

**DEPARTMENT OF B.E ELECTRONICS AND COMMUNICATION ENGINEERING**

**Embedded System & IOT Design**

**TITLE:-** Drone Telemetry Simulation System Using ESP32 and Web Dashboard

# Abstract

This project presents the design and development of a real-time Drone Telemetry Dashboard that integrates embedded systems, IoT, and web technologies for reliable monitoring of aerial platforms. The system employs an ESP32 microcontroller programmed with FreeRTOS for concurrent task execution, enabling efficient sensor data acquisition and network communication. Core sensing units include the MPU6050 accelerometer-gyroscope for orientation measurement, a DHT11 sensor for environmental monitoring, and a battery voltage sensing module for power status tracking. Acquired data is formatted into JSON packets and transmitted over Wi-Fi using WebSocket communication to a Node.js server. The server handles real-time data broadcasting to multiple clients, where a browser-based dashboard visualizes telemetry through dynamic charts and gauges. This architecture ensures low-latency, bidirectional communication with modular scalability for additional sensors or features. The project demonstrates a cost-effective and extensible solution for drone telemetry applications, offering a foundation for future enhancements such as GPS integration, cloud connectivity, and mobile platform support.

# Table of Contents

Contents

[Abstract 2](#_Toc206866668)

[Table of Contents 3](#_Toc206866669)

[1. Introduction 4](#_Toc206866670)

[1.1 Objectives 4](#_Toc206866675)

[2. System Design and Architecture 5](#_Toc206866676)

[2.1 Workflow 5](#_Toc206866677)

[5](#_Toc206866678)

[3. Hardware and Software Requirements 6](#_Toc206866679)

[3.1 Hardware 6](#_Toc206866680)

[3.2 Software 6](#_Toc206866681)

[4.1 Firmware (ESP32) 6](#_Toc206866682)

[4.2 Server (Node.js) 7](#_Toc206866683)

[4.3 Frontend Dashboard 7](#_Toc206866684)

[This frontend acts as the visualization layer, providing intuitive monitoring of drone telemetry for the user. 8](#_Toc206866685)

[5. Execution and Demo 8](#_Toc206866686)

[5. Results 9](#_Toc206866687)

[Appendix 10](#_Toc206866688)

[Appendix A: Full Firmware Code (firmware.ino) 10](#_Toc206866689)

[Appendix B: Server Code (server.js) 15](#_Toc206866690)

[Appendix C: Sample Frontend Code (from index.html or related JS) 18](#_Toc206866691)

[Appendix D: Project Repository 19](#_Toc206866692)

[References 19](#_Toc206866693)

# 1. Introduction

## Unmanned Aerial Vehicles (UAVs), popularly known as drones, are rapidly becoming indispensable across multiple domains such as defense, agriculture, logistics, surveillance, disaster management, and research. These aerial platforms require constant monitoring of their critical telemetry parameters to ensure safe, efficient, and reliable operation. Telemetry includes orientation (pitch, roll, yaw), linear acceleration, angular velocity, environmental conditions like temperature and humidity, and battery health. Accurate monitoring and visualization of this data not only improve drone performance but also enhance safety by providing early fault detection and predictive maintenance opportunities.

## Traditional telemetry systems often rely on expensive, proprietary hardware and communication modules, which limit accessibility and customization for academic projects or small-scale implementations. With the emergence of cost-effective microcontrollers like the ESP32 and advancements in IoT (Internet of Things) frameworks, it has become possible to design low-cost yet robust telemetry solutions that combine embedded programming, real-time communication, and web-based visualization.

## In this project, a Drone Telemetry Dashboard is developed using an ESP32 microcontroller, equipped with multiple onboard and external sensors, and supported by FreeRTOS for multitasking. The ESP32 collects data from the MPU6050 (accelerometer + gyroscope), DHT11 (temperature + humidity), and a custom battery voltage monitoring circuit. Using Wi-Fi as the communication medium, the ESP32 transmits this data in JSON format over a WebSocket protocol to a Node.js server. The server, in turn, broadcasts the telemetry data to connected clients, where it is displayed on a real-time dashboard developed using modern web technologies (HTML, CSS, JavaScript, and visualization libraries). This setup ensures minimal latency, efficient data transfer, and a scalable platform for real-time monitoring.

