INDUSTRY INTERNSHIP REPORT

ON

" AI ENABLED TEMPERATURE MONITORING AND OVERHEAT PREVENTION"

AT

MITSUBISHI ELECTRIC FACTORY AUTOMATION, AUTHORIZED TRAINING CENTRE, MIET JAMMU

AN INDUSTRY INTERNSHIP REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF DEGREE OF

BACHELOR OF TECHNOLOGY
In
Electronics and Communication Engineering

SUBMITTED BY
Tushar Singh
2021a3r006

AUTONOMOUS

SUBMITTED TO
Department of Electronics and Communication Engineering
Model Institute of Engineering and Technology (Autonomous)
Jammu, India
2025

CANDIDATES’ DECLARATION

I, **Tushar Singh**, **2021a3r006**, hereby declare that the work which is being presented in the Final Year Project Report entitled, “**AI ENABLED TEMPERATURE MONITORING AND OVERHEAT PREVENTION**”, in partial fulfillment of the requirement for the award of the degree of B.Tech. (Electronics and Communication Engineering) and submitted in the Department of Electronics and Communication Engineering, Model Institute of Engineering and Technology (Autonomous), Jammu is an authentic record of my own work carried out by me under the supervision and mentorship of **Dr. Parveen Kumar** (Assistant Professor, Department of Electrical Engineering, MIET Jammu). The matter presented in this report has not been submitted in this or any other University / Institute for the award of B.Tech. Degree.

Signature of the Student *Dated*:

Tushar Singh
2021a3r006

Department of Electronics and Communication Engineering
Model Institute of Engineering and Technology (Autonomous)
Kot Bhalwal, Jammu, India
(NAAC "A" Grade Accredited)

Ref. No.: MIET/ECE/2025/ PII **Date:**

CERTIFICATE

Certified that this final year project report entitled "**AI ENABLED TEMPERATURE MONITORING AND OVERHEAT PREVENTION**" is the bonafide work of **Tushar Singh (2021a3r006)** of 8th Semester, **Electronics and Communication Engineering**, **Model Institute of Engineering and Technology (Autonomous), Jammu**, who carried out the project work under my supervision during the academic session Feb 2025 – June 2025.

Dr Parveen Kumar
Mentor-Internal Supervisor
Assistant Professor
Department of Electrical Engineering, MIET

This is to certify that the above statement is correct to the best of my knowledge.

Dr. Surbhi Sharma
Assistant Professor
Head of Department
Department of Electronics and Communication Engineering, MIET

ACKNOWLEDGEMENT

Industry internship is an important aspect in the field of engineering, where contributions are made by many individuals and organizations. The present shape of this work has come forth after support and input from different spheres.

I am especially grateful to my project supervisor, Dr. Parveen Kumar, for his dedicated guidance, timely feedback, and continuous support at every step of this journey.

I wish to extend my heartfelt appreciation to Dr. Satyendra Singh for his academic support and valuable input, and to Mr. Karan Khajuria and Mr. Bhanu Pratap for their technical mentorship, encouragement, and valuable assistance during the development of this work.

I am deeply grateful to Dr. Surbhi Sharma, Head of Department (Electronics and Communication Engineering), for her constant motivation, insightful feedback, and unwavering academic support throughout this project.

I would also like to express my sincere gratitude to Prof. Ankur Gupta, Director, MIET (Autonomous), Jammu, for his visionary leadership and for fostering an environment of academic excellence and innovation.

Lastly, I express my gratitude to the Model Institute of Engineering and Technology (Autonomous), Jammu, for giving me the opportunity to work on this final year project as part of my B.Tech (Electronics and Communication Engineering) program.

Tushar Singh
2021a3r006

SELF EVALUATION

An internship is very important for a student at higher education levels. It is an opportunity to work at a firm or an organization for a fixed period. An internship gives practical skills, workplace experience and greater knowledge of industries or those sectors related to a student's future career. During my internship in automation with (PLCs), I gained valuable experience and developed important skills. I demonstrated proficiency in PLC programming by creating efficient and error-free code for different automation tasks. I effectively utilized functions, loops and conditional statements to control and monitor industrial processes. I developed a systematic approach to identifying and resolving issues in PLC systems. I effectively used debugging tools and techniques to diagnose problems, leading to quick resolutions and minimal downtime. I actively participated in team projects and effectively communicated with colleagues and superiors. I demonstrated the ability to explain complex concepts clearly and concisely, fostering collaboration and understanding. Throughout the internship, I actively sought opportunities to expand my knowledge and enhance my skills in PLC automation. I stayed updated with the latest advancements in the field and implemented new techniques whenever appropriate. Overall, I believe I made significant progress during my internship in automation with PLC. I demonstrated technical competence, problem- solving abilities, effective communication, and a strong work ethic. I am confident that the experience gained during this internship has prepared me well for future endeavors in the automation industry.

ABSTRACT

The project **"AI Enabled Temperature Monitoring and Overheat Prevention"** focuses on the development of an intelligent, automated system that not only tracks temperature in real time but also actively prevents overheating in industrial and sensitive electronic environments. Leveraging the reliability of programmable logic controllers (PLCs) alongside an array of industrial-grade and auxiliary sensors, the system continuously gathers and processes thermal and electrical data to maintain safe operating conditions. The data is fed into a custom AI-driven decision layer, which performs predictive analysis, identifies emerging risks before they escalate, and triggers preventive actions such as activating cooling fans, sounding alarms, or shutting down equipment when thresholds are approached. A user-friendly Flask-based web dashboard serves as the system's interface, providing operators with real-time visualization, historical trends, remote control capabilities, and alerts, thereby improving accessibility and operational safety. By merging traditional industrial automation with AI, the solution adapts dynamically to changing conditions, optimizes response strategies, and reduces reliance on manual monitoring. This project ultimately showcases a cost-effective, scalable, and future-ready approach to mitigating overheating risks, minimizing downtime, and enhancing the reliability and safety of automated setups across multiple domains.

Contents

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS USED

AI	Artificial Intelligence
DHT	Digital Humidity and Temperature Sensor
ESP	Embedded Serial Peripheral
HMI	Human Machine Interface
IR	Infrared
IoT	Internet of Things
ML	Machine Learning
NC	Normally Closed
NO	Normally Open
PLC	Programmable Logic Controller
PT100	Platinum Resistance Temperature Detector (RTD)
RTD	Resistance Temperature Detector
RPM	Revolutions Per Minute
SCADA	Supervisory Control and Data Acquisition
SMPS	Switched Mode Power Supply

Chapter 1

INTRODUCTION

1.1 Overview of the Project

In the evolving world of automation and smart systems, temperature monitoring has become an essential safety feature, particularly in industrial environments, server systems, embedded electronics, and manufacturing units. Overheating is a leading cause of system malfunctions, component failures, and even fire hazards. Traditional temperature control systems often rely on static thresholds and reactive measures, which may not respond effectively in dynamic or rapidly changing scenarios.

The fusion of Artificial Intelligence (AI) with Programmable Logic Controllers (PLCs) and sensor-based systems offers a smarter alternative. This project, titled "AI Enabled Temperature Monitoring and Overheat Prevention," presents an integrated solution combining real-time sensing, intelligent decision-making, and automated responses to prevent overheating. The system continuously tracks temperature through sensors and processes the data using AI-enabled logic. It issues alerts and controls outputs such as fans or cut-off mechanisms to maintain safe temperature limits.

