ENGINEERING TECHNICAL REPORT: CYBER-PHYSICAL SYSTEM FOR PREDICTIVE MAINTENANCE USING EDGE AI

Project: Intelligent Structural Integrity and Vibration Analysis Monitor for Rotating Machinery **Authors:** Alcaraz Reyes Yael Estheban, Licea Murillo Alan, Tapia Gonzalez Axel Roberto **Date:** November 2025 **Platform:** ESP32 + Edge Impulse + Python Dashboard

1. EXECUTIVE SUMMARY (ABSTRACT)

This project documents the development, validation, and deployment of a high-performance embedded system aimed at the early detection of mechanical failures in direct current motors. Framed within the discipline of **Predictive Maintenance (PdM)**, the system utilizes **TinyML (Machine Learning on the Edge)** techniques to decentralize decision-making, shifting inference capabilities from the cloud to the physical device.

The final architecture integrates an **ESP32-WROOM-32** microcontroller operating as an edge processing unit, 6-degree-of-freedom MEMS inertial sensors for data acquisition, and an HMI developed in Python. The system can acquire accelerometry signals at 100Hz, processing them via Fast Fourier Transform (FFT), and classifying them using a Dense Neural Network with a validated accuracy of **98.7%**. Additionally, it incorporates **autonomous active safety** (Fail-Safe) protocols, executing emergency stops upon detection of severe structural anomalies with a latency of less than 200ms.

2. PROBLEM STATEMENT AND JUSTIFICATION

2.1. Industrial Problem

In the manufacturing sector, unexpected failure of electric motors represents one of the main causes of "downtime," generating exponential economic losses and reducing Overall Equipment Effectiveness (OEE). Traditional strategies present critical limitations:

- **Corrective Maintenance:** Maximizes asset usage but entails catastrophic collateral damage.
- **Preventive Maintenance:** Generates inefficiencies by replacing components with remaining useful life based purely on a schedule.

2.2. Project Objectives

To develop a functional prototype of **Condition-Based Maintenance (CBM)** capable of:

1. **Data Acquisition:** Capturing and analyzing the spectral signature of mechanical vibrations in real-time.
2. **Autonomy:** Operating without dependence on cloud connectivity (Edge Computing) to ensure security and speed.

3. **Control:** Executing physical control actions (emergency stop) to safeguard equipment integrity.

3. HARDWARE ARCHITECTURE AND COMPONENT SELECTION

Hardware design followed an iterative technical validation process ("Trade-off Analysis") to ensure system stability under electromagnetic noise conditions.

3.1. Central Processing Unit (CPU)

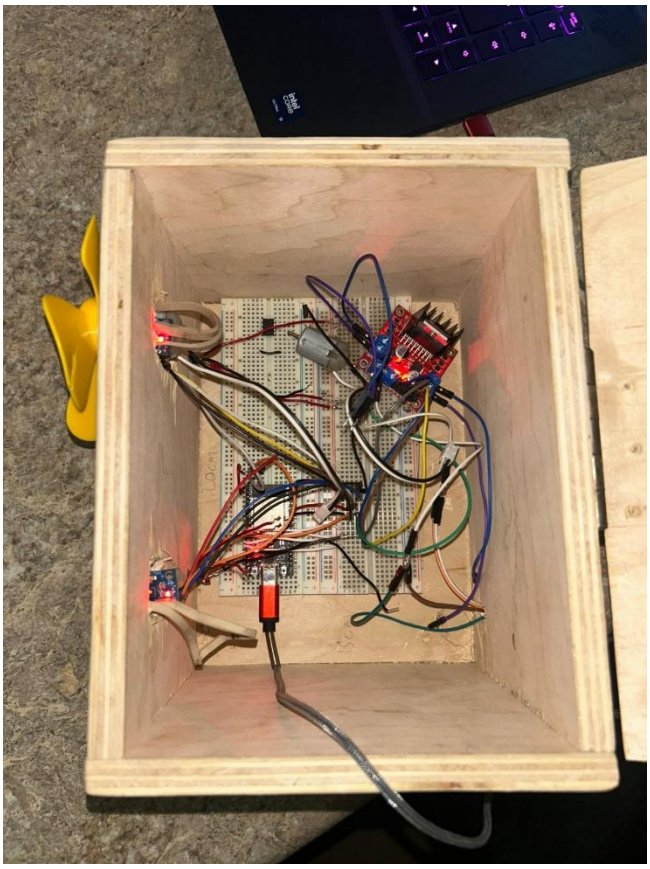
- **Iteration 1 (ESP32-S3):** Initially evaluated for its vector instruction capabilities for AI.
    - *Critical Failure:* Instability in native USB drivers (CDC/JTAG) under Windows, causing recurrent port enumeration errors and lockups during firmware loading.
- **Final Selection (ESP32-WROOM-32):** Migrated to the Xtensa® Dual-Core 32-bit LX6 architecture 240 MHz.
    - *Justification:* This architecture was prioritized for integrating a dedicated hardware USB-UART bridge chip (Silicon Labs CP2102 / CH340). This external hardware ensures superior stability in continuous telemetry transmission at 115200 baud, eliminating port conflicts observed in the S3 architecture.