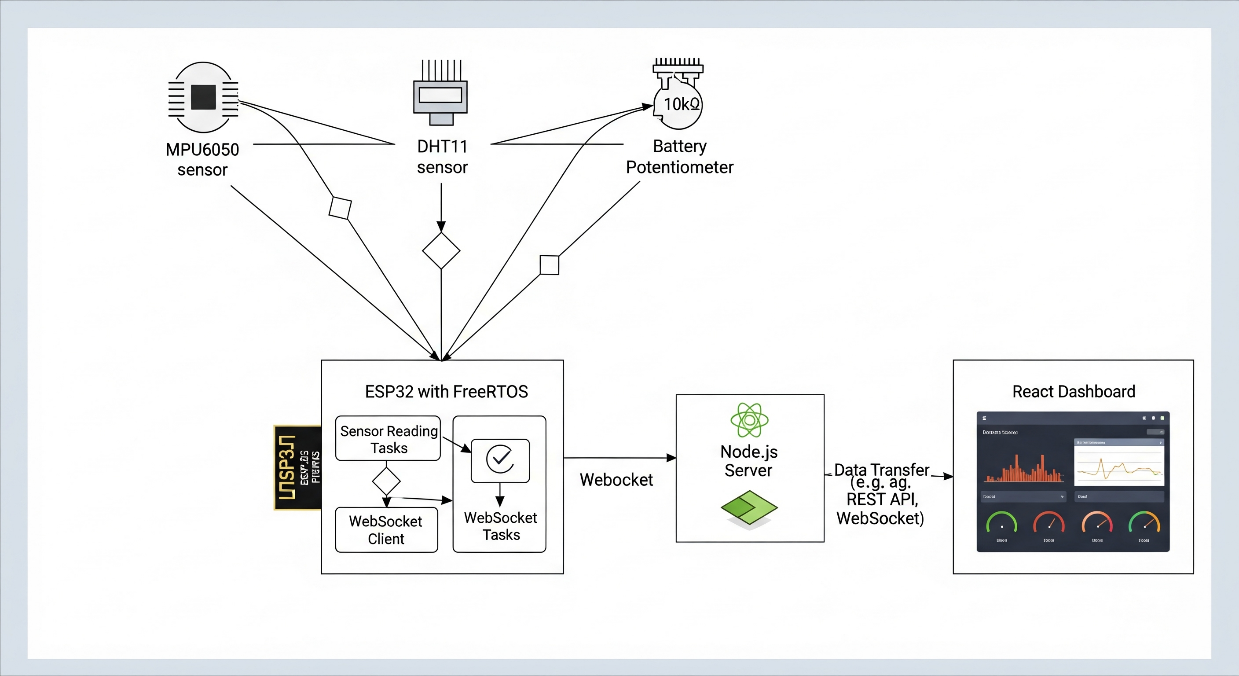
## The proposed solution is highly modular and can be extended to include additional sensors such as GPS, barometers, or air quality sensors. Furthermore, it can be adapted for cloud storage integration, mobile applications, and remote-control functionalities. By leveraging open-source tools and low-cost components, this project demonstrates an accessible and scalable telemetry solution suitable for both academic research and practical applications.

## 1.1 Objectives

- To design and implement a real-time telemetry acquisition system.  
- To establish wireless communication between drone sensors and a web server.  
- To visualize telemetry data on a user-friendly dashboard.  
- To explore expandability for additional sensors and functionalities.

# 2. System Design and Architecture

The system architecture consists of ESP32 firmware, Node.js server, and a frontend dashboard. The ESP32 gathers sensor data and sends it over Wi-Fi to the Node.js server using WebSockets. The server broadcasts the data to clients which visualize it on a dashboard.



## 2.1 Workflow

# 

# 3. Hardware and Software Requirements

## 3.1 Hardware

- ESP32 Development Board  
- MPU6050 (Accelerometer + Gyroscope)  
- DHT11 (Temperature & Humidity Sensor)  
- Battery Monitor Circuit  
- Supporting components (wires, breadboard, prototype frame)

## 3.2 Software

- Arduino IDE / PlatformIO  
- Node.js and npm  
- WebSocket Libraries  
- HTML, CSS, JavaScript (Frontend)  
- GitHub for version control.

**4. Implementation**

The implementation of the Drone Telemetry Dashboard is divided into three primary components: the firmware running on ESP32, the Node.js-based server, and the web-based frontend dashboard. Together, these modules ensure efficient real-time telemetry acquisition, processing, and visualization

## 4.1 Firmware (ESP32)

The firmware is developed in Arduino framework and deployed on the ESP32 microcontroller. To efficiently manage multiple tasks such as sensor data acquisition, packet formation, and network communication, FreeRTOS is utilized, allowing concurrent execution of tasks.

Key functionalities implemented include:

* Sensor Data Acquisition: Using libraries like Adafruit\_MPU6050 for the accelerometer/gyroscope and DHT.h for temperature-humidity sensing. Functions such as mpu.getEvent(&a, &g, &temp) are used to capture real-time orientation and environmental values.
* Battery Monitoring: Analog input pins of ESP32 are used with the analogRead()function to monitor voltage levels.
* Data Formatting: The ArduinoJson library structures sensor readings into JSO**M** packets, making them lightweight and easily transferable.
* Network Communication: The ESP32 connects to Wi-Fi using WiFi.begin(ssid, password) and establishes a WebSocket connection through webSocket.begin(server, port, path). Sensor data is periodically transmitted using webSocket.sendTXT(jsonString)
* This modular firmware design ensures that telemetry data is continuously collecte and transmitted with minimal latency.

## 4.2 Server (Node.js)

The server acts as a bridge between the ESP32 firmware and the frontend dashboard. It is implemented using Node.js with the ws (WebSocket) and express libraries.