To enhance accessibility, a custom-built web interface is developed for visualizing the live temperature data and system status remotely. This not only supports real-time monitoring but also allows users to make decisions from any location. The project contributes to safer operations, improved equipment longevity, and reduced risks in environments where thermal regulation is critical.

1.2 Problem Statement

In many traditional control systems, overheating-related failures occur due to the inability of static logic to adapt to changing conditions. PLC-based systems, although highly reliable in automation, usually lack predictive intelligence. They operate on predefined logic, without any learning or pattern recognition. In systems where temperature variations are not linear or predictable, this becomes a major limitation.
Moreover, many existing solutions do not include real-time remote monitoring or alerting features. Users are often unaware of rising temperatures until it’s too late. As industries shift toward Industry 4.0 and smart manufacturing, there is a growing demand for systems that are not just reactive, but proactively intelligent.

This project aims to solve the following key challenges:

Table 1.1: Key Problems Addressed by the Project

S. No.	Challenge	Impact
1	Lack of intelligent prediction for temperature trends	Causes delayed responses to overheating risks
2	Static threshold-based control logic in PLCs	Inflexible in dynamic or unpredictable environments
3	Absence of a user-friendly interface for real-time monitoring	Limits remote access and proactive decision-making
4	Lack of integrated alarm and automatic action triggers	Increases downtime and potential hardware damage

1.3 Objective of the Project

The primary objective of the project "AI Enabled Temperature Monitoring and Overheat Prevention" is to design and implement a smart and efficient thermal management system that integrates Artificial Intelligence with traditional automation technologies such as PLCs. The aim is to prevent system damage caused by overheating by enabling proactive and real-time temperature control. The specific objectives include:

- To develop a temperature sensing and monitoring system using industrial-grade sensors connected to a PLC.
- To integrate AI-based logic that can detect abnormal temperature trends and initiate timely actions.
- To design a responsive web interface for real-time temperature visualization, alert generation, and user interaction.
- To automate overheating responses like activating cooling devices or issuing alerts, ensuring uninterrupted system operations.
- To reduce downtime and improve safety, particularly in critical environments such as industrial panels, server rooms, or control cabinets.

The project is designed to be cost-effective, scalable, and easily deployable across a range of applications where temperature regulation is essential.

1.4 Scope and application

The project has a wide range of applications and is scalable across multiple domains that demand accurate and timely temperature management. By combining AI with industrial automation, the system ensures predictive analytics, data-driven decision-making, and user-friendly control through a centralized interface. The system supports real-time data acquisition, AI-driven anomaly detection, and automatic response mechanisms, and can be integrated with various industrial automation platforms, particularly where PLCs are already in use. It also includes a web-based monitoring system, allowing users to access system data and logs from remote locations, and its modular architecture enables future upgrades such as integrating more environmental sensors or cloud-based analytics. The solution finds applications in industrial control panels to prevent damage to sensitive electrical components due to overheating, data centers and server rooms to monitor thermal conditions and prevent failures, automated manufacturing lines to monitor heat-prone machines and take preemptive actions, smart homes and buildings to enhance HVAC systems with intelligent thermal control, and renewable energy systems to maintain temperature-sensitive battery packs or inverters in solar and wind setups. Overall, this project is a step toward the implementation of Industry 4.0, promoting smarter, safer, and more efficient industrial and environmental systems.

Chapter 2

LITERATURE REVIEW

Industrial automation is increasingly shifting towards integrated monitoring systems that combine real-time environmental sensing with machine health diagnostics. Sabo et al. [1] developed a smart IoT-based broiler room controller that monitors and regulates temperature, humidity, and lighting in poultry houses using an ESP32 microcontroller. Their system features a web-based interface for remote control and data visualization, highlighting the advantages of embedded IoT solutions in environmental management. This approach demonstrates how cost-effective microcontroller platforms can be used for real-time monitoring and control, a concept that can be extrapolated to industrial scenarios where machine environment monitoring is essential for operational safety.

The integration of wireless sensor networks (WSNs) with advanced technologies such as artificial intelligence and blockchain has further expanded the scope of monitoring systems. Dong and Feng [2] proposed a wireless temperature measurement network for electrical power engineering that employs AI models for fault prediction while using blockchain to secure data integrity. Their system triggers preventive actions when monitored parameters exceed safe thresholds, effectively reducing the likelihood of thermal failures in critical infrastructure. This study emphasizes the importance of predictive analytics and secure data management in modern industrial settings, aligning with the objectives of real-time health monitoring in high-risk environments.

In the healthcare sector, similar IoT-driven frameworks are used for cold chain management, particularly in the storage and transportation of vaccines. Jiang et al. [3] designed an IoT-enabled framework that continuously monitors the temperature of vaccine storage systems, raising alerts when anomalies occur. The system helps maintain regulatory compliance and prevents spoilage by ensuring temperature stability throughout the supply chain. This research highlights the applicability of remote temperature monitoring not only for healthcare but also for industrial domains where temperature control is critical to prevent equipment failure.

Machine learning techniques are increasingly utilized for predictive maintenance and thermal health assessment in various industries. Sukkam et al. [4] introduced a model that predicts a battery’s thermal health factor using real-world driving data from electric vehicles. Their machine learning model analyzes current, voltage, and environmental data to forecast overheating risks, enabling proactive maintenance interventions. This approach can be adapted for motor systems in industrial applications, where thermal monitoring is vital to avoid unexpected downtimes and ensure system longevity.

The role of artificial intelligence in optimizing thermal management systems extends beyond predictive maintenance. Ghahramani et al. [5] examined AI-driven HVAC systems that balance occupant comfort with energy efficiency. Their framework incorporates environmental inputs like temperature, humidity, and occupancy data, allowing AI agents to adjust system operations dynamically. Such adaptive control systems reduce energy costs while maintaining safe thermal conditions, principles that can be mirrored in industrial motor health management where dynamic environmental factors affect operational safety.

Remote monitoring interfaces have become a central feature of modern temperature management systems. Rahaman and Iqbal [6] developed an IoT thermostat using openHAB and MQTT, providing users with a web dashboard to monitor and control room temperatures remotely. This approach leverages cloud communication protocols to extend system control beyond local boundaries, offering similar advantages in industrial monitoring applications where decentralized access to motor and environmental data enhances decision-making capabilities.

IoT cloud platforms using flow-based programming tools have further simplified remote monitoring and control. Rattanapoka et al. [7] implemented an MQTT-based cloud platform with Node-RED for visual programming of IoT workflows. This system allows real-time data collection, historical trend analysis, and event-driven automation, streamlining complex monitoring tasks. In industrial settings, similar platforms can visualize sensor data, trigger alerts, and automate responses to abnormal conditions, making remote management more efficient and user-friendly.

Despite the advancement of IoT technologies, traditional PLC-based monitoring systems remain foundational in industrial environments due to their robustness and deterministic control capabilities. Ding and Li [8] developed a PLC-based temperature monitoring system that uses thermocouples and analog input modules to regulate heating elements through feedback loops. Their system, implemented in manufacturing processes such as plastics molding and food production, ensures consistent temperature control and integrates with SCADA for visualization and operator intervention. The hardware-software integration facilitates early fault detection and automated shutdowns, reducing the risk of thermal accidents.

Sensor networks for environmental and industrial monitoring have also been extensively reviewed in the context of agriculture and food industries. Ruiz-Garcia et al. [9] provided a comprehensive overview of wireless sensor technologies like RFID, ZigBee, and Bluetooth, highlighting their applications in temperature, humidity, and soil condition monitoring. These systems support proactive responses in precision farming and cold storage management, offering valuable insights for industrial adaptations where motor health and environmental conditions must be monitored together to maintain operational safety and efficiency.

