Machine Monitoring of Drilling Machine using IoT

Real-time Monitoring of Temperature, Vibration, and Position

GANESH HATWADE 23BSM025

BRANCH : SMART MANUFACTURING

Institution: PDDM IIITDMJ

CONTENT

1.INTRODUCTION

2.PROCEDURE STEPS

3.MACHINE DISCRIPTION

4.CIRCUIT DISCRIPTION

5.ESP32 SETUP

6.SOFTWARE SECTION

7. CHALLANGE & OBSTACLES

8.FUTURE SCOPE AND IMPROVEMENT

9.CONCLUSION

Introduction

Conventional drilling machines, widely used in industrial and educational workshops, often operate without any form of digital monitoring. This project proposes an IoT-based retrofitting solution that enables real-time health monitoring of such machines. The system integrates sensors to collect vital machine condition data and uses the ESP32 microcontroller to transmit data to a live dashboard via the internet.

The aim is to prevent unexpected failures, reduce downtime, and promote predictive maintenance — all at a low cost and without modifying the machine's mechanical structure.

Procedure Steps

1. Hardware Setup

- - Assemble sensors and connect to ESP32.
- - Ensure stable power supply and signal integrity.

2. Programming

- - Configure the ESP32 using Arduino IDE.
- - Implement sensor data reading and transmission.

3. Data Pipeline

- - Send data to InfluxDB or an HTTP endpoint.
- - Visualize real-time values on Grafana and a custom website.

4. Dashboard Configuration

- - Set up panels and alerts in Grafana.
- - Style the custom website using HTML/CSS.

Machine description(Radial Drilling Machine)

Radial Drilling Machine – Description of Each Tool/Component

A radial drilling machine is used to drill holes in large and heavy workpieces that cannot be easily moved. The arm of the machine can rotate and move horizontally, allowing flexibility in drilling operations.

1. Base

- A large and heavy casting that supports the entire machine.
- Provides stability and absorbs vibrations during drilling.
- Usually bolted to the floor.

2. Column

- A vertical cylindrical structure mounted on the base.
- It supports the radial arm and allows its vertical movement.
- The radial arm can rotate around the column for radial movement.

3. Radial Arm

- Horizontally mounted on the column; can move up/down and rotate around it.
- Carries the drilling head (drill spindle and motor).
- Allows the drill to reach different positions on a large workpiece.

4. Drill Head (Drilling Unit)

- Contains the motor, spindle, chuck, and feed mechanism.
- Mounted on the radial arm and can slide horizontally.
- Moves toward or away from the workpiece for precision alignment.

5. Spindle

- A rotating shaft that holds and drives the cutting tool (drill bit).
- Connected to the motor via gears or belts.
- Rotational speed can be adjusted depending on the material.

6. Chuck

- A clamping device attached to the end of the spindle.
- Holds the drill bit firmly during rotation.
- Can be a keyed or keyless chuck.

7. Motor

- Powers the spindle and drives the drilling operation.
- Can be single-phase or three-phase, depending on machine size.

8. Vertical Lifting Mechanism

- Used to move the radial arm up and down the column.
- Powered manually or by motor, depending on machine design.

9. Work Table

- Flat surface where the workpiece is clamped during drilling.
- May have T-slots to fix vices or clamps.
- Can sometimes be rotated or moved vertically.

10. Control Panel / Feed Mechanism

- Allows the operator to control spindle speed, feed rate, and power.
- Some advanced machines include automatic feed and depth stop.

Circuit Description

Sensors Used

1.**MPU6050** - 6-Axis Accelerometer

The **MPU6050** is a popular 6-axis motion tracking device that combines a 3-axis gyroscope and a 3-axis accelerometer on a single chip.

Parameter	Specification
Sensor Type	3-axis gyroscope + 3-axis accelerometer
Communication	I2C (up to 400kHz) or SPI (optional on some modules)
Operating Voltage	3.3V – 5V (depends on breakout board; chip itself is 3.3V)
Gyroscope Range	±250, ±500, ±1000, ±2000 °/s (configurable)
Accelerometer Range	±2g, ±4g, ±8g, ±16g (configurable)
Sensitivity (Gyro)	131 LSB/(°/s) at ±250 °/s
Sensitivity (Accel)	16384 LSB/g at ±2g
Temperature Sensor	Included, with accuracy of ±1°C
Operating Temperature	-40°C to +85°C
Power Consumption	~3.6mA in normal mode
Digital Output	16-bit ADC resolution
Motion Detection	Built-in motion detection and interrupt support
FIFO Buffer	1024 bytes
Dimensions	Depends on module (typical: 20mm x 15mm x 2mm)

2. **DS18B20** - Temperature Sensor

the **DS18B20** is a widely used digital temperature sensor that provides high accuracy and supports communication via the **1-Wire** protocol.