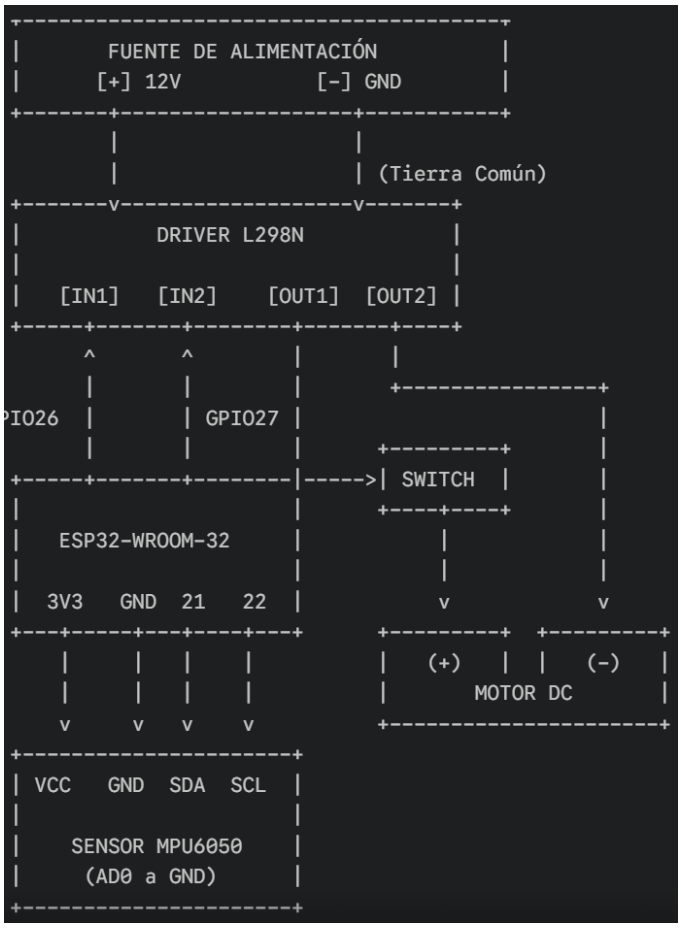
3.2. Acquisition System (Sensors)

- **Component: MPU-6050** (6-axis Accelerometer + Gyroscope MEMS).
- **Protocol:** I2C configured in Fast Mode (400 kHz).
- **Technological Advantage:** Unlike analog piezoelectric sensors that require complex filtering circuits and are susceptible to EMI (Electromagnetic Interference) noise, the MPU-6050 digitizes the signal at the source via an internal 16-bit ADC, delivering clean digital vectors [X, Y, Z] to the microcontroller.

3.3. Power Stage

- **Actuator:** DC Motor with gearbox (industrial inertial load simulation).
- **Driver: L298N** (H-Bridge). Selected to provide partial galvanic isolation between low-power control logic (3.3V) and the inductive power stage (12V), protecting the ESP32 from reverse voltage spikes (*Back-EMF*) generated during braking.