Collectively, these studies underline the importance of integrating AI, IoT, PLC, and machine learning techniques for comprehensive monitoring of both machine health and environmental parameters. Hybrid systems that leverage PLCs for critical safety control alongside IoT microcontrollers for secondary monitoring enable predictive maintenance strategies while maintaining the reliability of industrial operations. The convergence of these technologies allows for real-time data visualization, anomaly detection, and condition-based control, ultimately contributing to safer, more efficient, and smarter industrial ecosystems.

Chapter 3

AUTOMATION AND SYSTEM COMPONENTS

3.1 Introduction to Automation

Automation is the process of utilizing control systems, software, and hardware to perform tasks with minimal or no human intervention. It is a cornerstone of modern engineering, enabling increased efficiency, accuracy, and consistency across a wide range of applications—from manufacturing and industrial processes to infrastructure management and consumer technologies. By automating repetitive or complex tasks, systems can operate continuously, respond faster to changing conditions, and significantly reduce the risk of human error.

At its core, an automated system typically comprises sensors for data acquisition, controllers (such as PLCs or microcontrollers) to process inputs and execute logic, and actuators or devices that carry out the desired actions. These components work in a closed feedback loop, enabling real-time decision-making and control. With the integration of advanced technologies like IoT and AI, automation has evolved beyond static rule-based systems to intelligent frameworks capable of learning, adapting, and optimizing operations over time.

In engineering projects, automation is more than just an efficiency tool—it's a strategic framework. It ensures scalability, repeatability, and long-term sustainability by reducing manual dependency and enabling predictive control. Whether in industrial plants or smart infrastructure, automation lays the foundation for intelligent, responsive systems that can meet the demands of modern operations.

3.2 Role of PLC in System Control

A central component of the automated architecture is the Programmable Logic Controller (PLC), which governs real-time operations such as pump control based on the input received from various sensors. In this project, a Mitsubishi iQ-R PLC (Figure 3.1) has been deployed due to its industrial-grade reliability and compatibility with a wide range of input/output devices. The PLC continuously monitors water level data from sensors and executes predefined logic to regulate pump activity, thereby preventing overflow and underutilization. Its deterministic response time and low latency make it ideal for time-sensitive applications, ensuring immediate action in critical scenarios. Moreover, the use of ladder logic allows for easy customization and future scalability.

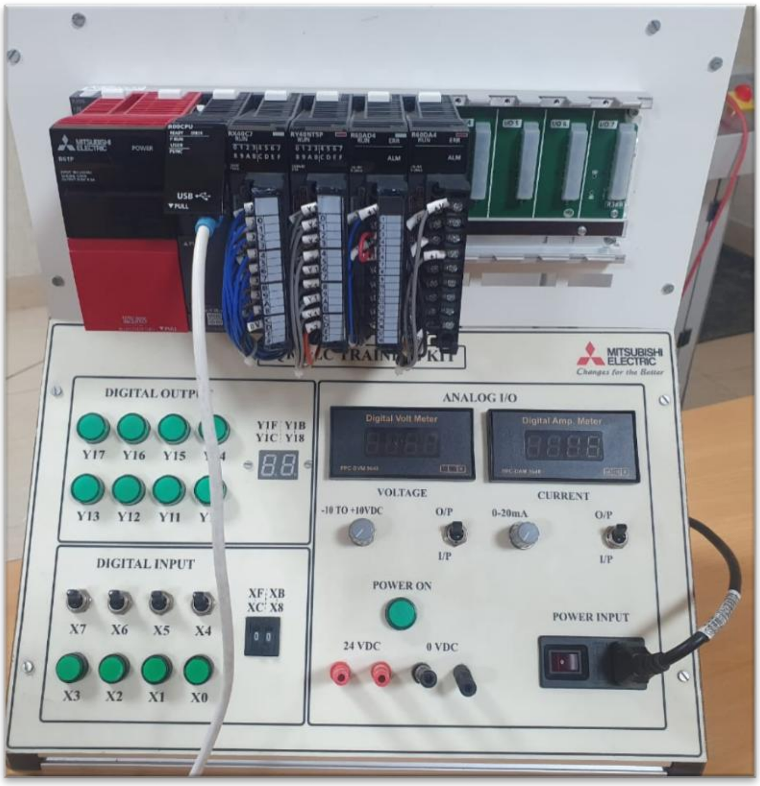


Figure 3.1: Mitsubishi iQ-R PLC used for motor control and safety automation

3.3 Sensor Integration for Real-Time Monitoring

In any automated system, the accuracy, responsiveness, and reliability of data acquisition heavily depend on the types and deployment of sensors. In this project, a combination of industrial-grade and microcontroller-compatible sensors has been integrated to continuously monitor both motor-specific and environmental parameters. The PLC handles critical data acquisition using sensors directly connected to analog input modules.

A PT100 RTD sensor, coupled with a 4–20 mA transmitter, is used for monitoring motor surface temperature (Figure 3.2). This configuration offers industrial-level precision and immunity to electrical noise, making it highly suitable for real-time thermal analysis in motor operations. Similarly, the motor's voltage is measured using a TX-205 DC voltage transducer, which provides a 0–10 V analog output signal that is fed directly into the PLC’s AD4 module.

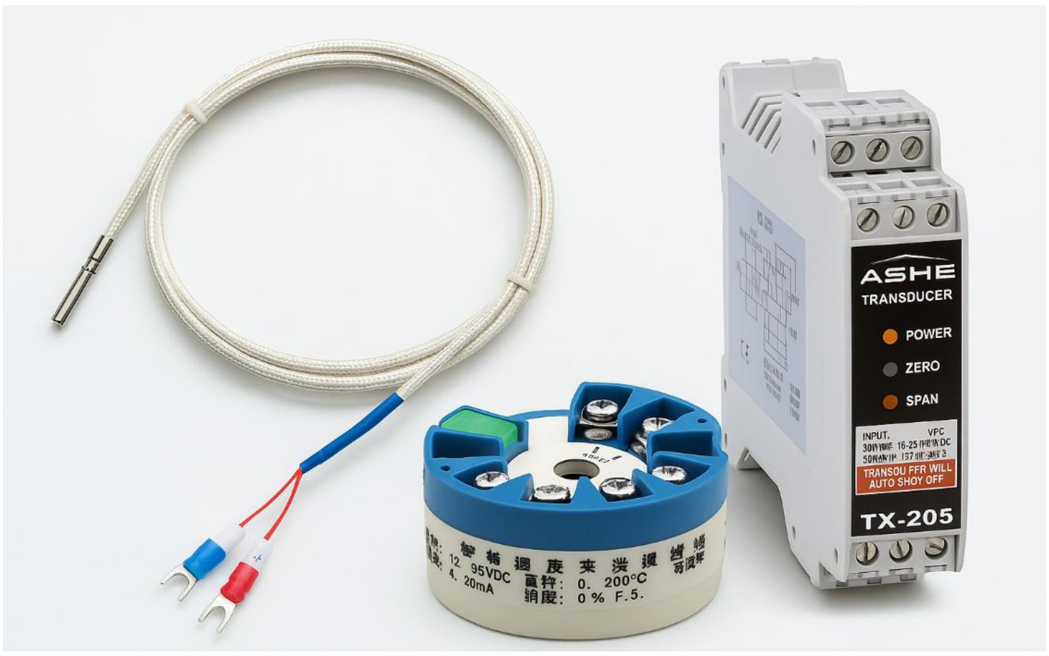


Figure 3.2: PT100 RTD and TX-205 Voltage Sensors

These analog signals are continuously read by the PLC, enabling deterministic control actions based on thermal and electrical conditions. By handling critical inputs, the PLC ensures that overheating or voltage anomalies can trigger immediate and reliable protective measures.