Parameter	Specification
Sensor Type	Digital temperature sensor
Communication	1-Wire digital interface (requires only one data pin)
Operating Voltage	3.0V to 5.5V
Temperature Range	-55°C to +125°C
Accuracy	±0.5°C from -10°C to +85°C
Resolution	9 to 12 bits (user selectable)
Conversion Time	93.75ms (9-bit) to 750ms (12-bit)
Unique Address	Each sensor has a unique 64-bit serial code
Power Modes	Normal and Parasite Power modes
Output Format	Digital (binary temperature data via 1-Wire)
Typical Standby Current	<1 µA
Dimensions	Varies by package (e.g., TO-92, waterproof capsule, etc.)

3. **HC-SR04** – Ultrasonic Distance Sensor

The **HC-SR04** is a commonly used ultrasonic sensor for measuring distance using sound waves. It's widely used in robotics, obstacle detection, and distance monitoring applications.

Parameter	Specification
Sensor Type	Ultrasonic proximity/distance sensor
Operating Voltage	5V DC
Operating Current	15 mA (typical)

Parameter	Specification
Measuring Range	2 cm to 400 cm (0.02 m to 4 m)
Best Accuracy Range	2 cm to 100 cm
Accuracy	±3 mm
Resolution	Approx. 1 mm
Measuring Angle	~15°
Trigger Input Signal	10 µs TTL pulse
Echo Output Signal	Pulse width proportional to distance
Interface Type	Digital (Trigger and Echo pins)
Dimensions	Approx. 45mm × 20mm × 15mm

4. **ESP32** – Wi-Fi + Bluetooth Microcontroller

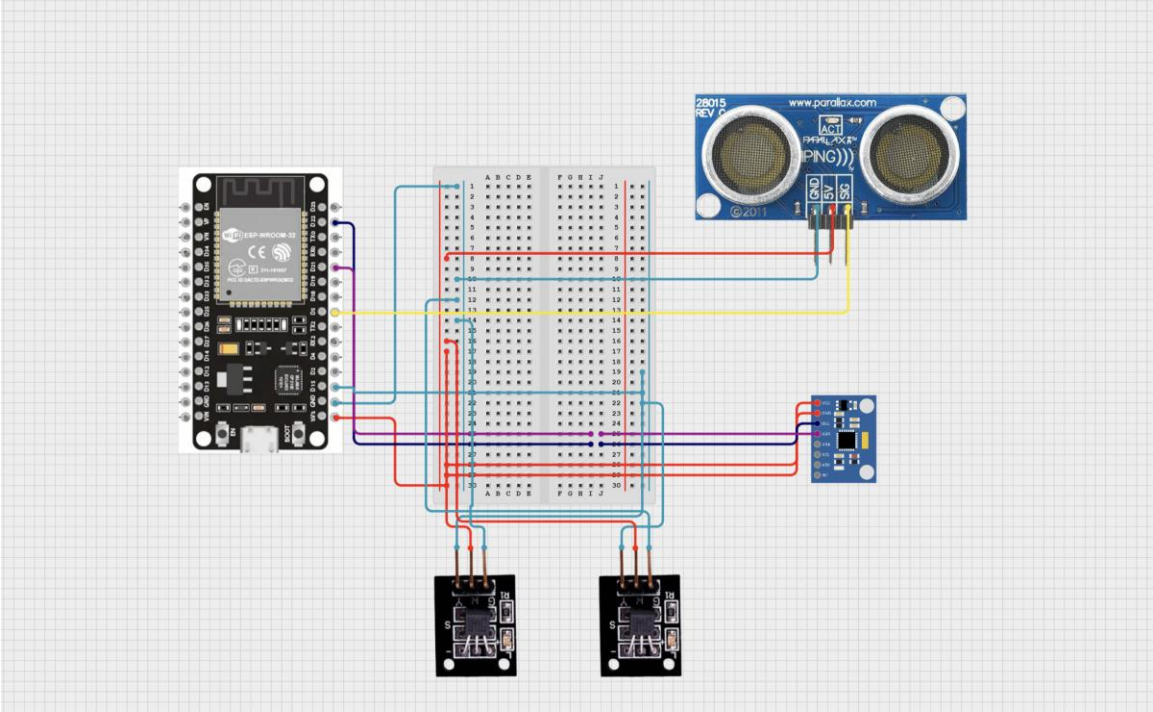
The **ESP32** is a powerful dual-core microcontroller with built-in Wi-Fi and Bluetooth, ideal for IoT projects. It supports a wide range of sensors, protocols, and applications.

