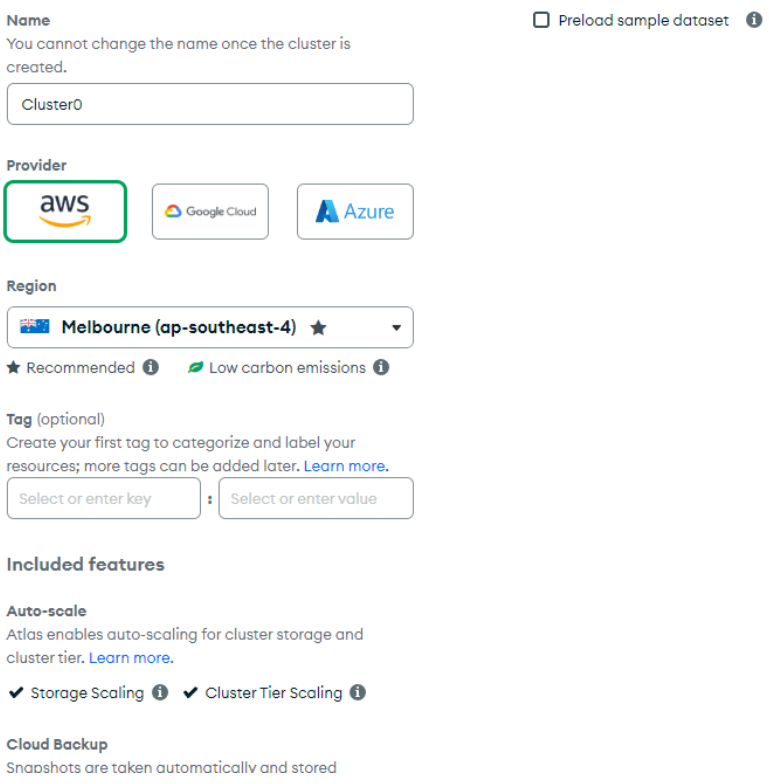


Click on the "Build a Cluster" button.

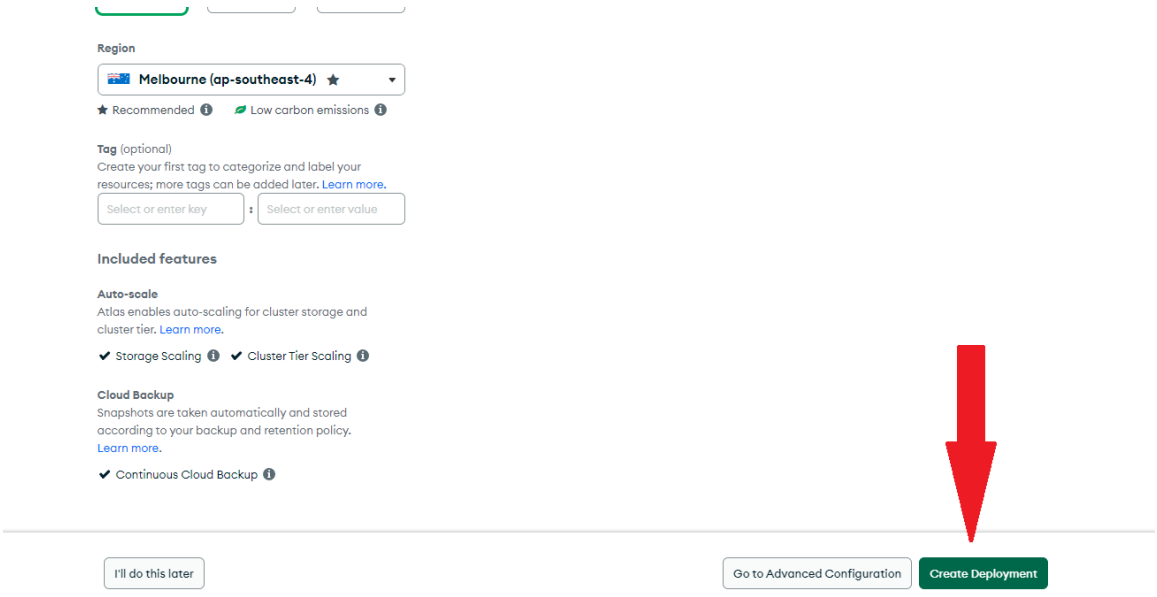
Select the free-tier cluster option for development purposes or your desired cluster type and click "Create Cluster".



Choose a cloud provider (e.g., AWS, Google Cloud, or Azure) and the preferred region.

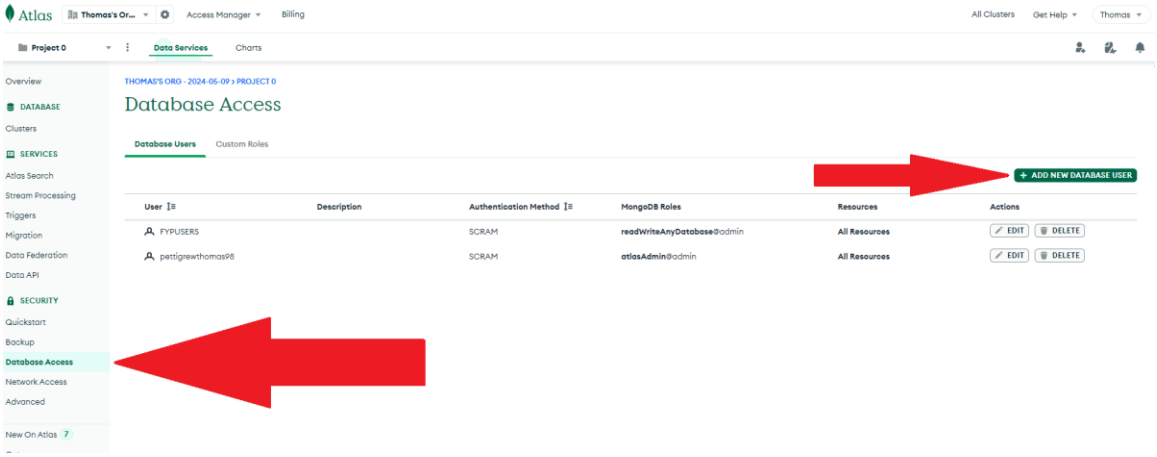


Once the cluster is created, navigate to the cluster dashboard and click "Connect".