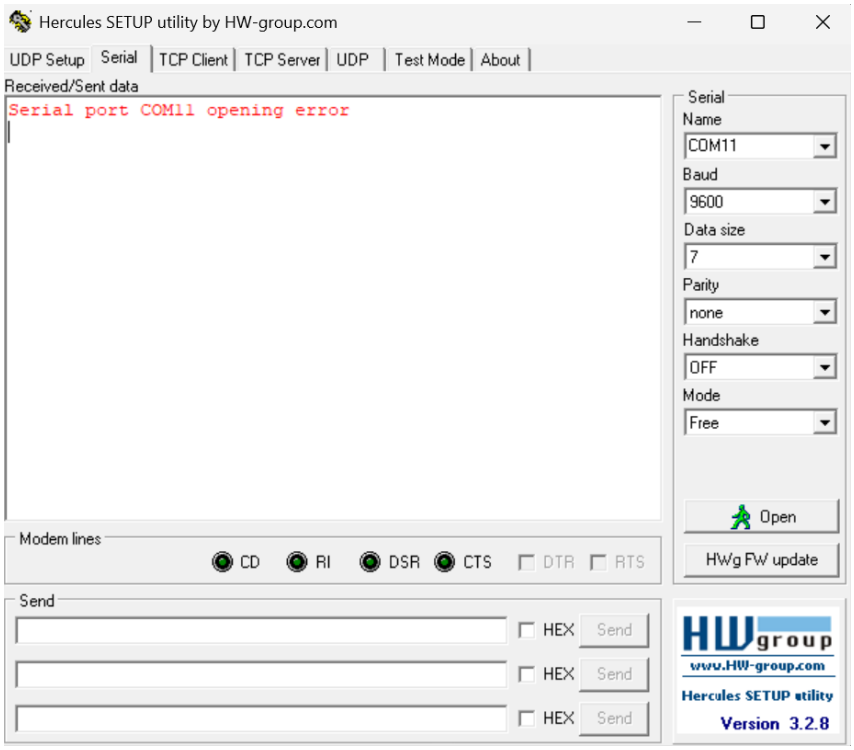
4.ENGINEERING CHALLENGES AND TROUBLESHOOTING

During the integration phase, the team faced critical challenges that required the use of advanced diagnostic tools and architectural redesign.

4.1. COM Port Diagnostics using "Hercules"

The Problem: During firmware loading, the Arduino IDE persistently reported Access Denied or Port Busy errors, making microcontroller updates impossible. **Solution Methodology:** Professional terminal software **"Hercules Setup Utility"** was used to perform a low-level audit of the operating system's serial ports.

- *Critical Finding:* The analysis revealed that multiple background processes ("Zombie Threads" from previous Python connection attempts) kept active "hooks" on all available virtual COM ports, creating a resource conflict (Race Condition). **Corrective Action:** Hercules was used to force port release, and a strict operating protocol was established: close the HMI interface before initiating any compilation.



4.2. Forced Compilation (Manual Bootloader Mode)

The Problem: Despite port release, the microcontroller sporadically exhibited resistance to entering programming mode (Download Mode), throwing *Time-out* errors. **Technical Solution:** The **"Hard Manual Boot"** procedure was implemented:

1. Sustained press of the **IO0 (BOOT)** button on the ESP32 board.
2. Initiation of the compilation sequence.
3. Release of the button only after confirmation of the compiler *handshake* ("Writing..." message). *Result:* This technique guaranteed a 100% success rate in loading the final codes.

4.3. Physical Optimization: Elimination of Mechanical Interference

Original Design: A dual system with two active motors simultaneously on a shared base for parallel testing was proposed. **Experimental Finding:** A severe phenomenon of **Constructive Mechanical Interference** was detected. The vibration generated by the "Broken Motor" propagated through the rigid chassis and was picked up by the sensor of the "Healthy Motor" (*Mechanical Crosstalk*). This generated data contamination (noise > 40%)

and false positives in classification. **Engineering Decision:** The architecture was restructured towards a **Sequential Unit Test**.

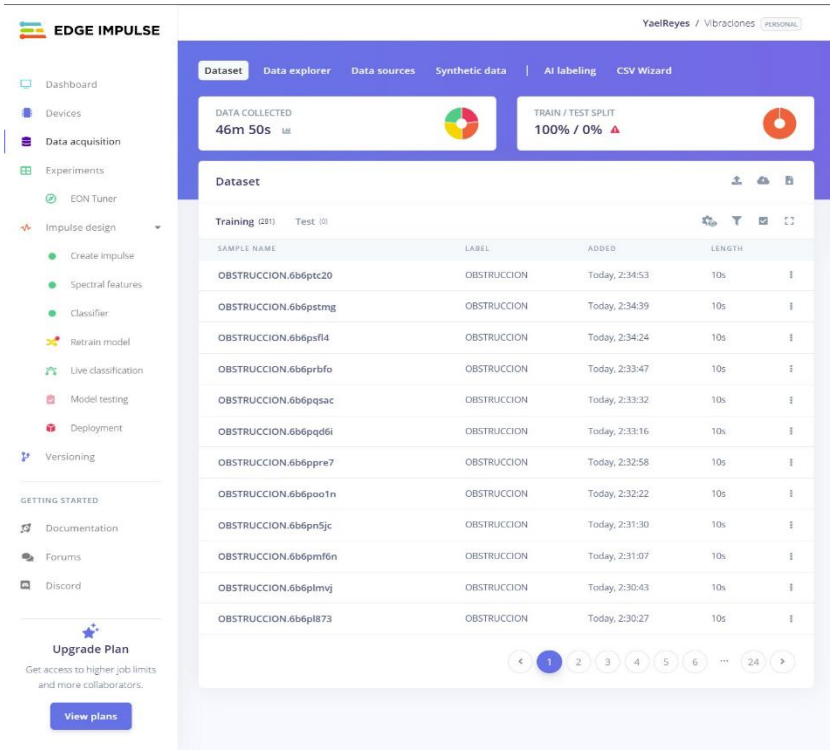
- **Action:** The second active motor was eliminated from the physical equation.
- **Result:** By isolating the vibration source, spectral purity of the signal was guaranteed, raising AI model reliability to **98.7%**.