- **Integrated Wi-Fi and Bluetooth** for wireless communication.
- **GPIOs support PWM**, capacitive touch, ADC, DAC, and interrupts.
- **Multiple Serial Interfaces** (UART, I2C, SPI) for sensor connections.
- **Deep Sleep Mode** for battery-powered applications.
- Compatible with **Arduino IDE**, PlatformIO, and ESP-IDF.
- Ideal for real-time monitoring, automation, and IoT systems.

Pin-by-Pin Circuit Description

Sensor	ESP32 Pin	Protocol	Notes
Mpu6050	SDA → D21	I2C	SCL → D22

DS18B20	D4	1-Wire	Pull-up resistor needed
HC-SR04	SDA → D21	I2C	Shares I2C bus
All Sensors	GND	-	Common ground
Power	3.3V/5V	-	Check sensor



ESP32 setup code

```
#include <WiFi.h>

#include <HTTPClient.h>

#include <Wire.h>

#include <OneWire.h>

#include <MPU6050.h>


// === WiFi Credentials ===

const char* ssid = "SDP";

const char* password = "123456789";


// === InfluxDB Info ===

const char* influxUrl = "https://us-east-1-1.aws.cloud2.influxdata.com";

const char* org = "6ec95b36c4baacb1";

const char* bucket = "CPPS";

const char* token = "Gti6d2hFaA11bZvausBTrW8-4UmzR5JYVx1wPSS1UcjFTpxosG-
BdKpPnYshjtDZIvjz2l_jfUN6a8Ttott4kQ==";


// === DS18B20 Setup ===

OneWire ds(15);

byte sensor1[8] = { 0x28, 0x61, 0x80, 0x3D, 0x00, 0x00, 0x00, 0xF4 };

byte sensor2[8] = { 0x28, 0x3F, 0xA3, 0x39, 0x00, 0x00, 0x00, 0x71 };
```

```
// === MPU6050 Setup ===
MPU6050 mpu;
const float accelScale = 9.81 / 16384.0;
const float gyroScale = 3.14159265 / (180.0 * 131.0);

// === HC-SR04 Pins ===
const int trigPin = 4;
const int echoPin = 2;

void setup() {
 Serial.begin(115200);

 // === WiFi Connect ===
 WiFi.begin(ssid, password);
 Serial.print("Connecting to WiFi");
 while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
 }
 Serial.println("\nConnected to WiFi");

 // === MPU6050 Init ===
 Wire.begin(21, 22);
 mpu.initialize();
 if (mpu.testConnection()) {
```

```
  Serial.println("MPU6050 connection successful");
 } else {
  Serial.println("MPU6050 connection failed");
 }

 // === HC-SR04 Init ===
 pinMode(trigPin, OUTPUT);
 pinMode(echoPin, INPUT);
}

// === DS18B20 Read ===
float readTemperature(byte addr[8]) {
 byte data[9];
 ds.reset();
 ds.select(addr);
 ds.write(0x44, 1);
 delay(750);
 ds.reset();
 ds.select(addr);
 ds.write(0xBE);
 for (int i = 0; i < 9; i++) {
  data[i] = ds.read();
 }
 int16_t raw = (data[1] << 8) | data[0];
 return (float)raw / 16.0;
}
```

```
// === HC-SR04 Read (cm) ===
float readDistanceCM() {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  long duration = pulseIn(echoPin, HIGH, 30000);  // Timeout after 30ms
  return duration * 0.0343 / 2;
}

void loop() {
  // === DS18B20 ===
  float temp1 = readTemperature(sensor1);
  float temp2 = readTemperature(sensor2);
  float avgTemp = (temp1 + temp2) / 2.0;

  // === MPU6050 ===
  int16_t ax_raw, ay_raw, az_raw;
  int16_t gx_raw, gy_raw, gz_raw;
  mpu.getMotion6(&ax_raw, &ay_raw, &az_raw, &gx_raw, &gy_raw, &gz_raw);

  float ax = ax_raw * accelScale;
  float ay = ay_raw * accelScale;
  float az = az_raw * accelScale;
```

```
float gx = gx_raw * gyroScale;
float gy = gy_raw * gyroScale;
float gz = gz_raw * gyroScale;

// === HC-SR04 ===
float distance = readDistanceCM();

// === Serial Output ===
Serial.println("=== Sensor Data (SI Units) ===");
Serial.printf("Temp1: %.2f °C, Temp2: %.2f °C, Avg: %.2f °C\n", temp1, temp2, avgTemp);
Serial.printf("Accel - X: %.2f m/s^2, Y: %.2f m/s^2, Z: %.2f m/s^2\n", ax, ay, az);
Serial.printf("Gyro  - X: %.4f rad/s, Y: %.4f rad/s, Z: %.4f rad/s\n", gx, gy, gz);
Serial.printf("Distance: %.2f cm\n", distance);

// === Influx Line Protocol ===
String data = "esp32_data,device=ESP32 ";
data += "temp1=" + String(temp1, 2) + ",";
data += "temp2=" + String(temp2, 2) + ",";
data += "avg_temp=" + String(avgTemp, 2) + ",";
data += "accel_x=" + String(ax, 2) + ",";
data += "accel_y=" + String(ay, 2) + ",";
data += "accel_z=" + String(az, 2) + ",";
data += "gyro_x=" + String(gx, 4) + ",";
data += "gyro_y=" + String(gy, 4) + ",";
data += "gyro_z=" + String(gz, 4) + ",";
```

```
  data += "distance_cm=" + String(distance, 2);


  // === Upload to InfluxDB ===
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    String fullUrl = String(influxUrl) + "/api/v2/write?org=" + org + "&bucket=" + bucket +
"&precision=s";


    http.begin(fullUrl);
    http.addHeader("Authorization", "Token " + String(token));
    http.addHeader("Content-Type", "text/plain");


    int response = http.POST(data);
    Serial.print("InfluxDB response: ");
    Serial.println(response);
    if (response != 204) {
      Serial.println("Error: " + http.getString());
    }
    http.end();
  } else {
    Serial.println("WiFi not connected!");
  }


  delay(10000);  // every 10 sec
}
```