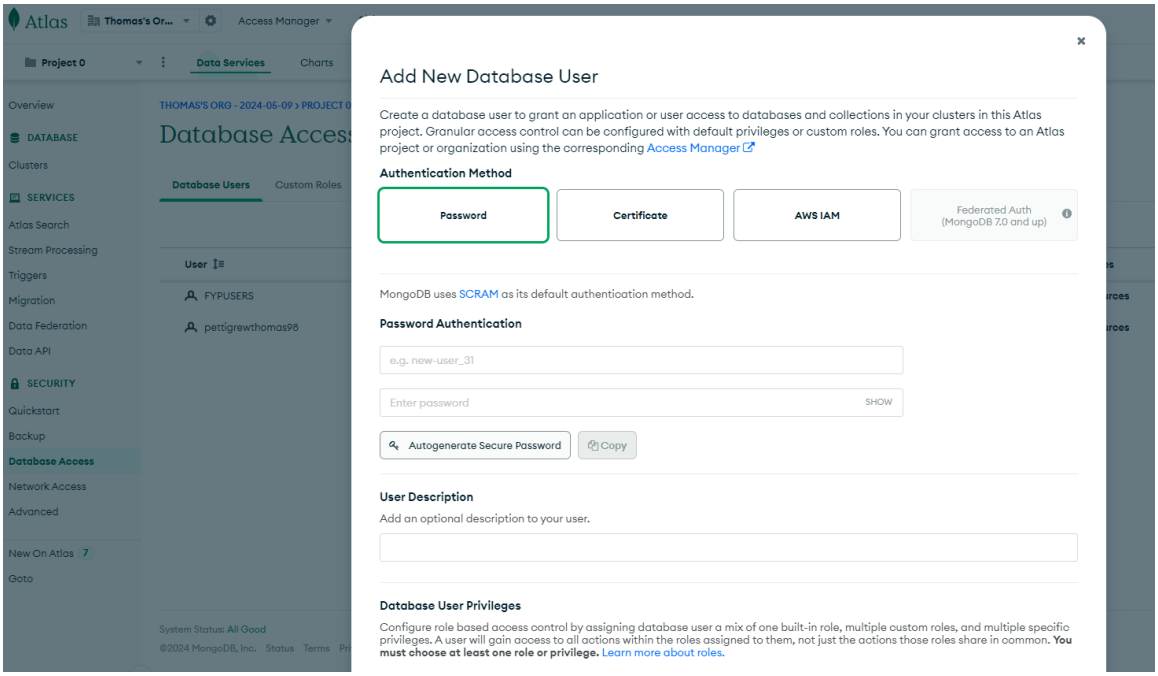


Set up a database user:

Choose "Create a Database User" and provide a username and password.

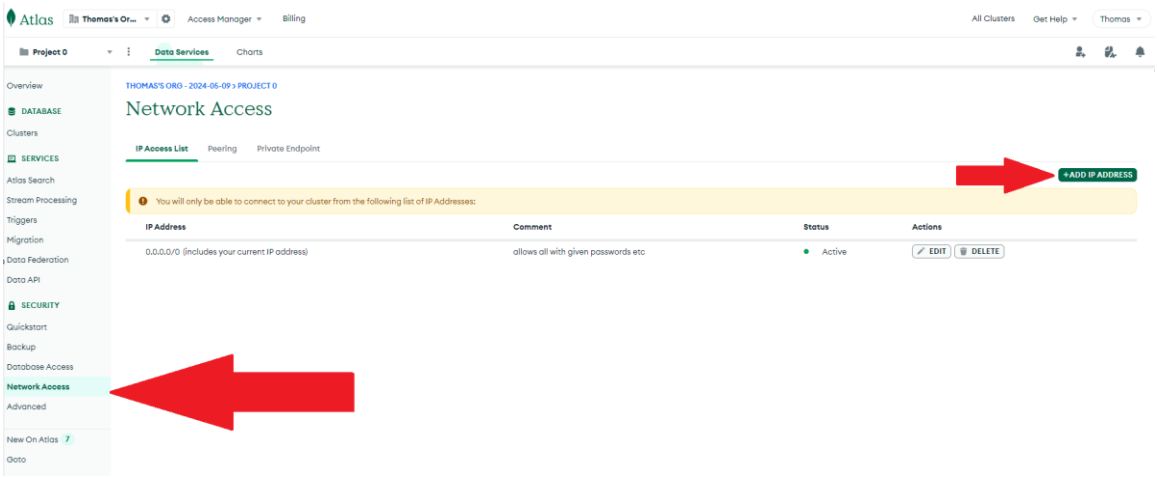


Note down the credentials as they will be needed for connecting the ESP32.



Add your IP address to the IP Whitelist:

Select "Add IP Address" and add your current IP or use 0.0.0.0/0 for unrestricted access (not recommended for production).



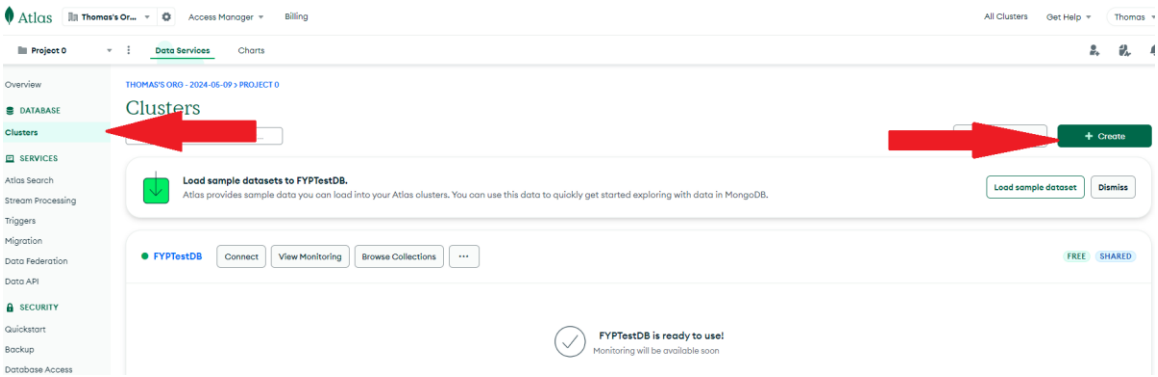
Get the connection string:

Choose "Connect Your Application" and copy the MongoDB URI connection string (e.g., mongodb+srv://<username>:<password>@cluster0.mongodb.net/test).

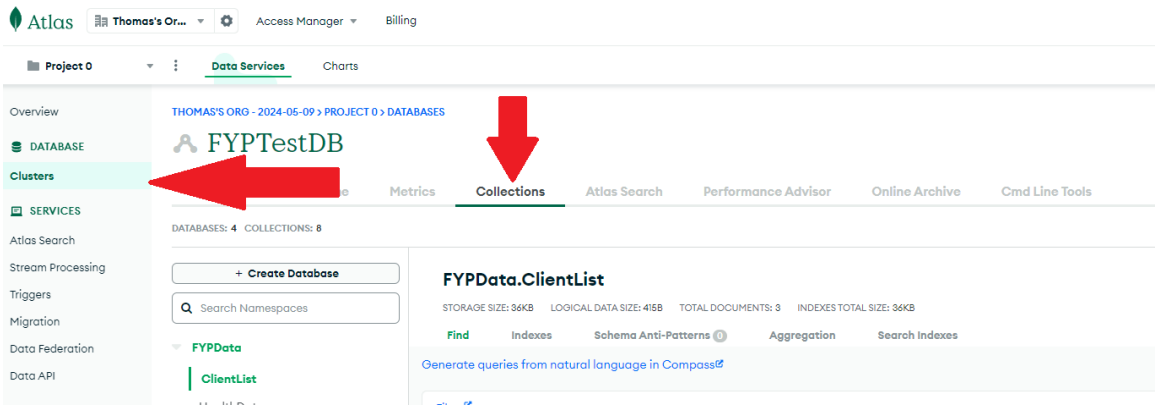
For more information on setting up MongoDB clusters, refer to MongoDB Atlas Documentation.

Create the Database and Collections:

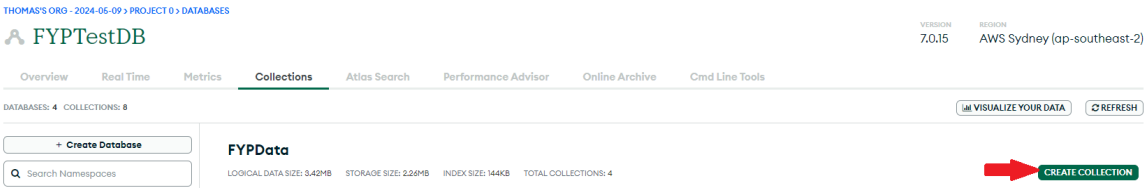
Navigate to your MongoDB Atlas dashboard and click on your cluster.



Open the Collections tab and create a new database by clicking "Add My Own Data".



Provide a database name (e.g., HealthMonitoring) and a collection name (e.g., TuningInformation).



Repeat the process to create additional collections as needed (e.g., HealthData, AutoTuneParameters).