On the non-critical side, an Arduino microcontroller is used to interface with sensors that monitor auxiliary or environmental parameters. The DHT22 sensor captures atmospheric temperature and humidity, offering digital output and stable readings for ambient conditions. The ACS712 current sensor measures the motor's electrical load using the Hall effect principle, providing an analog voltage proportional to the current. The motor’s speed is tracked using an LM393 IR-based sensor, which detects shaft revolutions and generates pulses that Arduino counts via interrupt functionality to calculate RPM (Figure 3.3).

These Arduino-connected sensors are not used for primary control but transmit supplementary data to the main server for logging, visualization, and predictive analysis. This layered integration — combining PLC-connected critical sensors with flexible microcontroller modules — ensures a full picture of system health while maintaining separation between safety-critical and secondary tasks, creating a robust foundation for future scalability.

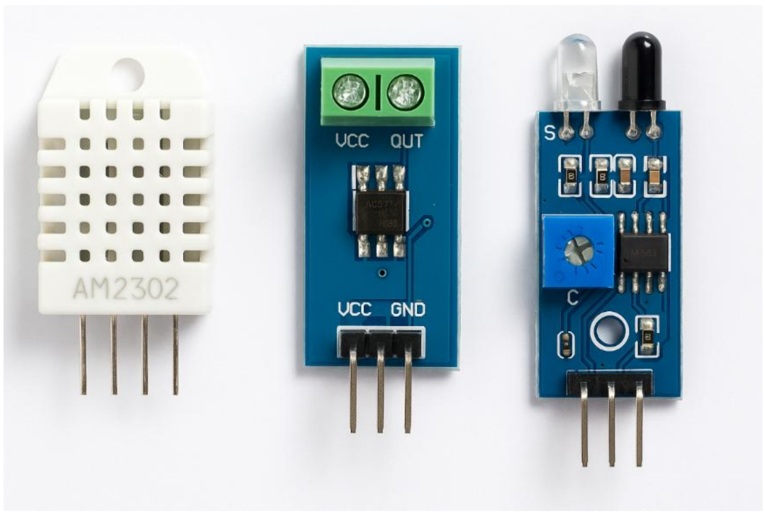


Figure 3.3: Arduino-based sensors

3.4 Communication Modules and Remote Access

Effective communication among the various subsystems in an automated setup is essential to maintain real-time awareness and control. In this system, a hybrid communication strategy is employed to link the PLC, Arduino, ESP32, and the central Python-based dashboard. The PLC, which handles core control logic, communicates with the main server through HTTP-based requests enabled via its built-in web server or using Mitsubishi's MC Protocol. This allows the Python application to retrieve sensor readings, monitor outputs, and even issue commands directly to the PLC.

For non-critical parameters gathered through the Arduino, data transfer is facilitated by the ESP32 module (Figure 3.4). Here, Arduino collects sensor data and sends it over a serial interface to the ESP32, which then forwards it to the Python server using HTTP POST requests. This ensures that real-time environmental and motor auxiliary data is made available to the server without disrupting the PLC's time-sensitive operations.

Figure 3.4: ESP32 module enabling GSM/Wi-Fi communication for remote access and alerts

Furthermore, the ESP32 supports both Wi-Fi and web server functionalities, enabling not just data transfer but also the possibility of expanding to remote cloud-based platforms in future iterations. While the current implementation focuses on a LAN-based dashboard, the underlying infrastructure is capable of supporting remote diagnostics and alerting mechanisms through network access. This modular approach to communication preserves system integrity while allowing extensibility. Importantly, this separation between PLC and ESP-based communication ensures that non-critical sensor nodes cannot interfere with critical machine operations, adhering to best practices in industrial automation. Overall, the communication layer forms the backbone of the system, integrating diverse hardware elements into a cohesive, responsive network.

3.5 Software and AI Integration

On the software front, the system features a web-based dashboard that provides a user-friendly interface to visualize real-time and historical water usage data. The PLC logic is designed and configured using GX Works software (Figure 3.5), which simplifies ladder programming and makes it easier to integrate hardware modules into the overall system architecture. This programming environment ensures that control logic remains transparent and easily modifiable, supporting long-term scalability

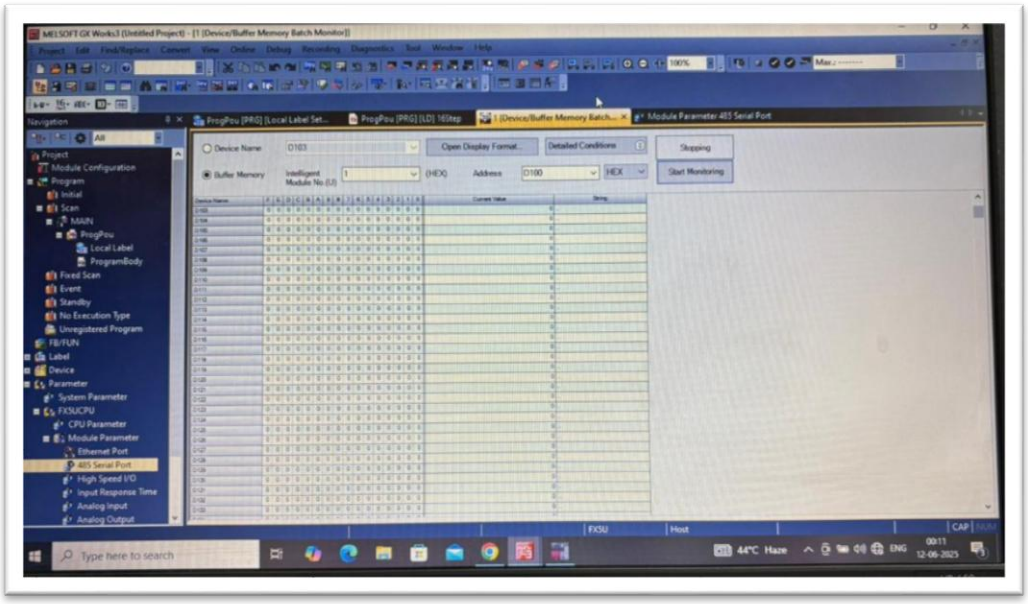


Figure 3.5: GX Works software interface used for PLC programming and system configuration

Beyond control logic, the solution integrates Artificial Intelligence algorithms for anomaly detection and consumption forecasting. By analysing historical usage patterns, the AI module predicts future demand, identifies inefficiencies, and suggests corrective actions. This enables data-driven decision-making and supports long-term resource optimization. The system's modular design allows these AI tools to operate seamlessly alongside the PLC and sensor network, creating a cohesive, intelligent water management platform.

Chapter 4
SYSTEM IMPLEMENTATION

This chapter presents the practical implementation of the AI-Enabled Industrial Motor Health and Environment Monitoring System. It explains how the hardware components were integrated, how the software modules were developed, and how the overall system was configured and deployed. The implementation brings together the hybrid control architecture involving Mitsubishi iQ-R Series PLC, Arduino with ESP32, and a Python-based AI-enabled monitoring dashboard. The chapter also covers the wiring setup, communication flow between devices, and integration of AI models for predictive maintenance. The aim is to show how each component contributes to the seamless operation of a smart, real-time industrial monitoring solution.