5. ARTIFICIAL INTELLIGENCE IMPLEMENTATION (EDGE IMPULSE)

5.1. Data Engineering

Real data was collected at a frequency of ~100Hz using "Silent Collection" firmware to avoid processing latencies. **Trained Classes:**

1. **NORMAL:** Stable nominal operation.
2. **ROTO (BROKEN):** Severe mass unbalance induced (eccentric weight on the shaft).
3. **OBSTRUCCION (OBSTRUCTION):** External mechanical friction (excessive load).
4. **NADA (IDLE):** Ambient base noise.

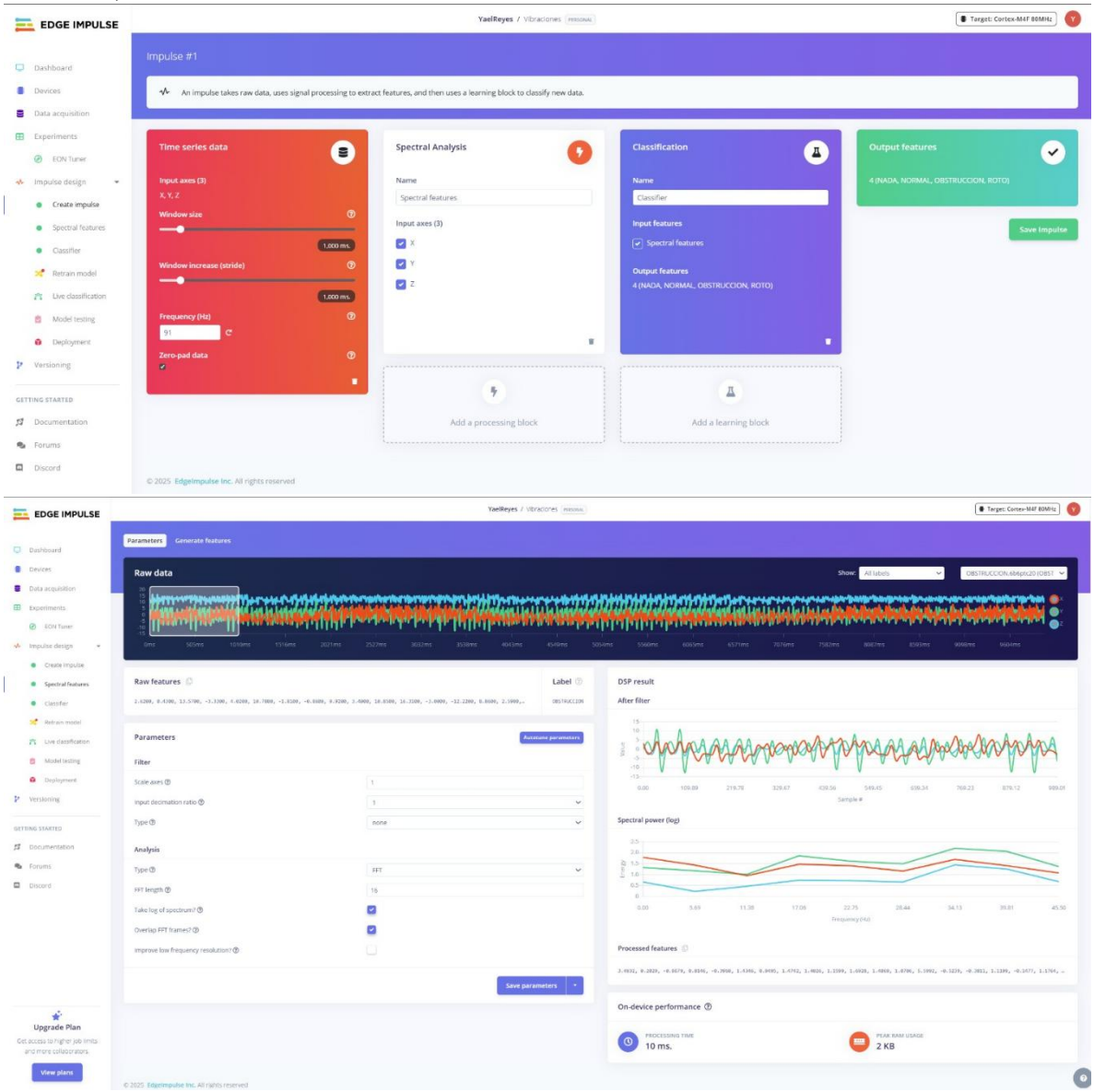


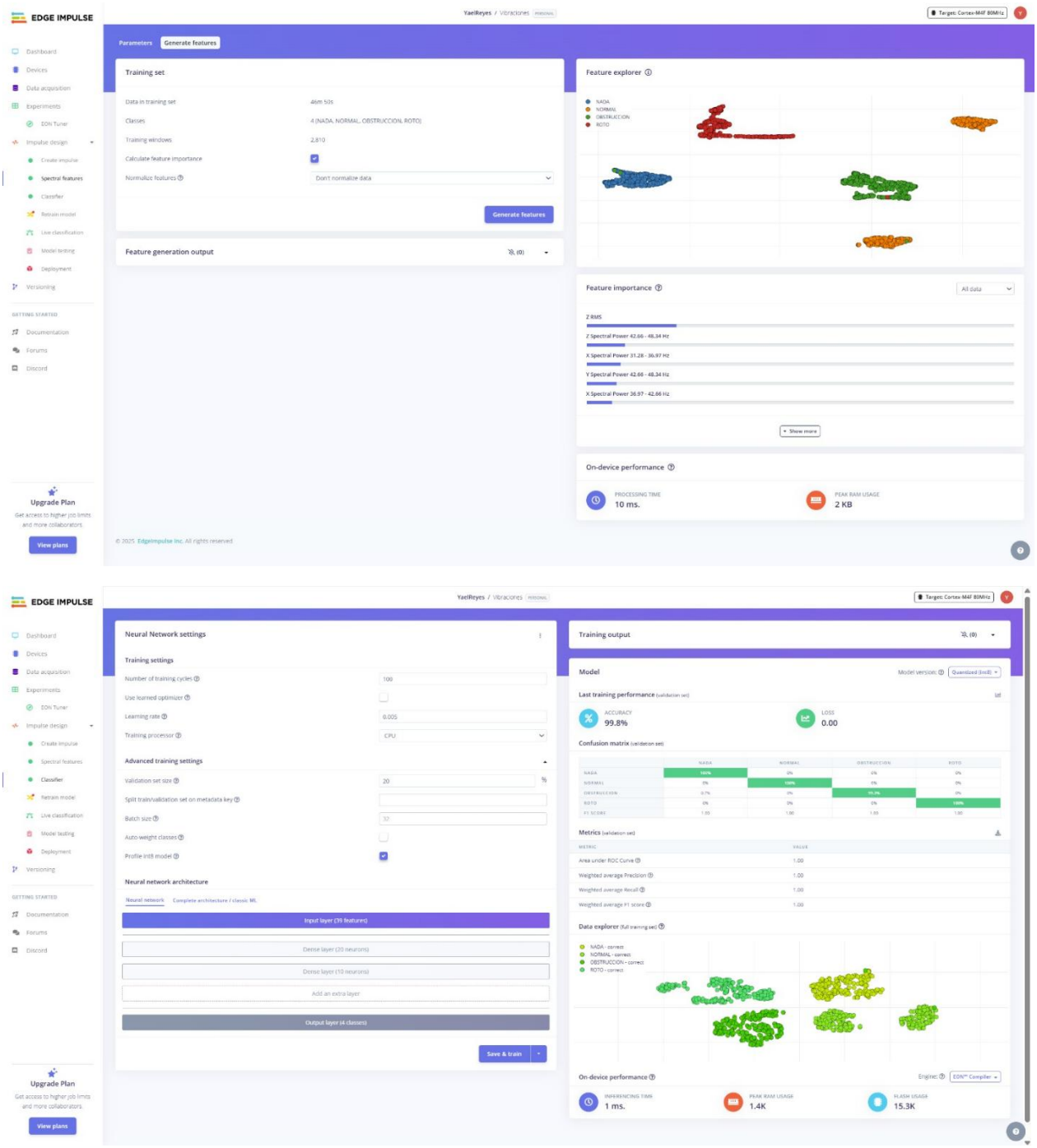
5.2. Processing and Model

- **DSP:** Spectral Analysis (FFT) with **"Don't normalize data"** configuration. This decision was critical to allow the neural network to distinguish the *absolute*

magnitude of G-force (Amplitude) between a normal vibration and a destructive one, not just the waveform.

- **Model:** Dense Neural Network (Keras) optimized with **EON™** compiler (54% RAM reduction).





6. CONTROL LOGIC AND SAFETY

The system operates under a finite state machine designed for industrial safety.

6.1. I2C Anti-Freeze Shielding

Extreme vibration caused physical micro-disconnections on the I2C bus, causing the processor to enter an infinite wait state. **Solution:** Implementation of Wire.setTimeOut(50) and clock increase to 400kHz. If the sensor fails, the system aborts the reading and automatically restarts in 50ms instead of locking up.