Set Up the MongoDB Database Connection in the Code

Use the MongoDB connection string obtained earlier and insert it into the project's configuration file or source code (e.g., config.h this will be included in the folder with the .ino File).

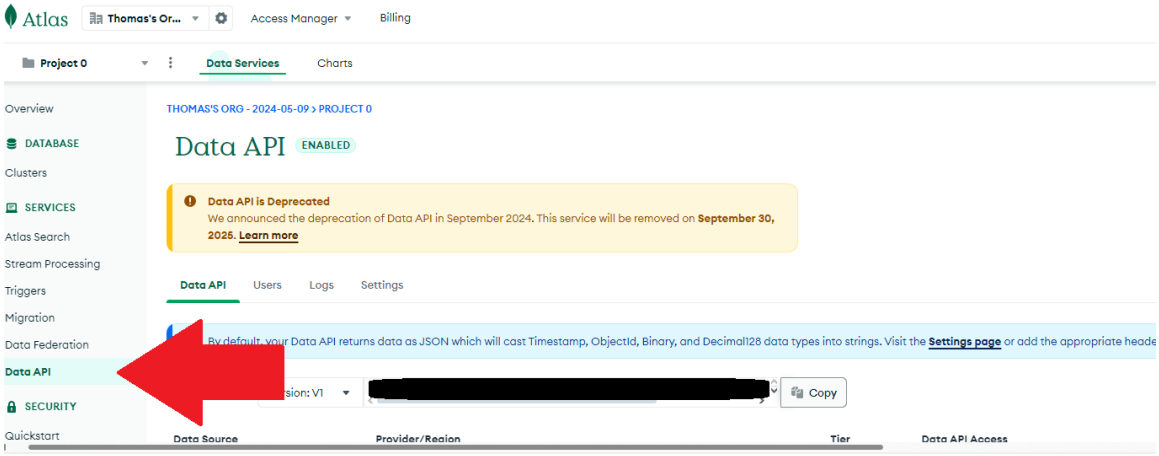
Define the database and collection names as constants to make future changes easier.

For more information on creating databases and collections, refer to [MongoDB Collections and Databases](#).

API Configuration

Create an API key for secure access to MongoDB:

In MongoDB Atlas, select Data API.



Click on "Create API Key" and give it a descriptive name.

Select appropriate permissions, such as "Read and Write Data".
Save the generated API key securely, as it will be needed for setting up connectivity between the ESP32 and MongoDB.

Configure Endpoints
Use the MongoDB connection string as your API endpoint.
Define REST API endpoints within your project code for sending and retrieving data. Here's an example configuration for a POST request to insert data into MongoDB (also found in the config.h):

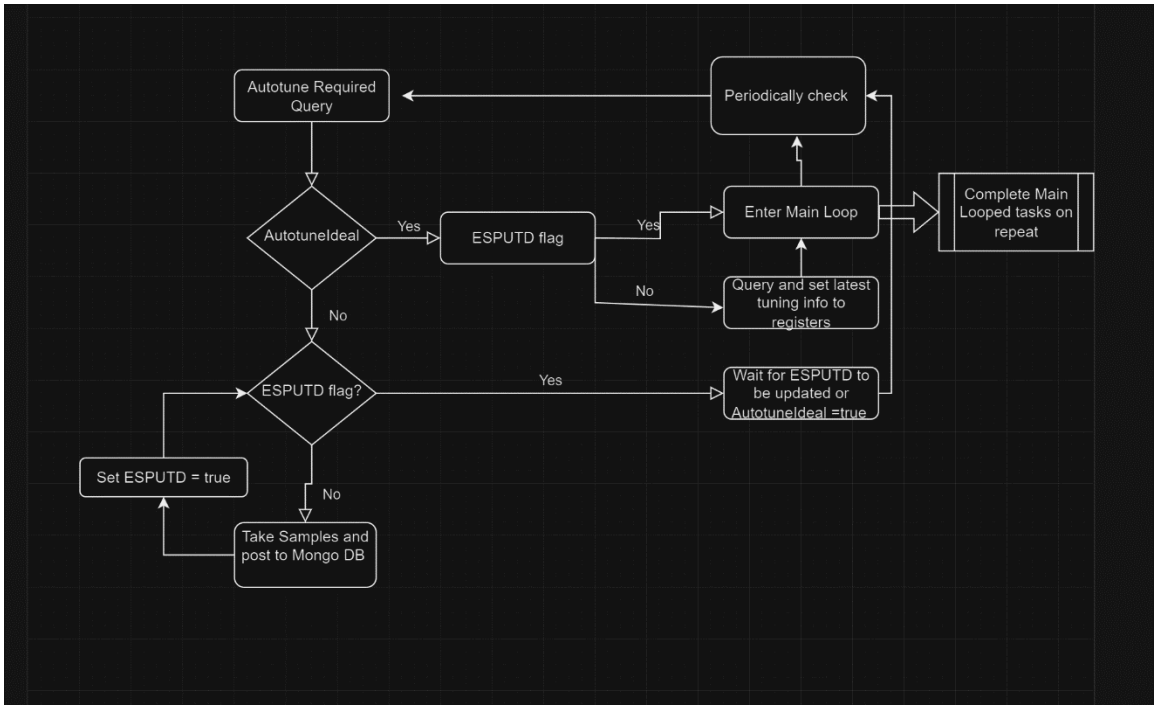
config.h

File Edit View

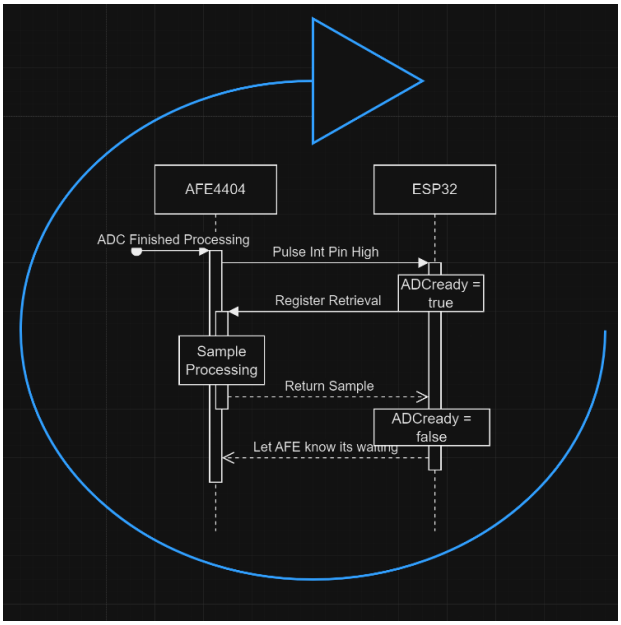
```
// config.h

// Wi-Fi settings
const char* ssid = "[redacted]";
const char* password = "[redacted]";

// MongoDB settings
const char* serverName = "[redacted]";
const char* apiKey = "[redacted]";
const char* deviceID =  ESP32Device2 ";

// Time Server URL
const char* timeServer = "https://www.timeapi.io/api/timezone/zone?timeZone=Australia/Melbourne";
```