Key functionalities include:

**WebSocket Communication:** The server listens for incoming JSON packets from the ESP32 using ws.on('message', callback). On receiving telemetry data, it immediately broadcasts the information to all connected clients.

**Static File Hosting:** Using express.static(), the server also delivers the dashboard frontend files (HTML, CSS, JS) to the client browser.

**Scalability:** The event-driven architecture of Node.js ensures the server can handle multiple clients simultaneously with minimal resource consumption.

Thus, the server ensures reliable, low-latency, and real-time transmission of telemetry data.

## 4.3 Frontend Dashboard

The frontend provides the user interface for real-time visualization of telemetry data. It is designed using HTML, CSS, and JavaScript, with integration of charting libraries such as Chart.js for graphical representation.

**Key features include:**

**WebSocket Client:** A WebSocket connection is established from the browser using new WebSocket("ws://server-ip:port"), which continuously listens for telemetry updates.

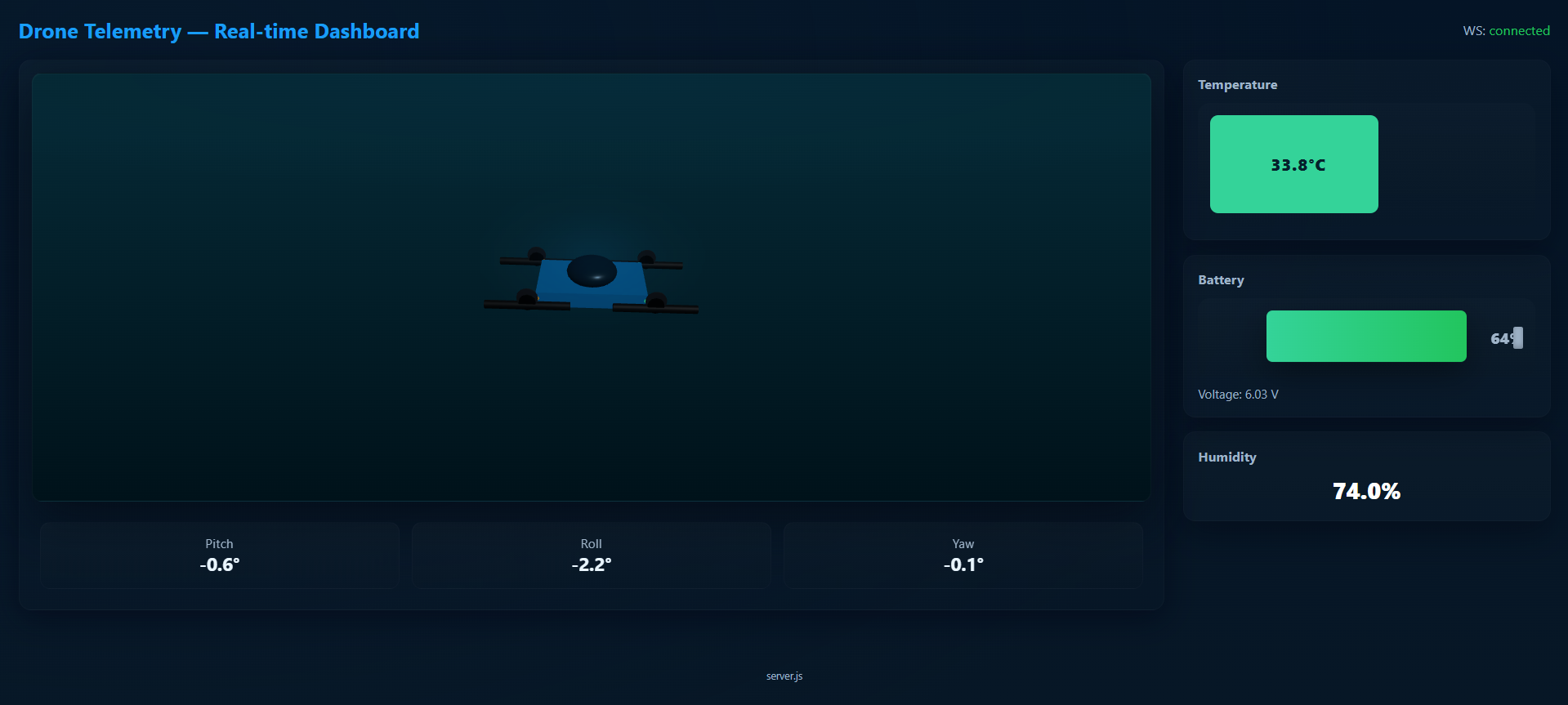
**Data Parsing and Visualization:** On receiving a JSON packet, JavaScript functions parse the values and update UI components such as gauges, line graphs, and text fields. Functions like chart.update() refresh data in real-time.