4.1 System Overview and Deployment Environment

The system was deployed on a test bench configured to replicate a typical industrial environment with focus on motor health monitoring. The setup included a Mitsubishi iQ-R Series PLC, Arduino Uno, ESP32 microcontroller, various analog and digital sensors, and a local server running Python-based modules. All components were powered using isolated power supplies, ensuring electrical separation between PLC logic, low-voltage microcontroller systems, and actuators. The PT100 RTD sensor with a 4–20 mA transmitter was connected to the analog module (R60AD4) of the PLC to monitor motor temperature, while a TX-205 DC voltage transducer supplied the motor voltage signal to the same analog input module.

Arduino handled environmental and auxiliary parameters, including atmospheric temperature and humidity via the DHT22, motor current through the ACS712, and motor RPM using an LM393 IR sensor. These readings were sent to the ESP32 over serial communication, which then forwarded the data to the local server using HTTP POST requests. In parallel, the PLC periodically shared its sensor data with the Python server via HTTP GET requests or Mitsubishi's MC Protocol, depending on configuration. The server hosted a Flask-based dashboard to present live values, trigger alarms, and run the AI-driven predictive maintenance module. This hybrid architecture — illustrated in Figure 4.1 — balanced reliability for safety-critical functions with flexibility for advanced diagnostics and data analysis.

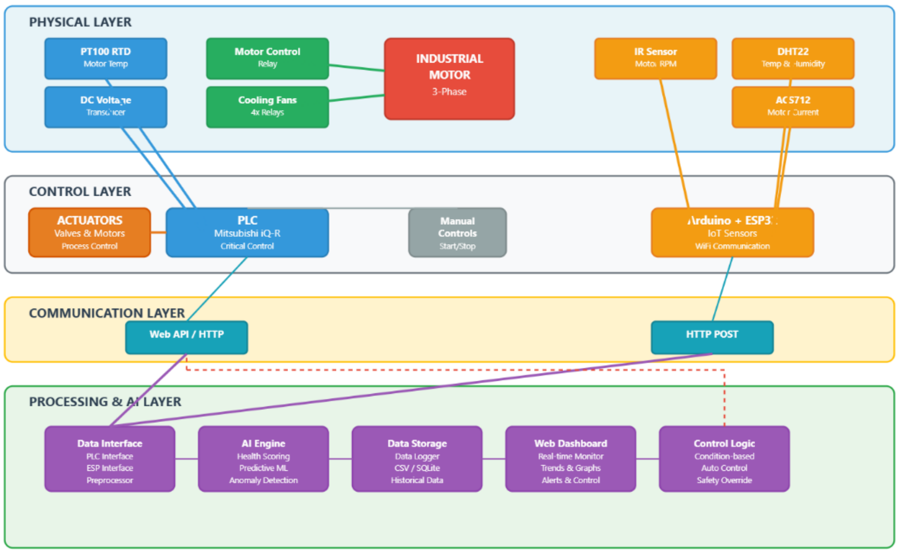


Figure 4.1: AI Enabled Industrial Motor Health & Environmental Monitoring System

4.2 PLC Hardware Wiring and Configuration

The PLC system forms the core of the motor control and monitoring unit, with the Mitsubishi iQ-R Series PLC chosen for its modularity, deterministic response, and industrial-grade I/O handling. The configuration included the R61P power supply module, CPU module, and the R60AD4 analog input module. Two key sensors fed into this analog module: the PT100 RTD (via a 4–20 mA transmitter) and the TX-205 voltage transducer (0–10 V), mapped to CH1 and CH2 of the AD4 module respectively. The wiring arrangement for these modules and terminal blocks is shown in Figure 4.2, illustrating the structured layout designed for clear signal flow and safe isolation.

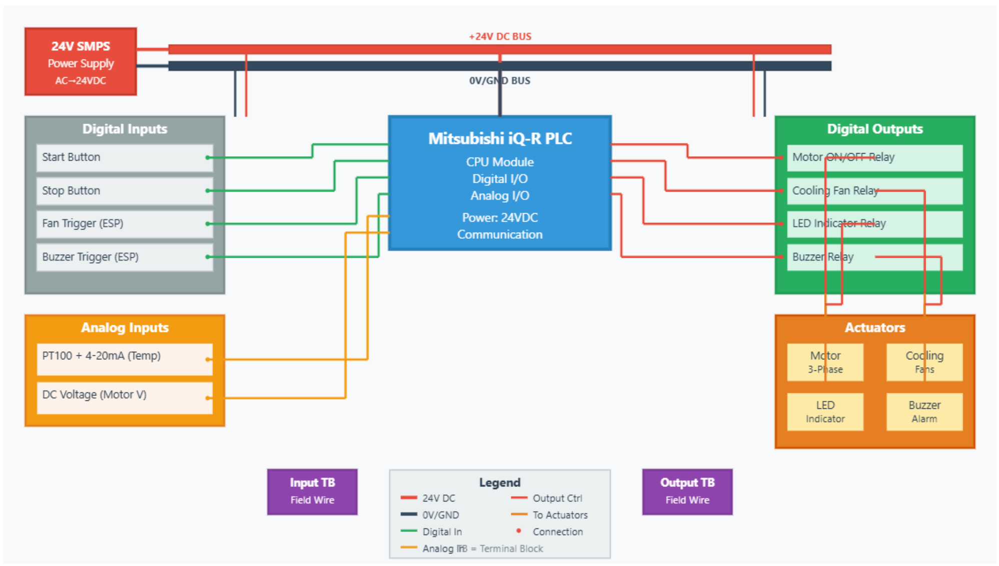


Figure 4.2: Wiring layout of iQ-R PLC modules and terminal blocks

The PLC’s digital inputs were connected to the Start and Stop push buttons (X0 and X1), while its digital outputs controlled all actuators — including the main motor relay (Y12), cooling fans (Y13–Y16), buzzer (Y17), and LED indicators (Y18). Each output drove 24 V relay circuits to safely switch high-load devices. Proper grounding and isolation practices were followed to ensure safe and reliable operation in industrial environments. Ladder logic was programmed in GX Works3, using sensor thresholds for automated actions like turning on cooling or triggering alarms, while extra digital inputs (X2 and X3) were reserved for override commands from the ESP32, allowing AI-based interventions.

Table 4.1: PLC I/O Channel Mapping and Sensor Assignments

Channel/Address	**Component**	**Type**	**Function**
X0	Start Switch	Digital Input	Start motor operation
X1	Stop Switch	Digital Input	Stop motor operation
X2	ESP Cooling Override	Digital Input	Turn on fans from AI module
X3	ESP Buzzer/LED Override	Digital Input	Activate alarm from AI prediction
Y12	Motor Relay	Digital Output	Turn on/off motor
Y13–Y16	Cooling Fans	Digital Output	Control fans (auto/manual)
Y17	Buzzer	Digital Output	Alarm sound on fault
Y18	LED Indicator	Digital Output	Visual warning signal
AD CH1	PT100 RTD (4–20 mA)	Analog Input	Motor temperature
AD CH2	TX-205 (0–10 V)	Analog Input	Motor voltage

4.3 Arduino + ESP32 Setup and Programming

The system's secondary monitoring layer used an Arduino Uno connected to an ESP32, handling non-critical data like ambient temperature, humidity, motor current, and shaft speed (RPM). As shown in Figure 4.3, this ESP32-based setup formed the communication bridge, ensuring supplementary parameters were gathered without overloading the PLC.

The DHT22 measured atmospheric temperature and humidity via a digital I/O pin, while the ACS712 Hall-effect sensor provided analog signals for motor current. Motor speed was tracked by the LM393 IR sensor, which sent pulses to Arduino to compute RPM.