Data Flow and Storage

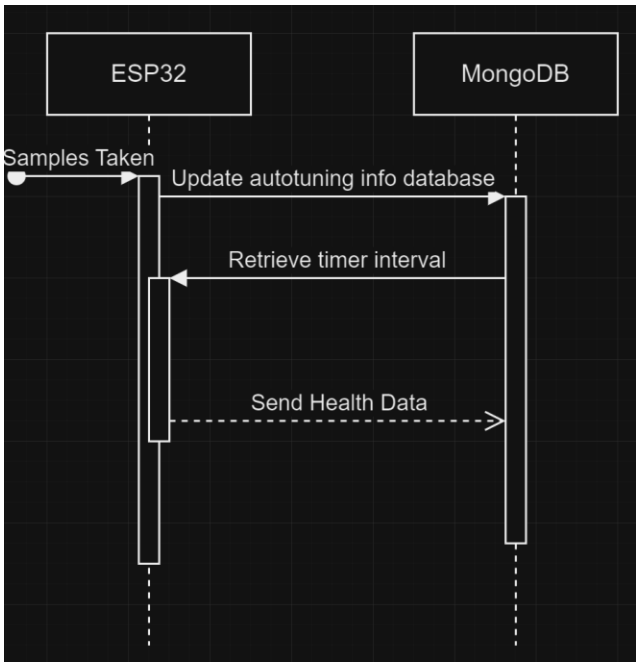
Autotune information flow:



AFE4404 and ESP32 interrupt data flow:



ESP to Mongo Dataflow:



Script Breakdown

DCDLFirmwareFinalRev.ino | config.h

```
#include <Wire.h>
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include "HeartRate5.h"
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <MCP79412RTC.h>
#include <Adafruit_NeoPixel.h>
#include <TimeLib.h>
#include "config.h"
#define PIN  3     // NeoPixel data pin
#define NUMPIXELS 4 // Number of NeoPixels
#define DELAYVAL 20 // Delay for fade timing
#define MPU_INT_PIN 2 // GPIO pin connected to the MPU6050 INT pin
Adafruit_NeoPixel pixels(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);
#define MOTION_DURATION 1  // Motion detection duration (in ms)
#define CHARGE_PIN 10 // Pin for detecting charging status
#define BATTERY_PIN 5 // Pin for battery level (IO5)
#define readingtoggle 11
#define  RESETZ_PIN 22
// Time Server URL
const float referenceVoltage = 3.3;  // Reference voltage for the ADC (max voltage)
const int adcMaxValue = 4095;  // Max ADC value (12-bit resolution)
const float voltageDividerFactor = 1.0;  // Adjust if using a resistor divider
// MongoDB Flags
bool autoTuneIdeal = false;
bool espUTD = false;
int motionThreshold = 20;  // Default threshold (in mg)
// Tuning Parameters
int led1Current = 20;  // Green LED current
int led3Current = 20;  // IR LED current        // Gain setting
float current_gain = 1;  // Example value, adjust as needed
float current_tia_cf = 2;    // Example value, adjust as needed
bool readRed = false;
Adafruit_MPU6050 mpu;
// RTC instance to track timestamps
MCP79412RTC RTC;
#define EEPROM_DEVICE_ID_ADDR 0x57
// HeartRate5 sensor instance
HeartRate5 heartRate5(0x58);  // I2C address of the sensor

// Function Declarations
void connectToWiFi();
bool checkAutoTuneRequired();
bool checkESPUTDFlag();
String getCurrentTimestamp();
void setRTCToSystemTime();
void createNewTuningEntry();
void getReadScheduleFromDB();
void performReadCycle();
void sendHealthDataToMongoDB(String dateTime, String greenData, String irData, String redData, String mpuData);
void updateTuningParameters();
void updateFlagsInMongoDB(bool newESPUTD, bool newAutoTuneIdeal, String dateTime);
void printCurrentSettings();
void goToSleepUntilNextScheduledRead();
volatile bool adcReadyFlag = false; // Flag to indicate that ADC is ready
struct IntervalSchedule {
    int intervalMinutes;  // Store the interval in minutes (e.g., 10 for 10 minutes)
};

IntervalSchedule schedule;


void IRAM_ATTR onAdcReady() {
    adcReadyFlag = true;
}
```

Main variable and library declarations. The ADC interrupt.

```
//Normal Setup Protocols
void setup() {

    Serial.begin(115200);
    connectToWiFi();
    pixels.begin(); // Initialize NeoPixel strip
    pixels.clear(); // Clear any previous state
    pinMode(CHARGE_PIN, INPUT);
     pinMode(readingtoggle, OUTPUT);
    // Setup MPU6050
    if (!mpu.begin(0x69)) {
        Serial.println("Failed to initialize MPU6050!");
        while (1);
    }
    configureMPU6050ForMotion(motionThreshold);  // Set threshold and enable interru
    enableMPUInterruptWakeup();                  // Enable wake-up via MPU interrupt



    // Set the RTC to system time using an external time server
    setRTCToSystemTime();


  // Initialize the HeartRate5 sensor
    heartRate5.begin();
    heartRate5.init();
    heartRate5.configureSamplingRate();

    // Configure interrupts for the HeartRate5 sensor
    heartRate5.enableProgrammableTimingInterrupt(0, 31999);
    heartRate5.enableAdcReadyInterrupt(15); // Set up the ADC_RDY interrupt on pin 1
    pinMode(4, INPUT);
    // Attach interrupt to the ADC_RDY pin
    attachInterrupt(digitalPinToInterrupt(4), onAdcReady, RISING);
    // Check if auto-tuning is required by querying MongoDB
    Serial.println("Checking if auto-tuning is required...");
    bool autoTuneCheck = checkAutoTuneRequired();
    Serial.print("AutoTuneIdeal: ");
    Serial.println(autoTuneIdeal);
    Serial.print("ESPUTD: ");
    Serial.println(espUTD);

    if (!autoTuneCheck) {
        Serial.println("Auto-tuning required. Starting tuning cycle...");
        runTuningCycle();
    } else {
        Serial.println("Auto-tuning not required. Proceeding to main program...");
    }
    // Retrieve the read schedule from MongoDB
    getReadScheduleFromDB();
    fetchAndApplyTuningParameters();
}
```

Main Setup Routine. Will run first on boot every time. Handles initial setup calls and sensor initializations.

```
void connectToWiFi() {
    Serial.print("Connecting to Wi-Fi...");
    // Disconnect any existing connection
    WiFi.disconnect(true);  // Disconnect any previous connection

    // Start the Wi-Fi connection process
    WiFi.begin(ssid, password);

    // Wait for the connection to be established, with a timeout mechanism
    unsigned long startAttemptTime = millis();
    const unsigned long connectionTimeout = 15000;  // 15 seconds timeout
    int i=0;

    while (WiFi.status() != WL_CONNECTED && (millis() - startAttemptTime) < connectionTimeout) {
        delay(500);
        Serial.print(".");
          pixels.setPixelColor(i, pixels.Color(10, 0, 0)); // Full green
                pixels.show();
                if(i < 4){
                    i++;
                }else {
                    pixels.clear();
                    i=0;
                }

    }


    if (WiFi.status() == WL_CONNECTED) {
        pixels.clear();
        Serial.println("Connected to Wi-Fi.");
         pixels.setPixelColor(0, pixels.Color(0, 10, 0));
           pixels.setPixelColor(1, pixels.Color(0, 10, 0));
             pixels.setPixelColor(2, pixels.Color(0, 10, 0));// Full green
                pixels.setPixelColor(3, pixels.Color(0, 10, 0));
                pixels.show();
                  delay(10);
                pixels.clear();

    } else {
        pixels.clear();
             pixels.setPixelColor(0, pixels.Color(0, 0, 10));
           pixels.setPixelColor(1, pixels.Color(0, 0 , 10));
           delay(200);
        pixels.show();

        Serial.println("Failed to connect to Wi-Fi within timeout period.");
        connectToWiFi();
    }

        pixels.clear(); // Clear any previous state
}
```