Software Section

Programming Platform

Arduino IDE is used to program the ESP32 microcontroller using C++ with the help of several libraries.

- Libraries Used:
- - Wire.h
- - OneWire.h
- - DallasTemperature.h
- - Adafruit_Sensor.h

Dashboard & Web Interface

Grafana Dashboard

Grafana displays real-time plots for each sensor. It provides threshold lines, time-based visualization, and export options.

 Live Graphs: Continuously updated charts for vibration, temperature, and distance.

 Time-Range Selector: Zoom into recent data or view historical trends.

 Custom Thresholds: Display red/green zones for temperature/vibration limits.

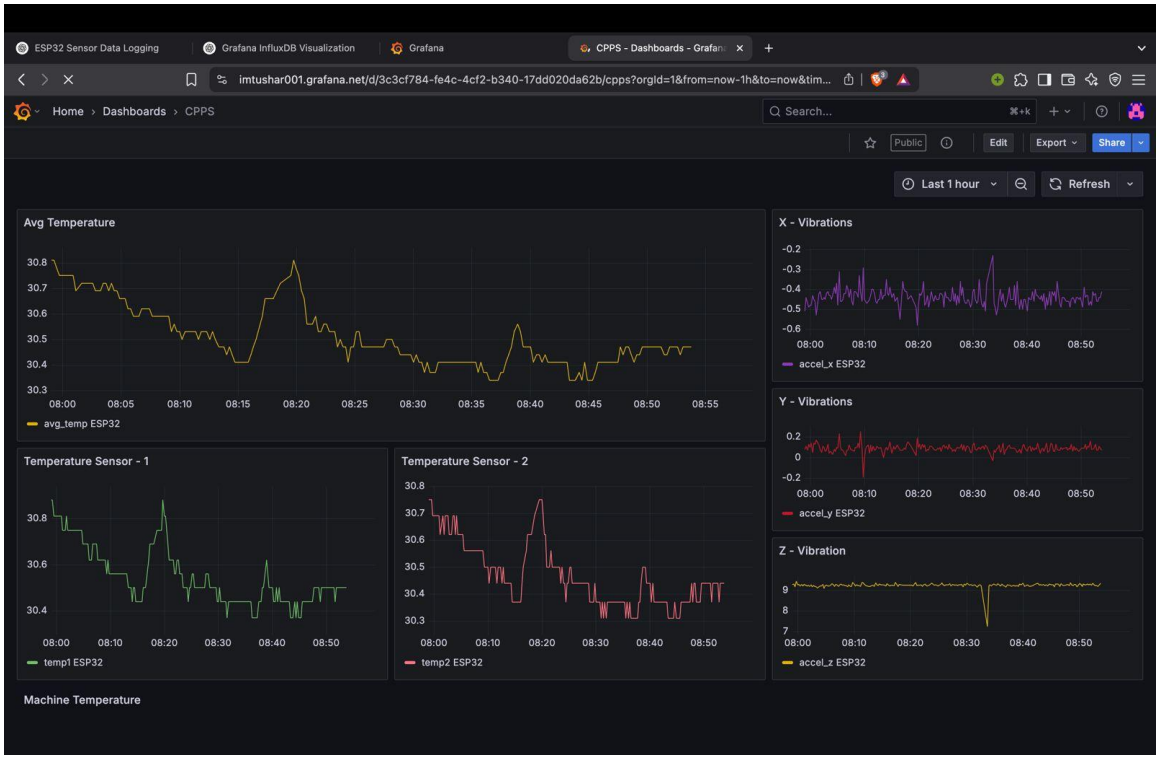
 Multi-Sensor Panels: Display data from all sensors in one unified view.