Arduino processed these readings, formatted them into JSON, and sent them to the ESP32 over serial. The ESP32, programmed via Arduino IDE, posted this data to the Python server over Wi-Fi at regular intervals. This layered design allows the AI module to send commands back to ESP32, which triggers relays linked to PLC inputs (X2, X3), influencing control logic without disrupting PLC's real-time operations.

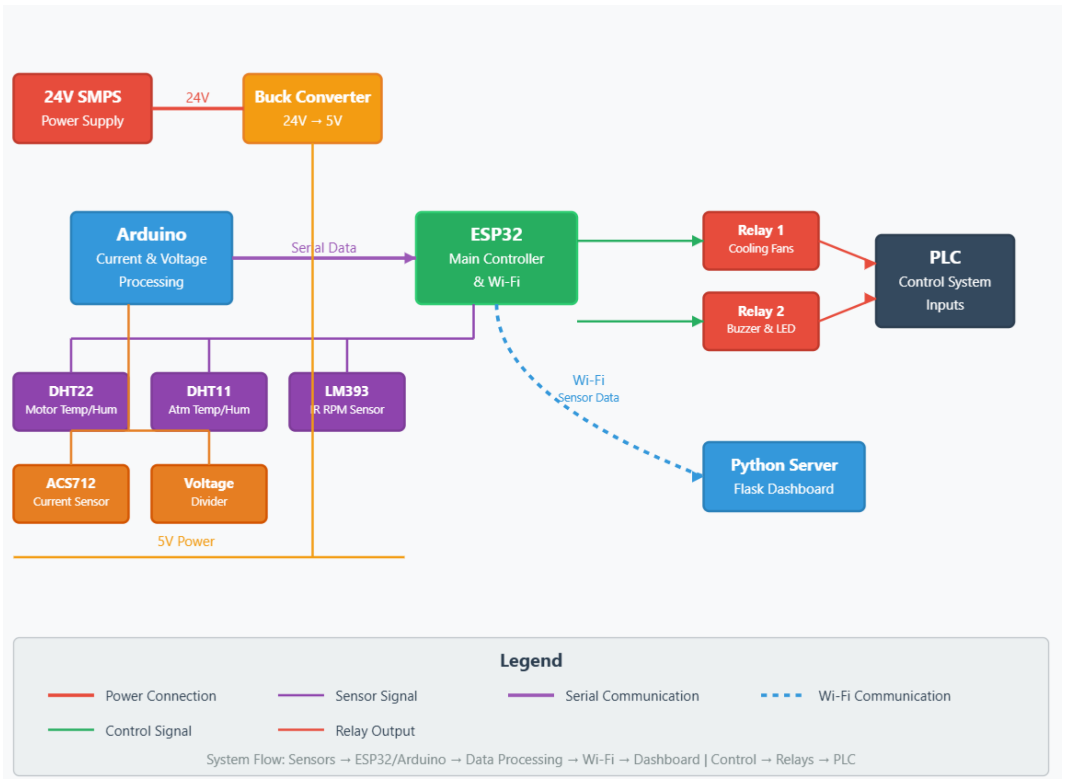


Figure 4.3: ESP32 Motor Monitoring System

4.4 Python-Based Server Implementation

At the core of the system lies a robust Python-based software layer that seamlessly bridges the hardware components with user interaction, serving as the operational backbone of the entire setup. The server, developed using Flask, a lightweight yet powerful web framework, runs locally on a network-accessible machine, enabling smooth communication and accessibility across devices. Its responsibilities go far beyond just hosting—it handles data acquisition, preprocessing, storage, real-time visualization, and AI-driven analysis. The central orchestration is managed through main.py, which integrates sensor data collection, interface modules, AI predictions, and dashboard control into a single workflow, ensuring that every module communicates efficiently and performs its role in harmony.

The communication between hardware and software is modular yet deeply interconnected. plc_interface.py establishes a reliable link with the Mitsubishi PLC via HTTP GET requests or Mitsubishi's MC Protocol, polling PLC memory registers for analog and digital I/O values at regular intervals. Meanwhile, esp_interface.py operates as an HTTP server endpoint, receiving JSON POST requests from the ESP32, parsing and validating the payload, and forwarding structured data for further processing. Once the raw inputs arrive, preprocessor.py takes over—cleaning and normalizing readings by converting analog signals into real-world units (for example, scaling 4–20 mA signals into °C or mapping voltage values), synchronizing timestamps, and gracefully handling missing data points. These processed readings are then committed to storage by data_logger.py, which supports both lightweight CSV logging for quick analysis and structured SQLite databases for long-term, query-friendly records.

The intelligence and usability of the system are elevated by its higher-level modules. ai_model.py uses trained regression and classification models to compute a motor health score and predict risks, drawing on historical patterns and real-time data alike. On the user-facing side, dashboard.py powers the interactive Flask-based interface, offering live graphs, sensor value updates, fault indicators, and manual override switches for motor and cooling control—all in real time. Supporting scripts like config.py centralize key settings, from IP addresses to safety thresholds, while utils.py hosts shared utility functions for scaling, unit conversion, and other common tasks. This carefully layered and modular design not only makes each component easier to update, test, and extend, but also ensures that the entire system remains flexible, maintainable, and ready for future upgrades.

Table 4.2: Python Modules and Their Functional Description

Module/File	**Function**
main.py	Entry point of the application; integrates all components
plc_interface.py	Communicates with Mitsubishi PLC to read analog and digital values
esp_interface.py	Receives HTTP POST data from ESP32 and forwards it for processing
preprocessor.py	Cleans, scales, and converts raw data to usable units
ai_model.py	Calculates motor health score and performs predictive maintenance logic
data_logger.py	Logs data into CSV files or a SQLite database
dashboard.py	Hosts the Flask web interface for live monitoring and control
config.py	Stores system configuration values such as IPs, thresholds, and log paths
utils.py	Helper functions for conversions, scaling, and repeated calculations

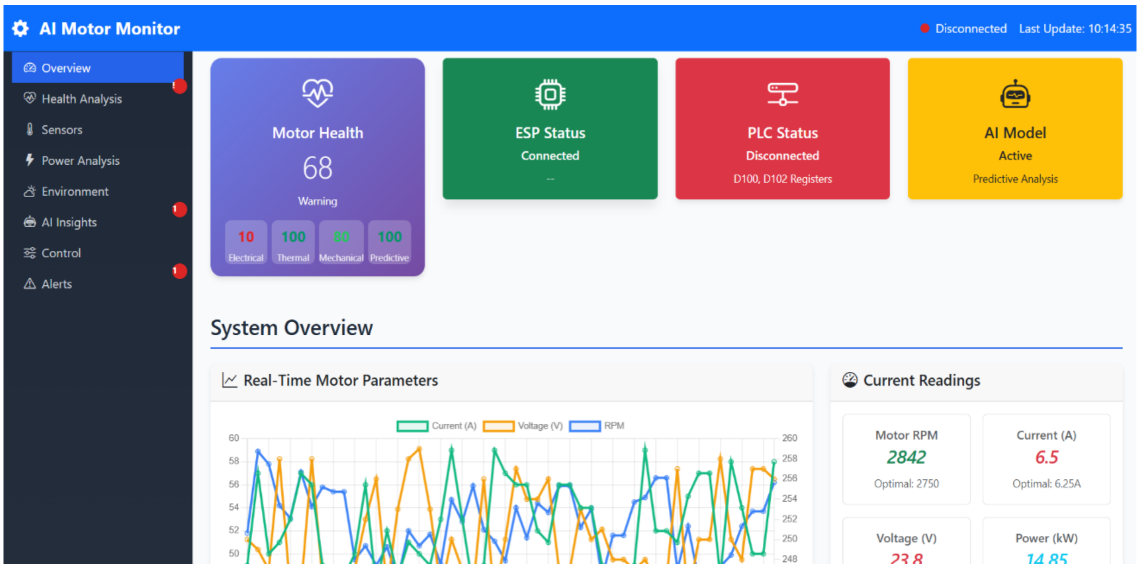
4.5 Dashboard Features and Controls

The user interface of the system was developed using a Flask-based dashboard that allows live visualization, system interaction, and alarm management—all through a local network. Figure 4.4 illustrates the dashboard layout, which serves as the front-end layer of the Python server, providing operators with clear and real-time insights into system performance without needing to interact directly with backend scripts or hardware.