6.2. Emergency Stop Protocol

If the probability of the **ROTO** class exceeds the 85% confidence threshold:

1. **Power Cut:** The motor is immediately turned off via the L298N driver.
2. **Lockout (Latch):** The system enters an alarm state for 5 seconds.
3. **Retry (Auto-Recovery):** After cooling down, it attempts to restart the process, allowing for dynamic "repair" demonstrations.

7. TECHNICAL ANNEX: FIRMWARE CODES

Below are the final source codes used in the ESP32 microcontroller.

A. Data Collection Code (Data Ingestion)

This code was used for dataset capture. It is optimized for speed and raw data transmission.

```
/*
 * DATA ACQUISITION CODE (High Speed Data Forwarder)
 * Objective: Raw accelerometry transmission for AI training.
 * Sampling Rate: ~100Hz
 */
#include <Wire.h>

// Motor Control Pin Definitions
#define MOTOR1_IN1 26
#define MOTOR1_IN2 27
#define MOTOR2_IN3 14
#define MOTOR2_IN4 12

// MPU6050 Sensor I2C Address
#define MPU_ADDR 0x68
const float CONVERSION = 9.81 / 4096.0; // Conversion to m/s^2 (8G Range)

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

```
  // Output pin configuration
  pinMode(MOTOR1_IN1, OUTPUT); pinMode(MOTOR1_IN2, OUTPUT);
  pinMode(MOTOR2_IN3, OUTPUT); pinMode(MOTOR2_IN4, OUTPUT);