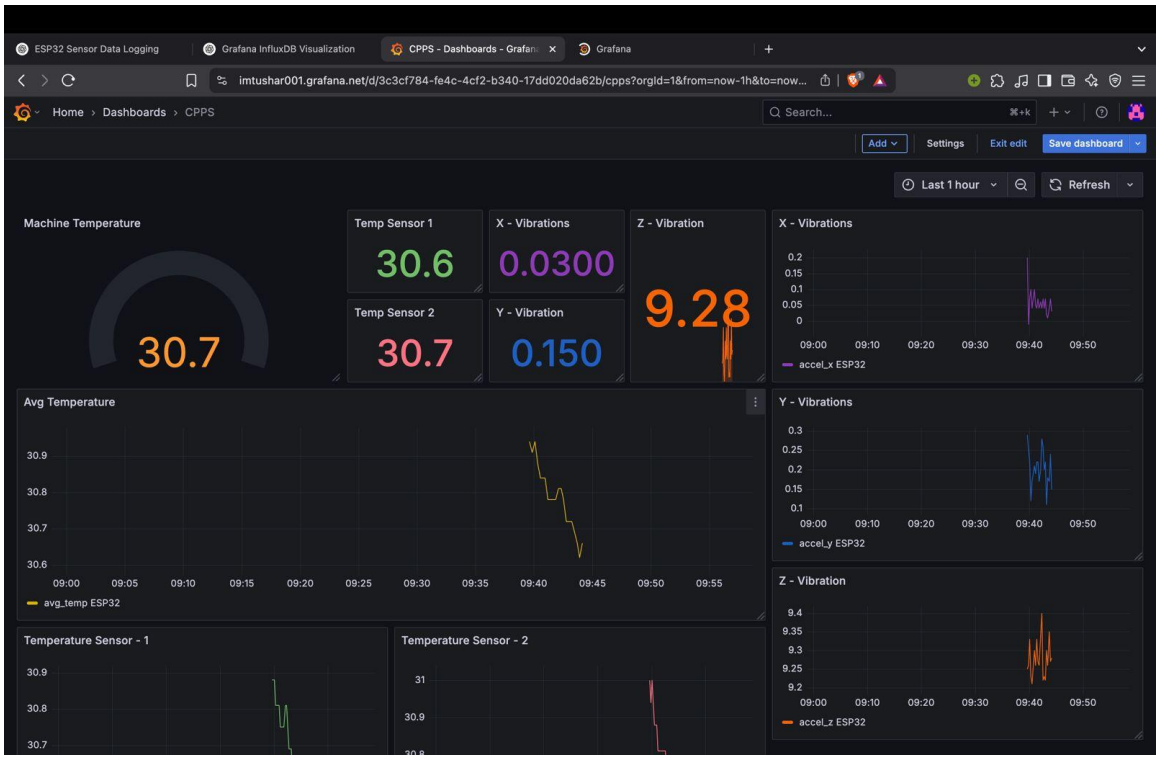
 Annotations: Mark important events on the timeline (e.g., power loss, overheating).