Function for initializing and connecting to the wifi network.

```
// Main Read Loop
void loop() {
  pixels.clear();
   // Check if the device is charging
    int batteryPercentage = 0;
     float batteryVoltage = 0;
   while (isCharging()) {
      // Read the battery percentage while charging
       batteryVoltage = readBatteryVoltage();
      int batteryPercentage = calculateBatteryPercentage(batteryVoltage);

      // Display battery percentage with charging status
      displayBatteryPercentage(batteryPercentage, true);

      // Log charging status
      Serial.println("Charging... Waiting for charging to complete.");

      delay(4000); // Check every 5 seconds while charging
      pixels.clear();

      delay(1000); // Check every 5 seconds while charging
   }

   if (autoTuneIdeal) {
      Serial.println("AutoTune is ideal and scheduled time matched. Performing read cycle.");
      performReadCycle();
  // After performing the read cycle, update the battery percentage in TuningInformation

      updateBatteryPercentageInTuningInfo();
      // After performing the read cycle, update the schedule and AutoTune status from MongoDB
      getReadScheduleFromDB();
      autoTuneIdeal = checkAutoTuneRequired();
       heartRate5.setLEDCurrent(0, 0, 0);

        // Display the battery percentage on NeoPixels
       displayBatteryPercentage(batteryPercentage,false);

   // Display for 20 seconds
   delay(20000);
  pixels.clear();
   pixels.show();

       goToSleepUntilNextScheduledRead();
   } else {
      Serial.println("AutoTune not ideal or not the scheduled time. Going back to sleep.");
        // Display the battery percentage on NeoPixels
   displayBatteryPercentage(batteryPercentage,false);

   // Display for 20 seconds
   delay(20000);
  pixels.clear();
   pixels.show();

    goToSleepUntilNextScheduledRead();  // Enter sleep mode to conserve power
   }
}
```

Main program loop. Runs on Repeat with sleep cycles between reads.

```
// Function to perform a read cycle (taking sensor readings and sending data to MongoDB)
void performReadCycle() {
    int totalSamples = 1875;  // Total samples for 125Hz over 15 seconds
     int sampleIndex = 0;
    uint16_t* greenSamples = (uint16_t*)malloc(totalSamples * sizeof(uint16_t));
    uint16_t* irSamples = (uint16_t*)malloc(totalSamples * sizeof(uint16_t));
    uint16_t* redSamples = (uint16_t*)malloc(totalSamples * sizeof(uint16_t));

    if (!greenSamples || !irSamples || !redSamples) {
        Serial.println("Failed to allocate memory for samples.");
        free(greenSamples);
        free(irSamples);
        free(redSamples);
        return;
    }
    String dateTime = getCurrentTimestamp();

     Serial.println("Collecting 1875 samples over 15 seconds for Green and IR LEDs...");

    heartRate5.setLEDCurrent(led1Current, 0, led3Current);
     sampleIndex = 0;

    while (sampleIndex < totalSamples) {
        if (adcReadyFlag) {
            adcReadyFlag = false;  // Reset the flag
              Serial.println("adc flag : " + adcReadyFlag);
            uint32_t greenValue, irValue, redValue;
            if (heartRate5.readSensor(greenValue, irValue, redValue)) {
                greenSamples[sampleIndex] = greenValue;
                irSamples[sampleIndex] = irValue;
                sampleIndex++;
            }
        }
    }

    heartRate5.setLEDCurrent(0, 15, 0);
    sampleIndex = 0;

    // Collect 1875 samples over 15 seconds for Red LED
    Serial.println("Collecting 1875 samples over 15 seconds for Red LED...");
    while (sampleIndex < totalSamples) {
        if (adcReadyFlag) {
            adcReadyFlag = false;  // Reset the flag

            uint32_t greenValue, irValue, redValue;
            if (heartRate5.readSensor(greenValue, irValue, redValue)) {
                redSamples[sampleIndex] = redValue;
                sampleIndex++;
            }
        }
    }
    // Convert sample arrays to comma-separated strings and send to MongoDB
    String greenData = "", irData = "", redData = "";
    processSampleDataAndFree(greenSamples, totalSamples, greenData);
    processSampleDataAndFree(irSamples, totalSamples, irData);
    processSampleDataAndFree(redSamples, totalSamples, redData);


    sendHealthDataToMongoDB(dateTime, greenData, irData, redData, readMPUData());
}
void processSampleDataAndFree(uint16_t* samples, int totalSamples, String& outputData) {
    if (!samples) {
        Serial.println("Error: No samples provided.");
        return; // Exit if sample pointer is null
    }

    // Convert samples to comma-separated string
    outputData.reserve(totalSamples * 6);  // Pre-allocate memory to avoid reallocations
    for (int i = 0; i < totalSamples; i++) {
        outputData += String(samples[i]);
        if (i < totalSamples - 1) outputData += ",";
    }

    // Free the allocated memory for samples
    free(samples);
```

Read Cycle, This gets the pulse ox samples

```
void processSampleDataAndFree(uint16_t* samples, int totalSamples, String& outputData) {
    if (!samples) {
        Serial.println("Error: No samples provided.");
        return; // Exit if sample pointer is null
    }

    // Convert samples to comma-separated string
    outputData.reserve(totalSamples * 6);  // Pre-allocate memory to avoid reallocations
    for (int i = 0; i < totalSamples; i++) {
        outputData += String(samples[i]);
        if (i < totalSamples - 1) outputData += ",";
    }

    // Free the allocated memory for samples
    free(samples);
}
void sendHealthDataToMongoDB(String dateTime, String greenData, String irData, String redData, String mpuData) {

    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "insertOne";
        DynamicJsonDocument* doc = new DynamicJsonDocument(50000);  // Allocate larger memory on heap

        if (!doc) {
            Serial.println("Failed to allocate memory for doc");
            return;
        }

        // Populate the document with data
        (*doc)["dataSource"] = "FYPTestDB";
        (*doc)["database"] = "FYPData";
        (*doc)["collection"] = "HealthData";
        (*doc)["document"]["DeviceID"] = deviceID;
        (*doc)["document"]["DateTime"] = dateTime;
        (*doc)["document"]["GreenLED"] = greenData;
        (*doc)["document"]["IRLED"] = irData;
        (*doc)["document"]["RedLED"] = redData;
        (*doc)["document"]["MPU6050"] = mpuData;
        Serial.print("Free heap memory: ");
        Serial.println(ESP.getFreeHeap());

        // Serialize the document to JSON
        String jsonPayload;
        serializeJson(*doc, jsonPayload);

        // Print the size of the JSON payload for debugging
        Serial.print("JSON Payload Size: ");
        Serial.println(jsonPayload.length());

        // Begin the HTTP request
        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        // Send the POST request
        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200 || httpResponseCode == 201) {
            Serial.println("Health data successfully sent to MongoDB.");
        } else {
            Serial.print("Error sending health data to MongoDB: ");
    Serial.println(httpResponseCode);
    String serverResponse = http.getString();
    Serial.println("Server response: " + serverResponse);
        }

        // Clean up
        delete doc;
        http.end();
    }
}
```