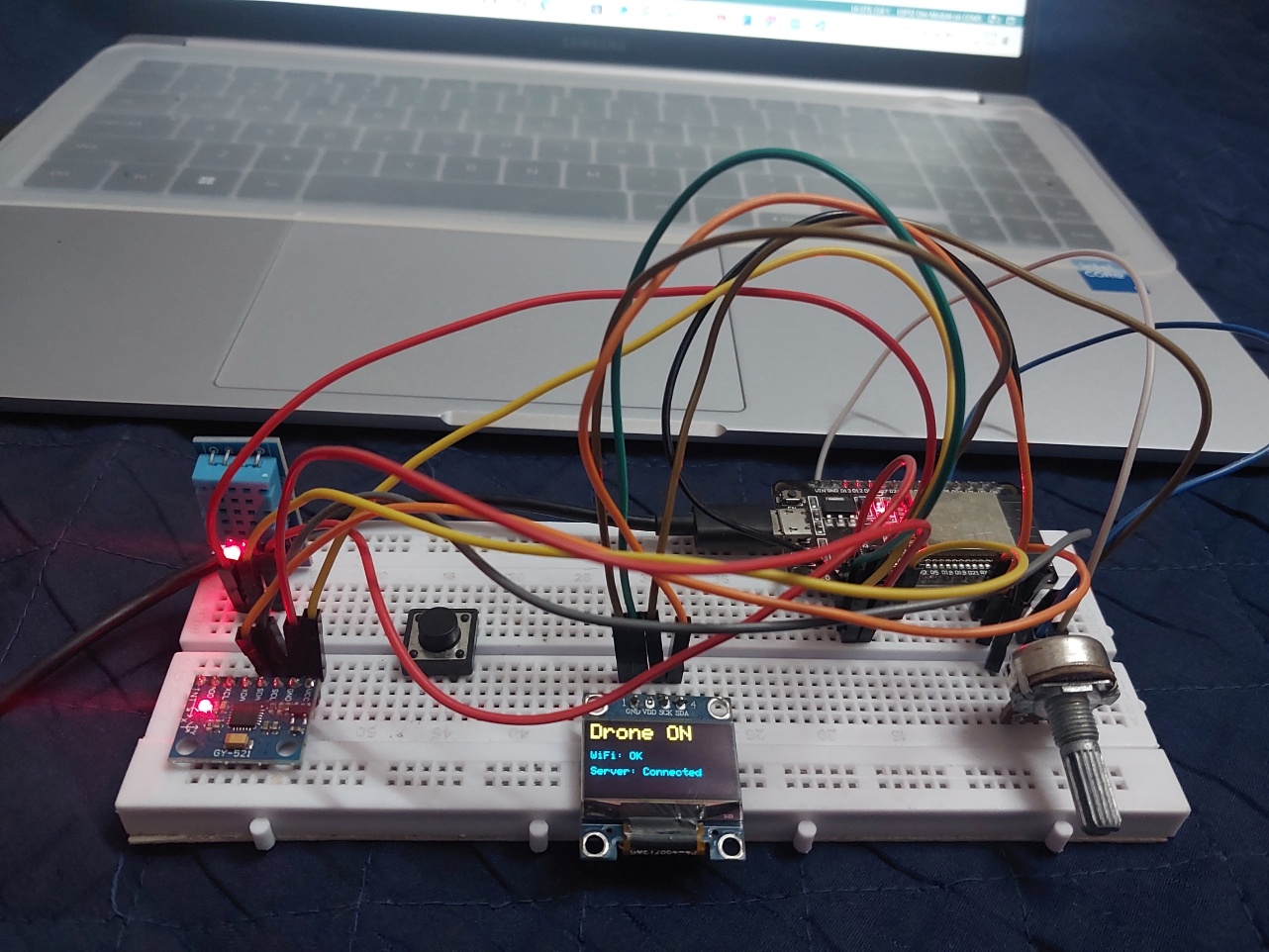
**User-Friendly Interface:** The dashboard is designed to present orientation, temperature, humidity, and battery levels in a clear and interactive format.

# This frontend acts as the visualization layer, providing intuitive monitoring of drone telemetry for the user.

# 5. Execution and Demo

The project was executed in the following steps:  
1. Upload firmware to ESP32.  
2. Run Node.js server (node server.js).  
3. Open dashboard in web browser.  
4. Visualize telemetry data in real-time.





# 5. Results

The Drone Telemetry Dashboard prototype was successfully designed and implemented. Real-time data acquisition, wireless communication, and visualization were demonstrated using the ESP32-based telemetry system.

Key results achieved:

* **Stable Wireless Connectivity**: The ESP32 maintained a reliable Wi-Fi link with the Node.js server, ensuring continuous data transfer with minimal packet loss.
* **Accurate Sensor Readings**: Orientation (pitch, roll, yaw), temperature, humidity, and battery voltage were measured consistently and formatted into JSON packets for structured communication.
* **Low Latency Data Transmission**: WebSocket-based communication allowed telemetry updates within milliseconds, making the system suitable for real-time drone monitoring.
* **Interactive Dashboard**: The web-based dashboard successfully displayed live sensor data through charts, gauges, and text fields. Updates occurred dynamically without page refresh, enhancing user experience.
* **Scalability Demonstration**: The architecture supports additional sensors and features, indicating potential for expansion into more advanced UAV telemetry systems.