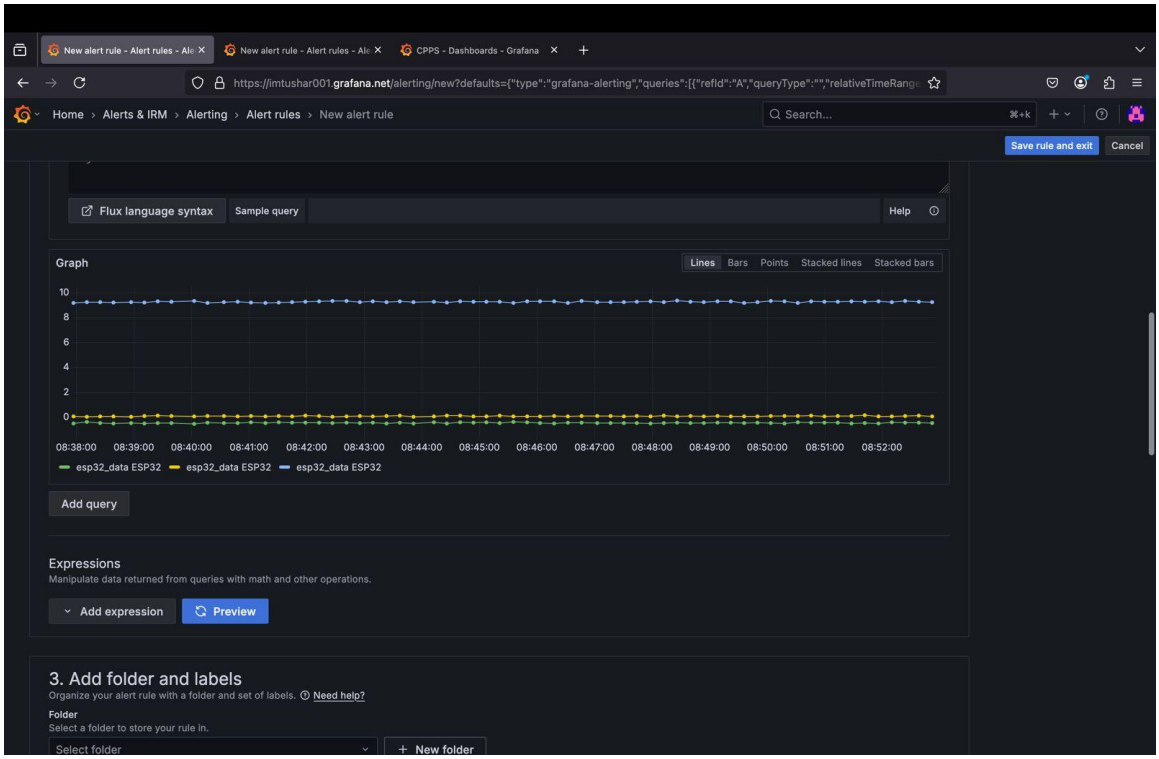
 Alerting Rules: Trigger notifications if sensor data crosses thresholds.

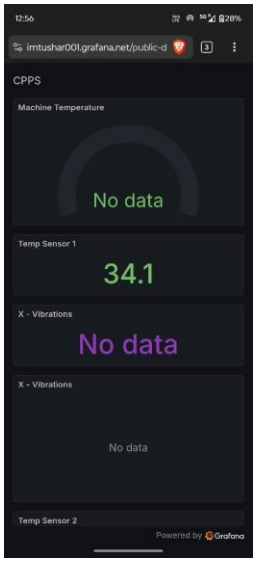
 User Access Control: Share dashboards with team members or restrict with roles.

 Integrations: Easily integrate with InfluxDB (used in your project) and MQTT brokers.


ESP32 Sensor Data Logging
Grafana InfluxDB Visualization
Grafana
CPPS - Dashboards - Grafana
imtushar001.grafana.net/d/3c3cf784-fe4c-4cf2-b340-17dd020da62b/cpps?orgId=1&from=now-1h&to=now&tim...
Home › Dashboards › CPPS
Search...
Public
Edit
Export
Share
Last 1 hour
Refresh
Avg Temperature
avg_temp ESP32
X - Vibrations
accel_x ESP32
Y - Vibrations
accel_y ESP32
Z - Vibration
accel_z ESP32
Temperature Sensor - 1
temp1 ESP32
Temperature Sensor - 2
temp2 ESP32
Machine Temperature


ESP32 Sensor Data Logging
Grafana InfluxDB Visualization
CPPS - Dashboards - Grafana
Grafana
imtushar001.grafana.net/d/3c3cf784-fe4c-4cf2-b340-17dd020da62b/cpps?orgId=1&from=now-1h&to=now...
Home › Dashboards › CPPS
Search...
Add
Settings
Exit edit
Save dashboard
Last 1 hour
Refresh
Machine Temperature
30.7
Temp Sensor 1
30.6
X - Vibrations
0.0300
Z - Vibration
9.28
Temp Sensor 2
30.7
Y - Vibration
0.150
X - Vibrations
accel_x ESP32
Avg Temperature
avg_temp ESP32
Y - Vibrations
accel_y ESP32
Z - Vibration
accel_z ESP32
Temperature Sensor - 1
Temperature Sensor - 2





MOBILE DASHBOARD (WHEN MACHINE IS OFF)

Custom Website

A user-friendly dashboard created using HTML, CSS, and JavaScript. Data is fetched using HTTP and visualized using Chart.js or Plotly.