Processes the samples then sends them to the Mongo Database.

```
// Handle motion event (e.g., a cough)
bool initializeMPU() {
    if (!mpu.begin()) {
        Serial.println("Failed to find MPU6050 sensor!");
        return false;
    }
    // Configure the MPU6050
    mpu.setAccelerometerRange(MPU6050_RANGE_2_G);
    mpu.setGyroRange(MPU6050_RANGE_250_DEG);
    mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);
    Serial.println("MPU6050 initialized successfully.");
    return true;
}
void updateThresholdFromMongoDB() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "/findOne";

        // Build the JSON request to find the threshold setting
            DynamicJsonDocument* doc = new DynamicJsonDocument(16384);  // Allocate on heap
        if (!doc) {
            Serial.println("Failed to allocate memory for doc");
            return;
        }
        (*doc)["dataSource"] = "FYPTestDB";
        (*doc)["database"] = "FYPData";
        (*doc)["collection"] = "TuningInformation";
        (*doc)["filter"]["DeviceID"] = deviceID;

        String jsonPayload;
        serializeJson(*doc, jsonPayload);

        // Send HTTP POST request
        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        int httpResponseCode = http.POST(jsonPayload);

        if (httpResponseCode == 200) {
            String response = http.getString();
            DynamicJsonDocument responseDoc(4096);
            DeserializationError error = deserializeJson(responseDoc, response);

            if (!error && responseDoc["document"].containsKey("MotionThreshold")) {
                // Update the motion threshold based on the MongoDB entry
                motionThreshold = responseDoc["document"]["MotionThreshold"].as<int>();
                Serial.print("Motion threshold updated to: ");
                Serial.println(motionThreshold);

                // Update the MPU6050 with the new threshold
                configureMPU6050ForMotion(motionThreshold);
            }
        }
        delete doc;
        http.end();
    }
}

void enableMPUInterruptWakeup() {
    esp_sleep_enable_ext1_wakeup(1ULL << MPU_INT_PIN, ESP_EXT1_WAKEUP_ANY_HIGH);  // Wake on rising edge (motion detected)
    Serial.println("MPU6050 motion interrupt enabled for wake-up.");
}
void configureMPU6050ForMotion(int threshold) {
    mpu.setAccelerometerRange(MPU6050_RANGE_2_G);  // Set accelerometer range
    mpu.setMotionDetectionThreshold(threshold);    // Set motion threshold
    mpu.setMotionDetectionDuration(1);             // Motion detection duration
    mpu.setMotionInterrupt(true);                  // Enable motion interrupt
    Serial.println("MPU6050 configured for motion detection.");
}
// Function to read MPU6050 data and format it as a string
String readMPUData() {
    sensors_event_t a, g, temp;
    mpu.getEvent(&a, &g, &temp);

    // Format the MPU data into a comma-separated string
    String mpuData = String(a.acceleration.x, 2) + "," +
                     String(a.acceleration.y, 2) + "," +
                     String(a.acceleration.z, 2) + "," +
                     String(g.gyro.x, 2) + "," +
                     String(g.gyro.y, 2) + "," +
                     String(g.gyro.z, 2);

    Serial.print("MPU Data: ");
    Serial.println(mpuData);  // Print the formatted data for debugging
    return mpuData;
}
```

All major MPU (accelerometer functions)

```
//Sleep mode Management
// Function to go to sleep until the next scheduled read
void goToSleepUntilNextScheduledRead() {
    Serial.println("Entering deep sleep mode based on the interval...");
    uint64_t sleepDuration = (uint64_t)schedule.intervalMinutes * 60 * 1000000;  // Convert to microseconds
    esp_sleep_enable_timer_wakeup(sleepDuration);
    Serial.print("Sleeping for ");
    Serial.print(schedule.intervalMinutes);
    Serial.println(" minutes.");
    esp_deep_sleep_start();
}
// Function to get the current timestamp from the RTC
String getCurrentTimestamp() {
    time_t now = RTC.get();
    return String(year(now)) + "-" + String(month(now)) + "-" + String(day(now)) + " " +
           String(hour(now)) + ":" + String(minute(now)) + ":" + String(second(now));
}









// Function to set RTC to system time using an external time server
void setRTCToSystemTime() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        http.begin(timeServer);  // Connect to the time server
        int httpResponseCode = http.GET();

        if (httpResponseCode == 200) {
            String payload = http.getString();
            DynamicJsonDocument doc(16384);
            deserializeJson(doc, payload);

            String dateTime = doc["utc_datetime"];
            int year = dateTime.substring(0, 4).toInt();
            int month = dateTime.substring(5, 7).toInt();
            int day = dateTime.substring(8, 10).toInt();
            int hour = dateTime.substring(11, 13).toInt();
            int minute = dateTime.substring(14, 16).toInt();
            int second = dateTime.substring(17, 19).toInt();

            // Use TimeLib to make a time_t from these components
            tmElements_t tm;
            tm.Year = year - 2024;  // Offset to epoch year
            tm.Month = month;
            tm.Day = day;
            tm.Hour = hour;
            tm.Minute = minute;
            tm.Second = second;
            time_t t = makeTime(tm);

            // Set the RTC
            RTC.set(t);
            Serial.println("RTC set to system time: " + getCurrentTimestamp());
        } else {
            Serial.print("Error getting time from server: ");
            Serial.println(httpResponseCode);
        }
        http.end();
    } else {
        Serial.println("Wi-Fi not connected. Unable to get system time.");
    }
}

void getReadScheduleFromDB() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "findOne";
        DynamicJsonDocument* doc = new DynamicJsonDocument(16384);  // Allocate on heap
        if (!doc) {
            Serial.println("Failed to allocate memory for doc");
            return;
        }
        (*doc)["dataSource"] = "FYPTestDB";
        (*doc)["database"] = "FYPData";
        (*doc)["collection"] = "TuningInformation";
        (*doc)["filter"]["DeviceID"] = deviceID;

        String jsonPayload;
        serializeJson((*doc), jsonPayload);

        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200) {
            String response = http.getString();
            DynamicJsonDocument responseDoc(16384);  // Adjust size based on expected response size
            if (deserializeJson(responseDoc, response) == DeserializationError::Ok) {
                if (responseDoc.containsKey("document") && !responseDoc["document"].isNull()) {
                    int interval = responseDoc["document"]["IntervalSchedule"]["IntervalMinutes"];
                    schedule.intervalMinutes = interval;
                    Serial.print("Read interval retrieved from MongoDB: ");
                    Serial.print(interval);
                    Serial.println(" minutes.");
                }
            }
            responseDoc.clear();  // Clear the memory for the response document
        }
        delete doc;
        http.end();
    }
}
```

All timing based functions including the sleep cycles.

```
bool isCharging() {
    return digitalRead(CHARGE_PIN) == LOW; // Adjust logic if your charging circuit differs
}
// Function to display battery percentage on NeoPixels
// Modify this function to show different colors when charging
void displayBatteryPercentage(int batteryPercentage, bool charging) {
    int ledCount = batteryPercentage / 25;  // Determine number of LEDs to light up fully
    int remainder = batteryPercentage % 25;  // Remaining percentage for partial LED

    for (int i = 0; i < NUMPIXELS; i++) {
        if (i < ledCount) {
            if (charging) {
                // Yellow color to indicate charging
                pixels.setPixelColor(i, pixels.Color(10, 10, 0));
            } else {
                // Green for fully lit LEDs when not charging
                pixels.setPixelColor(i, pixels.Color(0, 10, 0));
            }
        } else if (i == ledCount && remainder > 0) {
            int red = map(remainder, 0, 2, 0, 10);
            int green = map(remainder, 0, 2, 10, 0);
            pixels.setPixelColor(i, pixels.Color(red, green, 0)); // Partial LED color
        } else {
            pixels.setPixelColor(i, pixels.Color(0, 0, 0)); // Turn off unused LEDs
        }
    }
    pixels.show(); // Update LED colors
}
// Function to update battery percentage in the TuningInformation collection
void updateBatteryPercentageInTuningInfo() {
    float batteryVoltage = readBatteryVoltage();
    int batteryPercentage = calculateBatteryPercentage(batteryVoltage);

    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "updateOne";
       DynamicJsonDocument* doc = new DynamicJsonDocument(16384);  // Allocate on heap
        if (!doc) {
            Serial.println("Failed to allocate memory for doc");
            return;
        }
         (*doc)["dataSource"] = "FYPTestDB";
         (*doc)["database"] = "FYPData";
         (*doc)["collection"] = "TuningInformation";
         (*doc)["filter"]["DeviceID"] = deviceID;

        // Update only the BatteryPercentage in the database
         (*doc)["update"]["$set"]["BatteryPercentage"] = batteryPercentage;

        String jsonPayload;
        serializeJson(*doc, jsonPayload);

        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200 || httpResponseCode == 201) {
            Serial.println("MongoDB entry updated successfully with new BatteryPercentage.");
        } else {
            Serial.print("Error updating MongoDB entry with BatteryPercentage: ");
            Serial.println(httpResponseCode);
        }
        delete doc;
        http.end();
    }

}
// Function to read the battery level and calculate the percentage
float readBatteryVoltage() {
    int rawValue = analogRead(BATTERY_PIN);  // Use analogRead to read the value from IO2
    float voltage = (rawValue / float(adcMaxValue)) * referenceVoltage * voltageDividerFactor;
    return voltage;
}
// Function to calculate battery percentage based on voltage
int calculateBatteryPercentage(float voltage) {
    // Define the minimum and maximum voltage of your battery (adjust as per your battery specs)
    float minVoltage = 3.0;  // Minimum voltage for a drained battery (example value)
    float maxVoltage = 4.2;  // Maximum voltage for a fully charged battery (example value)

    // Calculate the percentage based on the min and max voltage
    int percentage = int(((voltage - minVoltage) / (maxVoltage - minVoltage)) * 100);

    // Constrain the percentage between 0 and 100
    if (percentage < 0) percentage = 0;
    if (percentage > 100) percentage = 100;

    return percentage;
}
```