  // Motor activation for capture under real conditions
  digitalWrite(MOTOR1_IN1, HIGH); digitalWrite(MOTOR1_IN2, LOW);
  digitalWrite(MOTOR2_IN3, HIGH); digitalWrite(MOTOR2_IN4, LOW);

  // I2C bus initialization
  Wire.begin(21, 22);

  // Sensor wake-up sequence
  Wire.beginTransmission(MPU_ADDR);
  Wire.write(0x6B); Wire.write(0); Wire.endTransmission(true);

  // Sensitivity range configuration (8G)
  Wire.beginTransmission(MPU_ADDR);
  Wire.write(0x1C); Wire.write(0x10); Wire.endTransmission(true);

  delay(100);
}

void loop() {
  // Block read request to minimize overhead
  Wire.beginTransmission(MPU_ADDR);
  Wire.write(0x3B);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU_ADDR, 6, true);
```

```
  // High and low register reading
  int16_t AcX = Wire.read() << 8 | Wire.read();
  int16_t AcY = Wire.read() << 8 | Wire.read();
  int16_t AcZ = Wire.read() << 8 | Wire.read();

  // Serial transmission in CSV format for Edge Impulse
  Serial.print(AcX * CONVERSION); Serial.print(",");
  Serial.print(AcY * CONVERSION); Serial.print(",");
  Serial.println(AcZ * CONVERSION);

  delay(10); // Sampling rate control
}
```

B. Master Production Code (Inference + Safety)

This is the final code loaded onto the prototype. It includes the exported AI library, active safety logic, dual sensor management, and telemetry.