These results validate the feasibility of building a **cost-effective, IoT-enabled telemetry dashboard** that can serve both academic and real-world UAV applications.

**6. Conclusion**

This project demonstrates the successful integration of **IoT, embedded systems, and web technologies** to build a functional drone telemetry dashboard. The system leverages the ESP32 for onboard sensing and communication, Node.js for efficient data handling, and a browser-based dashboard for real-time visualization.

The implementation highlights several important outcomes:

* Low-cost microcontrollers such as the ESP32 can be effectively used in UAV applications.
* WebSocket communication provides a robust method for real-time telemetry streaming.
* Modular design ensures expandability for future enhancements such as GPS tracking, altitude measurement, or camera feeds.

Overall, the project provides a **practical framework** for UAV telemetry monitoring that can be adapted to different drones and research environments.

**Future Scope**

* Integration of **GPS and altitude sensors** for advanced navigation telemetry.
* Addition of **flight log storage** in a database for post-flight analysis.
* Development of a **mobile app** for telemetry access on smartphones.
* Implementation of **security protocols** (e.g., encrypted communication) for safe UAV operation.

By combining affordability, flexibility, and real-time performance, this work lays the foundation for building more advanced UAV monitoring and control systems.

# Appendix

## Appendix A: Full Firmware Code (firmware.ino)

#include <WiFi.h>  
#include <WebSocketsClient.h>  
#include <Adafruit\_MPU6050.h>  
#include <Adafruit\_Sensor.h>  
#include <DHT.h>  
#include <ArduinoJson.h>  
#include <Wire.h>  
#include <Adafruit\_GFX.h>  
#include <Adafruit\_SSD1306.h>  
  
// ---------------- OLED Setup ----------------  
#define SCREEN\_WIDTH 128  
#define SCREEN\_HEIGHT 64  
#define OLED\_RESET -1  
#define SDA\_PIN 21  
#define SCL\_PIN 22  
  
Adafruit\_SSD1306 display(SCREEN\_WIDTH, SCREEN\_HEIGHT, &Wire, OLED\_RESET);  
volatile bool wsConnected = false; // shared flag for connection status  
  
// ---------------- Telemetry Setup ----------------  
#define SEND\_BINARY true  
#define PRINT\_EVERY\_N 10  
#define SENSOR\_DELAY\_MS 50  
#define SEND\_INTERVAL\_MS 50  
  
const char\* ssid = " ";  
const char\* password = " ";  
  
WebSocketsClient webSocket;  
const char\* websocket\_host = "192.168.1.9";  
const uint16\_t websocket\_port = 3000;  
const char\* websocket\_path = "/";  
  
Adafruit\_MPU6050 mpu;  
#define DHTPIN 4  
#define DHTTYPE DHT11  
DHT dht(DHTPIN, DHTTYPE);  
#define BATTERY\_PIN 34  
  
TaskHandle\_t sensorTaskHandle = NULL;  
TaskHandle\_t sendTaskHandle = NULL;  
TaskHandle\_t oledTaskHandle = NULL;  
SemaphoreHandle\_t dataMutex = NULL;  
  
struct TelemetryData {  
 float pitch;  
 float roll;  
 float yaw;  
 float temperature;  
 float humidity;  
 float batteryVoltage;  
} telemetry;  
  
volatile uint32\_t sendSeq = 0;  
volatile uint32\_t serialPrintCounter = 0;  
  
// ---------------- Helpers ----------------  
static inline void packUint32LE(uint8\_t \*buf, size\_t offset, uint32\_t v) {  
 buf[offset + 0] = (uint8\_t)(v & 0xFF);  
 buf[offset + 1] = (uint8\_t)((v >> 8) & 0xFF);  
 buf[offset + 2] = (uint8\_t)((v >> 16) & 0xFF);  
 buf[offset + 3] = (uint8\_t)((v >> 24) & 0xFF);  
}  
static inline void packFloatLE(uint8\_t \*buf, size\_t offset, float f) {  
 uint8\_t tmp[4];  
 memcpy(tmp, &f, 4);  
 buf[offset + 0] = tmp[0];  
 buf[offset + 1] = tmp[1];  
 buf[offset + 2] = tmp[2];  
 buf[offset + 3] = tmp[3];  
}  
  
// ---------------- WebSocket Events ----------------  
void webSocketEvent(WStype\_t type, uint8\_t \* payload, size\_t length) {  
 switch (type) {  
 case WStype\_CONNECTED:  
 Serial.println("[WS] Connected");  
 wsConnected = true;  
 break;  
 case WStype\_DISCONNECTED:  
 Serial.println("[WS] Disconnected");  
 wsConnected = false;  
 break;  
 default:  
 break;  
 }  
}  
  