Battery Management Functions.

All functions in relation to tuning of the sensor from the AI model communications through the database:

```
// AUTOTUNING SECTION

// Run the tuning cycle with better flag handling
void runTuningCycle() {
    // Fetch initial flag values before entering the loop
    autoTuneIdeal = checkAutoTuneRequired();
    espUTD = checkESPUTDFlag();
    fetchAndApplyTuningParameters();

    // Loop while AutoTuneIdeal is false and ESPUTD is false
    while (!autoTuneIdeal && !espUTD) {
        Serial.println("Starting tuning cycle...");

        // Perform the tuning process (e.g., collecting samples, adjusting parameters, etc.)
        Serial.println("Collecting 1875 samples over 15 seconds at 125Hz...");
        String dateTime = getCurrentTimestamp();
        heartRate5.setLEDCurrent(20, 0, 20);
        const int totalSamples = 1875;
        uint16_t* greenSamples = (uint16_t*)malloc(totalSamples * sizeof(uint16_t));
        uint16_t* irSamples = (uint16_t*)malloc(totalSamples * sizeof(uint16_t));

        if (!greenSamples || !irSamples) {
            Serial.println("Failed to allocate memory for samples.");
            if (greenSamples) free(greenSamples);
            if (irSamples) free(irSamples);
            return;
        }
digitalWrite(readingtoggle, LOW);
        int sampleIndex = 0;
 while (sampleIndex < totalSamples) {
        if (adcReadyFlag) {
            adcReadyFlag = false;  // Reset the flag
              Serial.println("adc flag : " + adcReadyFlag);
            uint32_t greenValue, irValue, redValue;
            if (heartRate5.readSensor(greenValue, irValue, redValue)) {
                greenSamples[sampleIndex] = greenValue;
                irSamples[sampleIndex] = irValue;
                sampleIndex++;
            }
        }
    }
      String greenData = "", irData = "";
        for (int i = 0; i < 1875; i++) {  // Update sample count to 1875 for Auto-Tuning
            greenData += String(greenSamples[i]) + (i < 1875 - 1 ? "," : "");
            irData += String(irSamples[i]) + (i < 1875 - 1 ? "," : "");
        }
       free(greenSamples);
       free(irSamples);
        // Send the collected readings to MongoDB with the timestamp
        sendTuningReadingsToMongoDB(greenData, irData, dateTime);



        // Set ESPUTD flag to true, indicating that new data is ready for AI processing
        updateFlagsInMongoDB(true, dateTime);
        espUTD = true;

        // Free allocated memory

        // Wait for ESPUTD to be set back to false by the AI
        Serial.println("Waiting for AI to process and set ESPUTD back to false...");
        while (espUTD) {
            delay(10000);  // Wait for 10 seconds before checking again
            espUTD = checkESPUTDFlag();
        }

        // Check the AutoTuneIdeal status again
        autoTuneIdeal = checkAutoTuneRequired();
        Serial.print("AutoTuneIdeal after cycle: ");
        Serial.println(autoTuneIdeal);
    }

    Serial.println("Exiting tuning cycle. Main loop will handle further operations.");
}
```

```
// Function to check the ESPUTD flag status
bool checkESPUTDFlag() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "findOne";
          DynamicJsonDocument* doc = new DynamicJsonDocument(65536);  // Allocate on heap
        if (!doc) {
            Serial.println("Failed to allocate memory for doc");

        }

        (*doc)["dataSource"] = "FYPTestDB";
        (*doc)["database"] = "FYPData";
        (*doc)["collection"] = "TuningInformation";
        (*doc)["filter"]["DeviceID"] = deviceID;

        String jsonPayload;
        serializeJson(*doc, jsonPayload);

        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200) {
            String response = http.getString();
            DynamicJsonDocument responseDoc(16384);  // Allocate document based on response size
            if (deserializeJson(responseDoc, response) == DeserializationError::Ok) {
                if (responseDoc.containsKey("document") && !responseDoc["document"].isNull()) {
                    if (responseDoc["document"]["ESPUTD"].is<bool>()) {
                        espUTD = responseDoc["document"]["ESPUTD"].as<bool>();
                    } else if (responseDoc["document"]["ESPUTD"].is<String>()) {
                        String espUTDStr = responseDoc["document"]["ESPUTD"].as<String>();
                        espUTD = (espUTDStr == "true");
                    }

                    // Clear the memory of the document
                    responseDoc.clear();
                      delete doc; // Clear the memory for the request document
                    http.end();
                    return espUTD;
                }
            }
        }
        if (doc) {
            delete doc;
        }
        http.end();
    }
    return false;
}



// Function to create a new MongoDB entry with default values and additional relevant data
void createNewTuningEntry() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "insertOne";
        DynamicJsonDocument* doc = new DynamicJsonDocument(16384);  // Allocate on heap

        if (!doc) {
            Serial.println("Failed to allocate memory for doc");
            return;
        }

        // Populate the JSON document with additional relevant fields
        (*doc)["dataSource"] = "FYPTestDB";
        (*doc)["database"] = "FYPData";
        (*doc)["collection"] = "TuningInformation";
        (*doc)["document"]["DeviceID"] = deviceID;
        (*doc)["document"]["AutoTuneIdeal"] = false;
        (*doc)["document"]["ESPUTD"] = false;

        // Tuning data
        (*doc)["document"]["TuningData"]["LED1_CURRENT"] = led1Current;
        (*doc)["document"]["TuningData"]["LED3_CURRENT"] = led3Current;
        (*doc)["document"]["TuningData"]["GAIN"] = current_gain;
        (*doc)["document"]["TuningData"]["TIA_CAPACITOR"] = current_tia_cf;

        // Battery data
        float batteryVoltage = readBatteryVoltage();
        int batteryPercentage = calculateBatteryPercentage(batteryVoltage);
        (*doc)["document"]["Battery"]["Percentage"] = batteryPercentage;

        // Interval schedule
        int defaultInterval = 10; // Default interval for scheduling (e.g., 10 minutes)
        (*doc)["document"]["IntervalSchedule"]["IntervalMinutes"] = defaultInterval;

        // Add current timestamp
        String dateTime = getCurrentTimestamp(); // Get the current timestamp from RTC
        (*doc)["document"]["DateTime"] = dateTime;

        // Add MPU6050 configuration data (example)
        (*doc)["document"]["MPU6050"]["MotionDuration"] = MOTION_DURATION;
        (*doc)["document"]["MPU6050"]["Threshold"] = motionThreshold;

        // Serialize and send the request to MongoDB
        String jsonPayload;
        serializeJson(*doc, jsonPayload);

        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200 || httpResponseCode == 201) {
            Serial.println("New tuning entry created in MongoDB with default parameters and additional data.");
        } else {
            Serial.print("Error creating new tuning entry: ");
            Serial.println(httpResponseCode);
        }

        // Clean up
        delete doc;
        http.end();
    } else {
        Serial.println("Wi-Fi not connected. Unable to create new tuning entry.");
    }
}
```

```
// Function to fetch tuning parameters from MongoDB and update the sensor
void fetchAndApplyTuningParameters() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "findOne";
        DynamicJsonDocument* doc = new DynamicJsonDocument(16384);  // Allocate on heap

        if (!doc) {
            Serial.println("Failed to allocate memory for doc");
            return;