The dashboard home page displays real-time sensor values from both the PLC and ESP32 subsystems. Parameters like motor temperature, voltage, load current, RPM, ambient temperature, and humidity are refreshed at short intervals using AJAX and WebSocket techniques. Each reading is visually represented using digital indicators and color-coded status bars. For example, motor temperature and current levels are dynamically coloured (green/yellow/red) based on predefined thresholds, helping users identify anomalies instantly.

Below the live data panel, the dashboard features time-series graphs plotted using Chart.js or Matplotlib embedded as images. These graphs display recent trends (e.g., the past 10 minutes or 1 hour) of key parameters such as temperature and RPM. Users can click tabs to view historical data for deeper analysis.

Manual control toggles are also available to trigger or override PLC-controlled actuators like the motor relay, fan relays, and buzzer. These toggles send commands to the Python server, which then writes to appropriate digital inputs of the PLC (like X2 or X3), ensuring safe relay actuation through standard logic. In the case of an anomaly predicted by the AI engine (such as overheating or irregular RPM), the dashboard immediately displays a visual and audible alarm on screen. These alerts can be acknowledged manually through the interface to reset the system to normal mode.


AI Motor Monitor

Disconnected Last Update: 10:14:35

Motor Health 68 Warning

10	100	80	100
Electrical	Thermal	Mechanical	Predictive

ESP Status Connected --

PLC Status Disconnected D100, D102 Registers

AI Model Active Predictive Analysis

System Overview

Current Readings

Motor RPM 2842 Optimal: 2750

Current (A) 6.5 Optimal: 6.25A

Voltage (V) 23.8

Power (kW) 14.85

Figure 4.4: Flask Dashboard

4.6 AI Model Integration and Predictive Maintenance

A key feature that sets this system apart from conventional monitoring platforms is its AI-enabled predictive maintenance engine, which adds an advanced layer of intelligence and foresight. This dedicated module, developed in ai_model.py, leverages both real-time and historical data to assess the motor's health and forecast potential risks such as overheating, excessive mechanical stress, or voltage imbalances before they escalate into failures. The system continuously gathers crucial input parameters—including motor temperature (from PT100 via PLC), current levels (from ACS712), RPM (from the IR sensor), and ambient temperature (from DHT22). These parameters are normalized and processed by a lightweight but effective machine learning model trained on labeled datasets that reflect both healthy and abnormal motor states, ensuring accurate interpretation of evolving conditions.

The AI model calculates a motor health score on a scale of 0 to 100, providing a clear, quantifiable indicator for operators. A score above 80 is considered healthy, scores between 50–80 fall into a warning zone requiring attention, and anything below 50 signals a potential or critical fault. This score is derived through either a weighted function or a regression output, depending on the chosen model configuration. In classification mode, the system can label the condition as "normal," "warning," or "fault," often utilizing algorithms like decision trees or logistic classifiers for reliable categorization.

When the health score dips below a defined threshold or the model detects a fault condition, it immediately signals the Python server to initiate preventive measures. These responses include:

- Sending a relay trigger to the PLC (via ESP) to activate cooling fans
- Turning on buzzer and LED indicators for on-site alerts
- Logging the incident with precise timestamps for historical analysis
- Generating a high-visibility visual alert on the dashboard for the user

By combining these actions, the predictive engine mitigates the risk of unplanned downtime, catching subtle degradation patterns before they escalate into costly breakdowns. It allows the system to adopt condition-based maintenance instead of traditional time-based schedules, significantly improving operational efficiency and equipment reliability. Importantly, the AI model is fully modular and designed for continuous evolution—it can be retrained with newly collected data over time, progressively enhancing its accuracy, adaptability, and long-term performance.

Table 4.3: AI Model Inputs and Parameters Used for Prediction

Parameter	Source	Unit	Role in AI Model
Motor Temperature	PT100 (via PLC)	°C	Key indicator of overheating
Motor Voltage	TX-205 (via PLC)	Volts	Checks voltage drops/spikes
Motor Current	ACS712 (Arduino)	Amperes	Detects load variation/stress
RPM	LM393 IR Sensor	Rotations/min	Identifies mechanical faults
Ambient Temp & Humidity	DHT22 (Arduino)	°C / %RH	Adjusts thresholds for contextual risk
Time Since Last Fault	Internal Timer	Minutes	Assists in trend-based prediction

4.7 Communication and Data Flow between Devices

The system architecture is designed to facilitate seamless data exchange between all hardware layers—PLC, Arduino, ESP32—and the central Python server. Communication is organized to ensure there is no conflict between time-critical operations handled by the PLC and the secondary tasks managed by the microcontrollers.

The Mitsubishi PLC communicates with the Python server either via HTTP GET requests (using the built-in WebServer module) or via MC Protocol through Ethernet. It shares real-time values such as analog input readings (motor temperature and voltage) and digital I/O states. This polling occurs at fixed intervals and is initiated by the Python server.

Figure 4.5 illustrates the overall communication flow between the PLC, Arduino, ESP32, and the Python server. On the microcontroller side, the Arduino collects sensor data—including RPM, current, temperature, and humidity—and sends it to the ESP32 via serial UART. The ESP32, connected to the local Wi-Fi network, converts this data into HTTP POST requests and pushes it to the Flask server. This two-layer approach ensures the Arduino focuses on precise sensor readings and interrupt handling, while the ESP32 handles networking and server interaction.