Here's a basic example of a **user-friendly web dashboard** built using **HTML**, **CSS**, and **JavaScript** with **Chart.js** to visualize live sensor data (temperature, vibration, and

distance). This version assumes you're fetching data from your ESP32 via HTTP in JSON format.

🌐 **Example: IoT Sensor Dashboard (Frontend Code)**

📁 **File: index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <title>Drilling Machine Health Dashboard</title>
 <link rel="stylesheet" href="style.css">
 <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body>
 <h1>Machine Health Dashboard</h1>
 <div class="chart-container">
  <canvas id="tempChart"></canvas>
  <canvas id="vibChart"></canvas>
  <canvas id="distChart"></canvas>
 </div>
 <script src="script.js"></script>
</body>
</html>
```

File: style.css

```
body {
  font-family: Arial, sans-serif;
  text-align: center;
  background-color: #f5f7fa;
  padding: 20px;
}

h1 {
  margin-bottom: 20px;
  color: #333;
}

.chart-container {
  display: grid;
  grid-template-columns: 1fr;
  gap: 20px;
  max-width: 900px;
  margin: auto;
}

canvas {
  background: #fff;
  border: 1px solid #ddd;
  padding: 10px;
  border-radius: 10px;
```

```
}
```

File: script.js

```
const tempCtx = document.getElementById('tempChart').getContext('2d');
const vibCtx = document.getElementById('vibChart').getContext('2d');
const distCtx = document.getElementById('distChart').getContext('2d');

const tempChart = new Chart(tempCtx, {
  type: 'line',
  data: {
    labels: [],
    datasets: [{
      label: 'Temperature (°C)',
      borderColor: 'red',
      data: [],
      fill: false
    }]
  }
});

const vibChart = new Chart(vibCtx, {
  type: 'line',
  data: {
    labels: [],
    datasets: [{
      label: 'Vibration (g)',
```

```
      borderColor: 'blue',
      data: [],
      fill: false
    }]
  }
});

const distChart = new Chart(distCtx, {
  type: 'line',
  data: {
    labels: [],
    datasets: [{
      label: 'Distance (cm)',
      borderColor: 'green',
      data: [],
      fill: false
    }]
  }
});

function fetchData() {
  fetch('http://<ESP32_IP>/data') // Replace with your ESP32 IP
    .then(response => response.json())
    .then(data => {
      const time = new Date().toLocaleTimeString();
```

```
    tempChart.data.labels.push(time);
    tempChart.data.datasets[0].data.push(data.temperature);
    tempChart.update();

    vibChart.data.labels.push(time);
    vibChart.data.datasets[0].data.push(data.vibration);
    vibChart.update();

    distChart.data.labels.push(time);
    distChart.data.datasets[0].data.push(data.distance);
    distChart.update();

    // Keep only the last 20 data points
    [tempChart, vibChart, distChart].forEach(chart => {
      if (chart.data.labels.length > 20) {
        chart.data.labels.shift();
        chart.data.datasets[0].data.shift();
      }
    });
  })
  .catch(err => console.error("Failed to fetch data:", err));
}