        }

        // Build the JSON request to find the tuning data
        (*doc)["dataSource"] = "FYPTestDB";
        (*doc)["database"] = "FYPData";
        (*doc)["collection"] = "TuningInformation";
        (*doc)["filter"]["DeviceID"] = deviceID;

        String jsonPayload;
        serializeJson(*doc, jsonPayload);

        // Send HTTP POST request
        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200) {
            String response = http.getString();
            DynamicJsonDocument responseDoc(98304);
            DeserializationError error = deserializeJson(responseDoc, response);

            if (!error) {
                if (responseDoc.containsKey("document") && !responseDoc["document"].isNull()) {
                    // Retrieve tuning parameters from the response
                    led1Current = responseDoc["document"]["TuningData"]["LED1_CURRENT"].as<int>();
                    led3Current = responseDoc["document"]["TuningData"]["LED3_CURRENT"].as<int>();
                    current_gain = responseDoc["document"]["TuningData"]["GAIN"].as<int>();
                    current_tia_cf = responseDoc["document"]["TuningData"]["TIA_CF"].as<float>();

                    // Update the sensor parameters with the retrieved values
                    updateTuningParameters();
                } else {
                    Serial.println("Tuning data not found for this device.");
                }
            } else {
                Serial.print("Failed to deserialize JSON response: ");
                Serial.println(error.c_str());
            }
        } else {
            Serial.print("HTTP Error: ");
            Serial.println(httpResponseCode);
        }

        // Clean up
        delete doc;
        http.end();
    } else {
        Serial.println("Wi-Fi not connected. Unable to fetch tuning parameters.");
    }
}


// Update flags in MongoDB with a timestamp
void updateFlagsInMongoDB(bool newESPUTD, String dateTime) {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "updateOne";

        // Allocate on the stack
        DynamicJsonDocument doc(16384);

        // Populate the document
        doc["dataSource"] = "FYPTestDB";
        doc["database"] = "FYPData";
        doc["collection"] = "TuningInformation";
        doc["filter"]["DeviceID"] = deviceID;
        doc["update"]["$set"]["ESPUTD"] = newESPUTD;
        doc["update"]["$set"]["LastUpdated"] = dateTime;

        String jsonPayload;
        serializeJson(doc, jsonPayload);

        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        if (httpResponseCode == 200 || httpResponseCode == 201) {
            Serial.println("ESPUTD flag updated successfully in MongoDB with timestamp.");
        } else {
            Serial.print("Error updating ESPUTD flag in MongoDB: ");
            Serial.println(httpResponseCode);
        }

        http.end();
    }
}
```

```
// Send readings to MongoDB with a timestamp
void sendTuningReadingsToMongoDB(String greenSamples, String irSamples, String dateTime) {
 if (WiFi.status() == WL_CONNECTED) {
       HTTPClient http;
       String url = String(serverName) + "updateOne";
      DynamicJsonDocument* doc = new DynamicJsonDocument(65536);  // Allocate on heap
       if (!doc) {
           Serial.println("Failed to allocate memory for doc");
           return;
       }
         (*doc)["dataSource"] = "FYPTestDB";
         (*doc)["database"] = "FYPData";
         (*doc)["collection"] = "TuningInformation";
         (*doc)["filter"]["DeviceID"] = deviceID;


         (*doc)["update"]["$set"]["MostRecentReadings"]["GreenLED"] = greenSamples;
         (*doc)["update"]["$set"]["MostRecentReadings"]["IRLED"] = irSamples;
         (*doc)["update"]["$set"]["MostRecentReadings"]["DateTime"] = dateTime;

       String jsonPayload;
       serializeJson(*doc, jsonPayload);

       http.begin(url);
       http.addHeader("Content-Type", "application/json");
       http.addHeader("api-key", apiKey);

       int httpResponseCode = http.POST(jsonPayload);
       if (httpResponseCode == 200 || httpResponseCode == 201) {
           Serial.println("MongoDB entry updated successfully with new readings and timestamp.");
       } else {
           Serial.print("Error updating MongoDB entry with readings: ");
           Serial.println(httpResponseCode);
       }
        delete doc;
       http.end();
   }
}
void updateTuningParameters() {
    // Adjust the HeartRate5 sensor parameters based on current tuning values
    heartRate5.setLEDCurrent(led1Current, 0, led3Current);
    heartRate5.setTIASettings(current_gain, current_tia_cf);
    printCurrentSettings();
}

// Check MongoDB if auto-tuning is required, create a new entry if it doesn't exist
bool checkAutoTuneRequired() {
    if (WiFi.status() == WL_CONNECTED) {
        HTTPClient http;
        String url = String(serverName) + "findOne";
          DynamicJsonDocument* doc = new DynamicJsonDocument(65536);  // Allocate on heap
        if (!doc) {
            Serial.println("Failed to allocate memory for doc");

        }
          (*doc)["dataSource"] = "FYPTestDB";
          (*doc)["database"] = "FYPData";
          (*doc)["collection"] = "TuningInformation";
          (*doc)["filter"]["DeviceID"] = deviceID;

        String jsonPayload;
        serializeJson(*doc, jsonPayload);
        Serial.println("AutoTune Check - JSON Payload: " + jsonPayload);  // Print payload for debugging

        http.begin(url);
        http.addHeader("Content-Type", "application/json");
        http.addHeader("api-key", apiKey);

        int httpResponseCode = http.POST(jsonPayload);
        Serial.print("AutoTune Check - HTTP Response Code: ");
        Serial.println(httpResponseCode);  // Print HTTP response code for debugging

        if (httpResponseCode == 200) {
            String response = http.getString();
            Serial.println("AutoTune Check - HTTP Response: " + response);  // Print the server response for debugg

            // Deserialize the response to check if document exists
            DynamicJsonDocument responseDoc(65536);  // Allocate document based on response size
            DeserializationError error = deserializeJson(responseDoc, response);
            if (!error) {
                if (responseDoc.containsKey("document") && !responseDoc["document"].isNull()) {
                    // If document exists, set flags and return the auto-tune status
                    autoTuneIdeal = responseDoc["document"]["AutoTuneIdeal"].as<bool>();
                    espUTD = responseDoc["document"]["ESPUTD"].as<bool>();
                    http.end();
                    return autoTuneIdeal;
                } else {
                    // Document not found or null, create a new tuning entry
                    Serial.println("Document not found. Creating a new entry in MongoDB...");
                    createNewTuningEntry();  // Create a new document with default values
                    http.end();
                    return false;  // Return false, as we just created a new entry and auto-tune needs to be run
                }
            } else {
                Serial.print("AutoTune Check - Failed to deserialize JSON response: ");
                Serial.println(error.c_str());  // Print deserialization error message
            }
        } else {
            Serial.print("AutoTune Check - HTTP Error: ");
            Serial.println(httpResponseCode);  // Print HTTP response error code
        }

          delete doc;
        http.end();
    } else {
        Serial.println("AutoTune Check - Wi-Fi not connected.");
    }
    return false;  // Default to requiring tuning if any error occurs
}
```