// ---------------- Sensor Task ----------------  
void sensorTask(void \*parameter) {  
 sensors\_event\_t a, g, tempSensor;  
 (void) parameter;  
 for (;;) {  
 mpu.getEvent(&a, &g, &tempSensor);  
 float t = dht.readTemperature();  
 float h = dht.readHumidity();  
 if (isnan(t)) t = -127.0f;  
 if (isnan(h)) h = -127.0f;  
 int raw = analogRead(BATTERY\_PIN);  
 float voltage = raw \* (3.3f / 4095.0f) \* 2.0f;  
 float pitch = atan2(a.acceleration.y, a.acceleration.z) \* 180.0f / PI;  
 float roll = atan2(-a.acceleration.x, sqrt(a.acceleration.y \* a.acceleration.y + a.acceleration.z \* a.acceleration.z)) \* 180.0f / PI;  
 float yaw = g.gyro.z \* 57.29577951308232f;  
  
 if (xSemaphoreTake(dataMutex, pdMS\_TO\_TICKS(50)) == pdTRUE) {  
 telemetry.pitch = pitch;  
 telemetry.roll = roll;  
 telemetry.yaw = yaw;  
 telemetry.temperature = t;  
 telemetry.humidity = h;  
 telemetry.batteryVoltage = voltage;  
 xSemaphoreGive(dataMutex);  
 }  
 vTaskDelay(pdMS\_TO\_TICKS(SENSOR\_DELAY\_MS));  
 }  
}  
  
// ---------------- Send Task ----------------  
void sendTask(void \*parameter) {  
 (void) parameter;  
 uint8\_t binBuf[32];  
 StaticJsonDocument<256> jsonDoc;  
 char jsonBuf[256];  
  
 for (;;) {  
 TelemetryData local;  
 if (xSemaphoreTake(dataMutex, pdMS\_TO\_TICKS(50)) == pdTRUE) {  
 local = telemetry;  
 xSemaphoreGive(dataMutex);  
 } else {  
 vTaskDelay(pdMS\_TO\_TICKS(SEND\_INTERVAL\_MS));  
 continue;  
 }  
  
 uint32\_t seq = ++sendSeq;  
 uint32\_t ts\_ms = (uint32\_t) millis();  
  
 if (SEND\_BINARY) {  
 packUint32LE(binBuf, 0, seq);  
 packUint32LE(binBuf, 4, ts\_ms);  
 packFloatLE(binBuf, 8, local.pitch);  
 packFloatLE(binBuf, 12, local.roll);  
 packFloatLE(binBuf, 16, local.yaw);  
 packFloatLE(binBuf, 20, local.temperature);  
 packFloatLE(binBuf, 24, local.humidity);  
 packFloatLE(binBuf, 28, local.batteryVoltage);  
 webSocket.sendBIN(binBuf, sizeof(binBuf));  
 } else {  
 jsonDoc.clear();  
 jsonDoc["seq"] = seq;  
 jsonDoc["ts\_ms"] = ts\_ms;  
 jsonDoc["pitch"] = local.pitch;  
 jsonDoc["roll"] = local.roll;  
 jsonDoc["yaw"] = local.yaw;  
 jsonDoc["temperature"] = local.temperature;  
 jsonDoc["humidity"] = local.humidity;  
 jsonDoc["battery"] = local.batteryVoltage;  
 size\_t len = serializeJson(jsonDoc, jsonBuf, sizeof(jsonBuf));  
 webSocket.sendTXT((const uint8\_t\*)jsonBuf, len);  
 }  
  
 serialPrintCounter++;  
 if (serialPrintCounter >= PRINT\_EVERY\_N) {  
 serialPrintCounter = 0;  
 Serial.printf("SENT #%u t=%lu ms P/R/Y=%.1f/%.1f/%.1f T=%.1fC H=%.1f%% V=%.2fV\n",  
 seq, (unsigned long)ts\_ms,  
 local.pitch, local.roll, local.yaw,  
 local.temperature, local.humidity, local.batteryVoltage);  
 }  
  
 vTaskDelay(pdMS\_TO\_TICKS(SEND\_INTERVAL\_MS));  
 }  
}  
  
// ---------------- OLED Task ----------------  
void oledTask(void \*parameter) {  
 (void) parameter;  
 for (;;) {  
 display.clearDisplay();  
 display.setTextSize(2);  
 display.setTextColor(SSD1306\_WHITE);  
  
 // Line 1: Drone ON  
 display.setCursor(0, 0);  
 display.println("Drone ON");  
  
 // Line 2: WiFi  
 display.setTextSize(1);  
 display.setCursor(0, 25);  
 display.print("WiFi: ");  
 display.println(WiFi.isConnected() ? "OK" : "No");  
  
 // Line 3: WS Connection  
 display.setCursor(0, 40);  
 display.print("Server: ");  
 display.println(wsConnected ? "Connected" : "No link");  
  
 display.display();  
 vTaskDelay(pdMS\_TO\_TICKS(500)); // update every 0.5s  
 }  
}  
  
// ---------------- Setup ----------------  
void setup() {  
 Serial.begin(115200);  
 delay(100);  
  
 // Start I2C and OLED  
 Wire.begin(SDA\_PIN, SCL\_PIN);  
 if (!display.begin(SSD1306\_SWITCHCAPVCC, 0x3C)) {  
 Serial.println("SSD1306 allocation failed");  
 for (;;);  
 }  
 display.clearDisplay();  
 display.setTextSize(2);  
 display.setTextColor(SSD1306\_WHITE);  
 display.setCursor(0, 10);  
 display.println("Booting...");  
 display.display();  
  