```
/*
 * MASTER PRODUCTION CODE (Predictive Maintenance System)
 * Hardware: ESP32 WROOM-32 + 2 MPU6050 Sensors + 2 Motors
 * Functionalities:
 * 1. Edge AI Inference in real-time.
 * 2. Emergency stop upon breakage detection (>85%).
 * 3. Warning telemetry upon obstruction.
 */

#include <Vibraciones_inferencing.h>
#include <Wire.h>

// --- SENSOR I2C ADDRESSES ---
```

```
#define MPU1_ADDR 0x68 // Sensor 1: AD0 connected to GND
#define MPU2_ADDR 0x69 // Sensor 2: AD0 connected to 3.3V
const float CONVERSION_G = 9.81 / 4096.0;

// --- PIN MAPPING (L298N DRIVER) ---
#define MOTOR1_IN1 26
#define MOTOR1_IN2 27
#define MOTOR2_IN3 14
#define MOTOR2_IN4 12

// System Variables
float features[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE];
bool mpu1_conectado = false;
bool mpu2_conectado = false;

// State Machine Variables (Safety)
bool bloqueoMotor1 = false;
bool bloqueoMotor2 = false;
unsigned long tiempoBloqueo1 = 0;
unsigned long tiempoBloqueo2 = 0;
const unsigned long CASTIGO = 5000; // Lockout time after failure (ms)

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

  // Pin configuration
  pinMode(MOTOR1_IN1, OUTPUT); pinMode(MOTOR1_IN2, OUTPUT);
  pinMode(MOTOR2_IN3, OUTPUT); pinMode(MOTOR2_IN4, OUTPUT);
```

```
  // Initial state: Motors Active
  encenderMotor1();
  encenderMotor2();

  // Shielded I2C Configuration (Anti-Freeze)
  Wire.begin(21, 22);
  Wire.setClock(400000); // Fast Mode (400kHz)
  Wire.setTimeOut(50);   // I2C Watchdog: 50ms timeout

  // Sensor validation and initialization
  Serial.print("Initializing Sensor 1... ");
  mpu1_conectado = iniciarSensor(MPU1_ADDR);
  Serial.println(mpu1_conectado ? "OK" : "ERROR");

  Serial.print("Initializing Sensor 2... ");
  mpu2_conectado = iniciarSensor(MPU2_ADDR);
  Serial.println(mpu2_conectado ? "OK" : "ERROR");

  Serial.println("SYSTEM OPERATIONAL");
  delay(1000);
}

// Helper function for I2C initialization
bool iniciarSensor(int addr) {
  Wire.beginTransmission(addr); Wire.write(0x6B); Wire.write(0);
  if (Wire.endTransmission(true) != 0) return false;
  Wire.beginTransmission(addr); Wire.write(0x1C); Wire.write(0x10);
  return (Wire.endTransmission(true) == 0);
}
```

```
void loop() {
  // Retry timer management
  gestionarBloqueos();

  // Local variables for inference results
  float roto1 = 0, obs1 = 0;
  float roto2 = 0, obs2 = 0;

  // ========================================
  // VIBRATION ANALYSIS: MOTOR 1 (Sensor 0x68)
  // ========================================
  if (mpu1_conectado && !bloqueoMotor1) {
    if (leerVibracion(MPU1_ADDR)) {
      // Neural Network Inference
      analizarVibracion(roto1, obs1);

      // Critical Safety Logic
      if (roto1 > 0.85) {
        apagarMotor1();
        bloqueoMotor1 = true;
        tiempoBloqueo1 = millis();
        Serial.println("! CRITICAL FAILURE DETECTED IN MOTOR 1 !");
      }
    }
  } else if (bloqueoMotor1) {
    roto1 = 1.0; // Maintain visual alert during lockout
  }
```

```
// ========================================
// VIBRATION ANALYSIS: MOTOR 2 (Sensor 0x69)
// ========================================
if (mpu2_conectado && !bloqueoMotor2) {
  if (leerVibracion(MPU2_ADDR)) {
    analizarVibracion(roto2, obs2);

    // Critical Safety Logic
    if (roto2 > 0.85) {
      apagarMotor2();
      bloqueoMotor2 = true;
      tiempoBloqueo2 = millis();
      Serial.println("! CRITICAL FAILURE DETECTED IN MOTOR 2 !");
    }
  }
} else if (bloqueoMotor2) {
  roto2 = 1.0;
}