// Fetch every 2 seconds
setInterval(fetchData, 2000);
```

Expected ESP32 Output Format (via /data endpoint):

{

"temperature": 32.5,

"vibration": 0.12,

"distance":08

Conclusion : we have a just tested it we cannot perform complete task on this website

Own website image

Working Flow

1. **1. ESP32 connects to Wi-Fi and initializes sensors**
   - **Wi-Fi Connection**: When the ESP32 powers on, it attempts to connect to a predefined Wi-Fi network using the SSID and password configured in the code.
   - **Sensor Initialization**:
     - The **MP6050**(vibration sensor) is initialized via the I2C protocol.

    - The **DS18B20** (temperature sensor) is initialized using the OneWire protocol.
    - The **HCSR04** or **Ultrasonic Sensor** (distance sensor) is also initialized over I2C or digital pins.
- **Status Feedback**: Once all sensors are active and ESP32 is connected, the device may indicate successful setup via a serial message or onboard LED blink.

2. **2. Data is read, formatted, and transmitted**
    - **Sensor Reading Loop**:
        - ESP32 reads raw values from each sensor at regular intervals (e.g., every 1–2 seconds).
        - Temperature is read in °C, vibration as acceleration in g, and distance in cm or mm.
    - **Data Formatting**:
        - The sensor values are formatted into a structured format, usually **JSON** (e.g., {temperature: 34.2, vibration: 0.03, distance: 52}).
    - **Transmission Methods**:
        - **HTTP POST**: Data is sent to an endpoint (e.g., InfluxDB's /write API).
        - **MQTT (optional)**: ESP32 could publish sensor values to MQTT topics.
        - **Local Web Server**: ESP32 may host a lightweight webpage with live readings using AJAX or SSE.

3. **3. InfluxDB stores the data**
    - **Time-Series Database**: InfluxDB is optimized for storing and querying time-stamped data. It's perfect for IoT applications.
    - **Data Schema**:
        - Each sensor's reading is stored as a **measurement** (e.g., temperature, vibration, distance).
        - Tags can be used for metadata (e.g., machine_id=drill_01).
        - The **timestamp** is automatically added or included by ESP32.
    - **Example Line Protocol Format**:
    - temperature,sensor=DS18B20 value=34.2
    - vibration,sensor=ADXL345 x=0.02,y=0.01,z=0.03

4. **4. Grafana and website fetch and display the data in real-time**
5. 🔹 **Grafana Dashboard**
    - **Visualization Panels**:
        - Temperature → Gauge or Line Chart
        - Vibration → Waveform Graph
        - Distance → Line Graph with Threshold Zones
    - **Real-time Refresh**: Panels auto-refresh at defined intervals (e.g., every 5 seconds).
    - **Threshold Alerts**: Grafana can show red zones or even send alerts (email/Slack) if values exceed limits.

6. 🔹 **Custom Website**
   - **Frontend (HTML/CSS/JS)**:
     - Uses AJAX or WebSocket to pull live data from ESP32 or a REST API.
     - Graphs can be rendered using libraries like **Chart.js** or **Plotly**.
   - **Responsiveness**:
     - Web dashboard is optimized for PC, tablet, or mobile view.
   - **Features**:
     - Real-time charts, raw value display, possibly a refresh or settings menu.

Challenges & Obstacles

explanation for each challenge:

- **Sensor calibration for vibration was difficult**: Fine-tuning the ADXL345 sensor to accurately detect minor vibrations required repeated testing and filtering to eliminate noise.
- **Wi-Fi dropout in metallic environment**: The presence of metal structures around the drilling machine weakened the Wi-Fi signal, causing intermittent disconnections.
- **InfluxDB setup and syntax learning curve**: Initial configuration of InfluxDB and understanding its line protocol and query language (Flux/InfluxQL) took time and effort.
- **Managing real-time synchronization**: Ensuring that data updates appeared simultaneously on both the Grafana dashboard and custom website required precise timing and efficient data handling.

Future Scope & Improvements

Here is the **future scope** of your project explained in the same concise manner:

- **Add AI-based predictive alerts**: Integrating machine learning models could help predict failures in advance based on sensor data trends.
- **Use motor current sensors for load analysis**: Monitoring motor current can provide insights into mechanical load and potential stalling or overloading.

- **Implement email/SMS alerts on threshold breach**: Real-time alerts can notify users instantly when dangerous conditions are detected.
- **Monitor multiple machines via MQTT**: Using MQTT protocol allows scalable monitoring of several machines from a centralized system.
- **Make website mobile-friendly with secure access**: A responsive and secured web dashboard would improve usability and data privacy for operators.

Conclusion

Here is the **future scope** of your project explained in the same concise manner:

- **Add AI-based predictive alerts**: Integrating machine learning models could help predict failures in advance based on sensor data trends.
- **Use motor current sensors for load analysis**: Monitoring motor current can provide insights into mechanical load and potential stalling or overloading.
- **Implement email/SMS alerts on threshold breach**: Real-time alerts can notify users instantly when dangerous conditions are detected.
- **Monitor multiple machines via MQTT**: Using MQTT protocol allows scalable monitoring of several machines from a centralized system.
- **Make website mobile-friendly with secure access**: A responsive and secured web dashboard would improve usability and data privacy for operators.