 // WiFi  
 WiFi.begin(ssid, password);  
 Serial.print("Connecting WiFi");  
 uint32\_t wifiStart = millis();  
 while (WiFi.status() != WL\_CONNECTED) {  
 delay(300);  
 Serial.print(".");  
 if (millis() - wifiStart > 10000) {  
 Serial.println();  
 wifiStart = millis();  
 }  
 }  
 Serial.println();  
 Serial.print("Connected. IP: ");  
 Serial.println(WiFi.localIP());  
 WiFi.setSleep(false);  
  
 // MPU6050  
 if (!mpu.begin()) {  
 Serial.println("MPU6050 not found! Halting.");  
 while (1) { delay(1000); }  
 }  
 mpu.setAccelerometerRange(MPU6050\_RANGE\_8\_G);  
 mpu.setGyroRange(MPU6050\_RANGE\_500\_DEG);  
 mpu.setFilterBandwidth(MPU6050\_BAND\_21\_HZ);  
  
 // DHT + battery  
 dht.begin();  
 analogReadResolution(12);  
  
 // WebSocket  
 webSocket.begin(websocket\_host, websocket\_port, websocket\_path);  
 webSocket.onEvent(webSocketEvent);  
 webSocket.setReconnectInterval(3000);  
  
 // Mutex + RTOS tasks  
 dataMutex = xSemaphoreCreateMutex();  
 if (dataMutex == NULL) {  
 Serial.println("Failed to create mutex — halting.");  
 while (1) { delay(1000); }  
 }  
 xTaskCreatePinnedToCore(sensorTask, "SensorTask", 4096, NULL, 2, &sensorTaskHandle, 1);  
 xTaskCreatePinnedToCore(sendTask, "SendTask", 4096, NULL, 2, &sendTaskHandle, 1);  
 xTaskCreatePinnedToCore(oledTask, "OledTask", 4096, NULL, 1, &oledTaskHandle, 1);  
  
 Serial.println("Setup complete.");  
}  
  
// ---------------- Loop ----------------  
void loop() {  
 webSocket.loop();  
 delay(2);  
}

## Appendix B: Server Code (server.js)

// server.js  
// Simple low-latency WebSocket relay for ESP32 binary telemetry (32 bytes packets).  
// - Forwards binary packets unchanged to other connected clients (browsers).  
// - Disables permessage-deflate (compression) for lower latency.  
// - Sets TCP\_NODELAY to reduce small-packet buffering.  
  
const express = require('express');  
const http = require('http');  
const path = require('path');  
const WebSocket = require('ws');  
  
const PORT = process.env.PORT || 3000;  
const PUBLIC\_DIR = path.join(\_\_dirname, 'public'); // serves front-end  
  
// ---------- Express + HTTP server ----------  
const app = express();  
app.use(express.static(PUBLIC\_DIR));  
app.get('/', (req, res) => res.sendFile(path.join(PUBLIC\_DIR, 'index.html')));  
app.get('/health', (req, res) => res.json({ status: 'ok' }));  
  
const server = http.createServer(app);  
  
// ---------- WebSocket server (no compression) ----------  
const wss = new WebSocket.Server({  
 server,  
 perMessageDeflate: false // disable compression to reduce latency  
});  
  
console.log('Starting WebSocket server...');  
  
wss.on('connection', (ws, req) => {  
 const remote = req.socket.remoteAddress + ':' + req.socket.remotePort;  
 console.log(`Client connected: ${remote}`);  
  
 // Disable Nagle's algorithm (send small packets immediately)  
 if (ws.\_socket && typeof ws.\_socket.setNoDelay === 'function') {  
 ws.\_socket.setNoDelay(true);  
 }  
  
 ws.on('message', (message, isBinary) => {  
 // Fast path: binary telemetry packets (expected 32 bytes)  
 if (isBinary && Buffer.isBuffer(message) && message.length === 32) {  
 // Optional: quick parse for logging (non-blocking, low-cost)  
 try {  
 // seq: uint32le, ts\_ms: uint32le, then 6 floats (little-endian)  
 const seq = message.readUInt32LE(0);  
 const ts\_ms = message.readUInt32LE(4);  
 const pitch = message.readFloatLE(8);  
 const roll = message.readFloatLE(12);  
 const yaw = message.readFloatLE(16);  
 const temp = message.readFloatLE(20);  
 const hum = message.readFloatLE(24);  
 const batt = message.readFloatLE(28);  
  
 const now = Date.now();  
 const latencyEstimateMs = now - ts\_ms; // coarse E2E estimate (ESP32 millis -> server epoch might differ)  
  