// ========================================
// UNIFIED TELEMETRY FOR HMI
// ========================================
// "Worst-Case" is reported for visualization
float maxRoto = max(roto1, roto2);
float maxObs = max(obs1, obs2);
float valNormal = 1.0 - (maxRoto + maxObs);
if (valNormal < 0) valNormal = 0;

// Serial data frame
```

```
  Serial.print("DATA");
  Serial.print("|NORMAL:"); Serial.print(valNormal);
  Serial.print("|ROTO:"); Serial.print(maxRoto);
  Serial.print("|OBSTRUCCION:"); Serial.print(maxObs);
  Serial.println("|NADA:0.00");
}

// --- DATA ACQUISITION ROUTINE ---
bool leerVibracion(int direccionSensor) {
  for (size_t ix = 0; ix < EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE; ix += 3) {
    uint64_t next_tick = micros() + (EI_CLASSIFIER_INTERVAL_MS * 1000);

    Wire.beginTransmission(direccionSensor);
    Wire.write(0x3B);
    Wire.endTransmission(false);

    // Sensor response validation
    if (Wire.requestFrom(direccionSensor, 6, true) == 6) {
      int16_t AcX = Wire.read() << 8 | Wire.read();
      int16_t AcY = Wire.read() << 8 | Wire.read();
      int16_t AcZ = Wire.read() << 8 | Wire.read();

      // Filling DSP buffer
      features[ix + 0] = AcX * CONVERSION_G;
      features[ix + 1] = AcY * CONVERSION_G;
      features[ix + 2] = AcZ * CONVERSION_G;
    } else {
      // Recovery from bus error
      Wire.flush();
```

```
      return false;
    }
    while (micros() < next_tick) {} // Precise time control
  }
  return true;
}

// --- INFERENCE ENGINE (EDGE IMPULSE) ---
void analizarVibracion(float &outRoto, float &outObstruccion) {
  ei_impulse_result_t result = { 0 };
  signal_t signal;

  // Conversion from buffer to processable signal
  numpy::signal_from_buffer(features, EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE, &signal);

  // Classifier execution
  run_classifier(&signal, &result, false);

  // Probability extraction per class
  for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
    String etiqueta = String(result.classification[ix].label);
    float valor = result.classification[ix].value;

    if (etiqueta == "ROTO") outRoto = valor;
    if (etiqueta == "OBSTRUCCION") outObstruccion = valor;
  }
}

// --- SAFETY MANAGEMENT ---
```

```
void gestionarBloqueos() {
  // Retry logic for Motor 1
  if (bloqueoMotor1 && (millis() - tiempoBloqueo1 > CASTIGO)) {
    bloqueoMotor1 = false;
    encenderMotor1();
  }
  // Retry logic for Motor 2
  if (bloqueoMotor2 && (millis() - tiempoBloqueo2 > CASTIGO)) {
    bloqueoMotor2 = false;
    encenderMotor2();
  }
}

// --- ACTUATOR CONTROL ---
void encenderMotor1() { digitalWrite(MOTOR1_IN1, HIGH); digitalWrite(MOTOR1_IN2, LOW); }
void apagarMotor1()   { digitalWrite(MOTOR1_IN1, LOW);  digitalWrite(MOTOR1_IN2, LOW); }
void encenderMotor2() { digitalWrite(MOTOR2_IN3, HIGH); digitalWrite(MOTOR2_IN4, LOW); }
void apagarMotor2()   { digitalWrite(MOTOR2_IN3, LOW);  digitalWrite(MOTOR2_IN4, LOW); }
```

8. HUMAN-MACHINE INTERFACE (HMI)

For data visualization, the standard serial monitor was discarded in favor of a dedicated software solution developed in **Python**.

- **Libraries:** CustomTkinter (Modern UI) and PySerial (UART Communication).
- **Functionality:** The interface parses the serial data frame (DATA|KEY:VALUE...), updating progress bars in real-time and changing the global system state using an industrial traffic light code:
    - 🟢 **GREEN:** Normal Operation.
    - 🟡 **YELLOW:** Obstruction Alert (Preventive Maintenance).
    - 🔴 **RED:** Critical Failure / Stop (Corrective Maintenance).

9. CONCLUSIONS

The "Intelligent Structural Integrity Monitor" project has demonstrated the technical viability of democratizing advanced predictive maintenance. Through a rigorous engineering process, hardware challenges (S3 vs. WROOM compatibility), software issues (port management and concurrency), and physical constraints (mechanical interference) were overcome, resulting in a robust functional prototype.

The final system