 // Light logging (one line)  
 // Comment out if too chatty for performance  
 console.log(`T#${seq} | lat~${latencyEstimateMs}ms | P:${pitch.toFixed(1)} R:${roll.toFixed(1)} Y:${yaw.toFixed(1)} | T:${temp.toFixed(1)}C H:${hum.toFixed(1)}% V:${batt.toFixed(2)}V`);  
 } catch (err) {  
 // ignore parse errors, still forward below  
 // (we intentionally don't fail on parse errors)  
 }  
  
 // Broadcast the binary packet to all other connected clients (browsers)  
 wss.clients.forEach(client => {  
 if (client !== ws && client.readyState === WebSocket.OPEN) {  
 client.send(message, { binary: true }, (err) => {  
 if (err) {  
 // small error reporting; don't crash  
 console.warn('Forward error (binary):', err.message || err);  
 }  
 });  
 }  
 });  
  
 return; // done with this message  
 }  
  
 // Text / JSON path: forward as text to other clients  
 if (!isBinary) {  
 // optionally validate JSON, but here we just forward quickly  
 wss.clients.forEach(client => {  
 if (client !== ws && client.readyState === WebSocket.OPEN) {  
 client.send(message, { binary: false }, (err) => {  
 if (err) {  
 console.warn('Forward error (text):', err.message || err);  
 }  
 });  
 }  
 });  
 return;  
 }  
  
 // Other binary sizes: forward as-is (generic)  
 if (isBinary && Buffer.isBuffer(message)) {  
 wss.clients.forEach(client => {  
 if (client !== ws && client.readyState === WebSocket.OPEN) {  
 client.send(message, { binary: true }, (err) => {  
 if (err) console.warn('Forward error (other-binary):', err.message || err);  
 });  
 }  
 });  
 return;  
 }  
 });  
  
 ws.on('close', (code, reason) => {  
 console.log(`Client disconnected: ${remote} code=${code} reason=${String(reason).slice(0,100)}`);  
 });  
  
 ws.on('error', (err) => {  
 console.warn(`WS error from ${remote}:`, err && err.message ? err.message : err);  
 });  
});  
  
// start server  
server.listen(PORT, () => {  
 console.log(`HTTP + WS server listening on http://localhost:${PORT}`);  
});

## Appendix C: Sample Frontend Code (from index.html or related JS)

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width,initial-scale=1" />

<title>Drone Telemetry Dashboard</title>

<link rel="stylesheet" href="styles.css">

<!-- Three.js from CDN -->

<script src="https://cdn.jsdelivr.net/npm/three@0.160.0/build/three.min.js"></script>

</head>

<body>

<div id="container">

<header>

<h1>Drone Telemetry — Real-time Dashboard</h1>

<div id="status">WS: <span id="wsStatus">connecting...</span></div>

</header>

<main>

<section class="left-panel">

<div id="three-container"></div>

<div class="metrics">

<div class="metric">

<div class="label">Pitch</div>

<div id="pitchVal" class="value">--°</div>

</div>

<div class="metric">

<div class="label">Roll</div>

<div id="rollVal" class="value">--°</div>

</div>

<div class="metric">

<div class="label">Yaw</div>

<div id="yawVal" class="value">--°</div>

</div>

</div>

</section>

<aside class="right-panel">

<div class="card">

<h3>Temperature</h3>

<div id="tempGauge" class="gauge">

<div id="tempFill" class="gauge-fill">--°C</div>

</div>

</div>

<div class="card">

<h3>Battery</h3>

<div id="battery" class="battery">

<div id="batteryFill" class="battery-fill"></div>

<div id="batteryPct" class="battery-pct">--%</div>

</div>

<div class="note" id="batteryVolt">Voltage: -- V</div>

</div>

<div class="card small">

<h3>Humidity</h3>

<div id="humidityVal" class="value large">--%</div>

</div>

</aside>

</main>

<footer>

<small>  <code>server.js</code> </small>

</footer>

</div>

<script src="script.js"></script>

</body>

</html>

## Appendix D: Project Repository

GitHub link to source code and related files:  
<https://github.com/Kazimmohamed/drone_telemetry_dashboard>

# References

1. Espressif Systems. *ESP32 Technical Reference Manual*. [Online]. Available: https://docs.espressif.com
2. Arduino. *Arduino IDE Documentation*. [Online]. Available: https://docs.arduino.cc/
3. Adafruit Industries. *Adafruit MPU6050 Library*. [Online]. Available: <https://github.com/adafruit/Adafruit_MPU6050>
4. Adafruit Industries. *Adafruit SSD1306 OLED Library*. [Online]. Available: <https://github.com/adafruit/Adafruit_SSD1306>
5. Adafruit Industries. *DHT Sensor Library*. [Online]. Available: <https://github.com/adafruit/DHT-sensor-library>
6. Benoit Blanchon. *ArduinoJson Library*. [Online]. Available: https://arduinojson.org/
7. Node.js Foundation. *Node.js Documentation*. [Online]. Available: https://nodejs.org/en/docs/
8. WebSocket API. *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\_API
9. Chart.js. *Chart.js Documentation*. [Online]. Available: https://www.chartjs.org/docs/
10. Three.js. *Three.js Documentation*. [Online]. Available: https://threejs.org/docs/