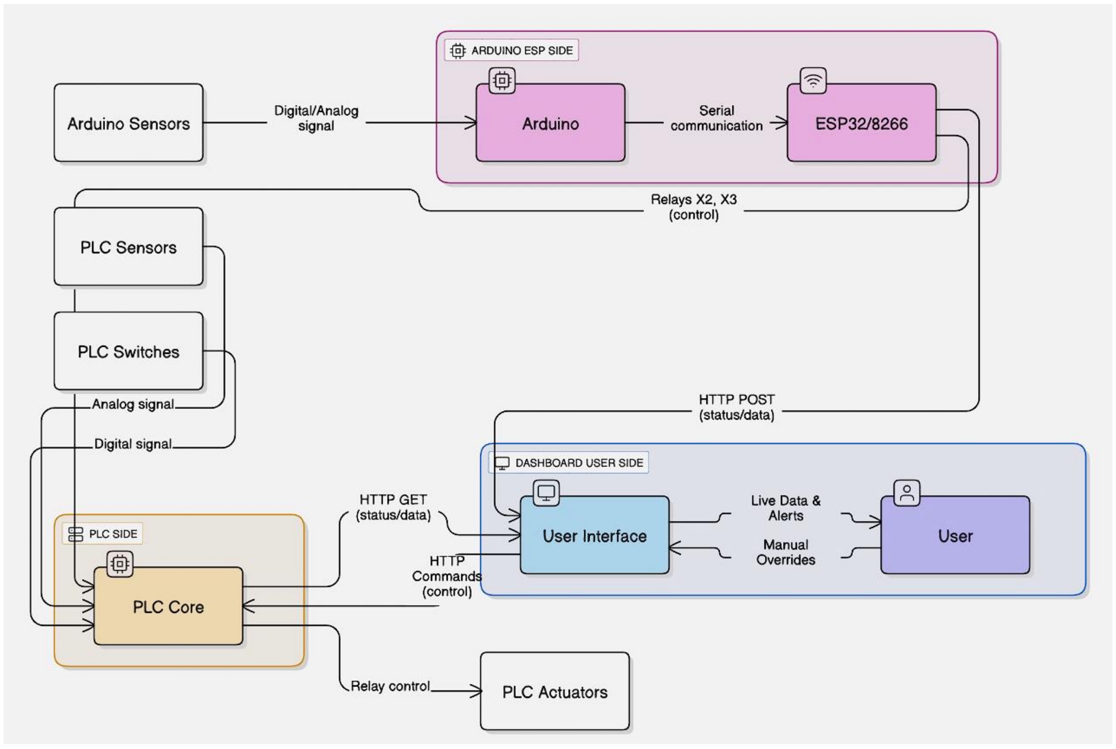


Figure 4.5: Communication flow diagram between PLC, ESP, Arduino, and Python Server

The Python server serves as the central hub, receiving input from both the PLC and ESP32, logging the data into its database, and processing it through the AI module. Based on thresholds or predictive insights, it can send control commands back to the PLC by triggering ESP-controlled relay signals wired into the PLC's digital inputs (X2 and X3).

This setup provides clear operational advantages. By isolating the PLC from microcontroller tasks, non-critical failures cannot interfere with safety-critical logic. The modular design also supports easy scalability, allowing new sensor nodes or cloud services to be added in future. Finally, the layered communication ensures that control decisions are made with minimal latency, preserving real-time responsiveness across the system.

Chapter 5

RESULTS AND TESTING

The testing phase was conducted to ensure that the designed system operates as intended in both hardware and software aspects. The primary objectives of testing were to validate the correct functioning of the PLC-based control system, the accurate acquisition of sensor data, reliable communication between devices, and the real-time display of information on the monitoring dashboard. Furthermore, the system was tested for its response to abnormal operating conditions to assess the effectiveness of the AI-based predictive maintenance model. The overall goal was to verify system reliability, data integrity, and response time under simulated industrial conditions.

5.1 Hardware Testing Results

The hardware components were tested individually and as part of the integrated system. The PLC output logic was first verified by manually triggering the Start and Stop switches and observing the corresponding actuation of the motor, cooling fans, and alarms. Each digital output (Y12 to Y18) was confirmed to respond accurately based on the ladder logic programmed in the Mitsubishi iQ-R PLC, as demonstrated in Figure 5.1.

Sensor testing was carried out by exposing each sensor to known conditions and comparing its output with standard reference values. The PT100 RTD sensor with 4–20 mA transmitter accurately measured the motor's surface temperature, while the TX-205 voltage transducer provided reliable motor voltage readings. On the Arduino side, the DHT22 sensor recorded ambient temperature and humidity, the ACS712 sensor effectively captured motor load current, and the LM393 IR sensor successfully detected RPM based on shaft rotation.

All sensors were wired according to the designed circuit, and their outputs were monitored on the PLC's web interface and Arduino serial monitor. The hardware testing confirmed that both the critical control system and the auxiliary monitoring subsystem functioned correctly and responded promptly to input changes. There were no observed delays in actuator response or sensor data acquisition during the test cycles.

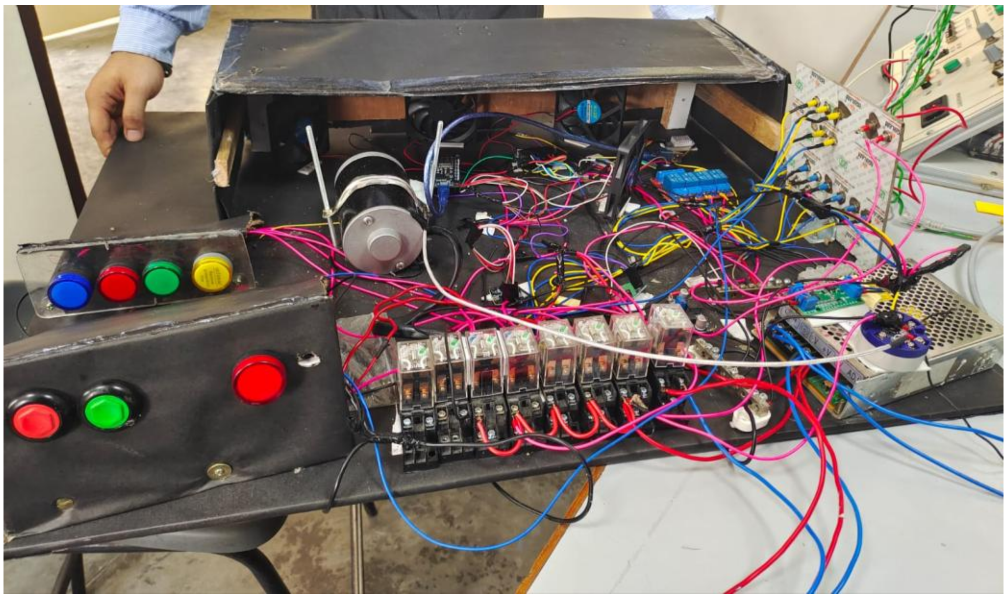


Figure 5.1: Hardware in operation showing PLC outputs and sensor testing setup

Table 5.1: Summary of Sensor and Actuator Testing

Component	**Test Performed**	**Expected Outcome**	**Result**
PT100 + Transmitter	Temperature rise simulation	4–20 mA range matched	Passed
TX-205 Voltage	Supply voltage change	Output scaled within 0–10 V	Passed
DHT22 Sensor	Exposed to varying temp/humidity	Accurate digital readings	Passed
ACS712 Sensor	Motor load test	Current output scaled correctly	Passed
LM393 IR Sensor	RPM counting with reflective patch	Correct pulse count and RPM	Passed
PLC Digital Outputs	Start/Stop/Fan/Buzzer test	Actuators responded as programmed	Passed

5.2 Communication and Dashboard Testing

The communication flow between the Arduino, ESP32, PLC, and the Python server was thoroughly tested under controlled conditions. The Arduino successfully transmitted sensor data via serial communication to the ESP32, which, in turn, forwarded this data to the Python server using HTTP POST requests. The server processed the received JSON payloads without loss or corruption of data.

The Mitsubishi PLC communicated effectively with the Python server through HTTP GET requests using the PLC's WebServer module. Real-time analog and digital input values were consistently read and updated on the server side at predefined polling intervals. This confirmed the reliability of both direct communication (PLC to Python) and indirect communication (Arduino/ESP32 to Python).

On the Flask-based dashboard (shown in Figure 5.2), all monitored parameters—including motor temperature, voltage, current, RPM, ambient temperature, and humidity—were displayed accurately and updated in near real-time. The dashboard also successfully reflected actuator statuses and allowed manual control commands (like turning on fans or alarms) to be sent to the PLC via the Python server. All control actions performed via the dashboard triggered the expected response in the hardware setup, with negligible delay.

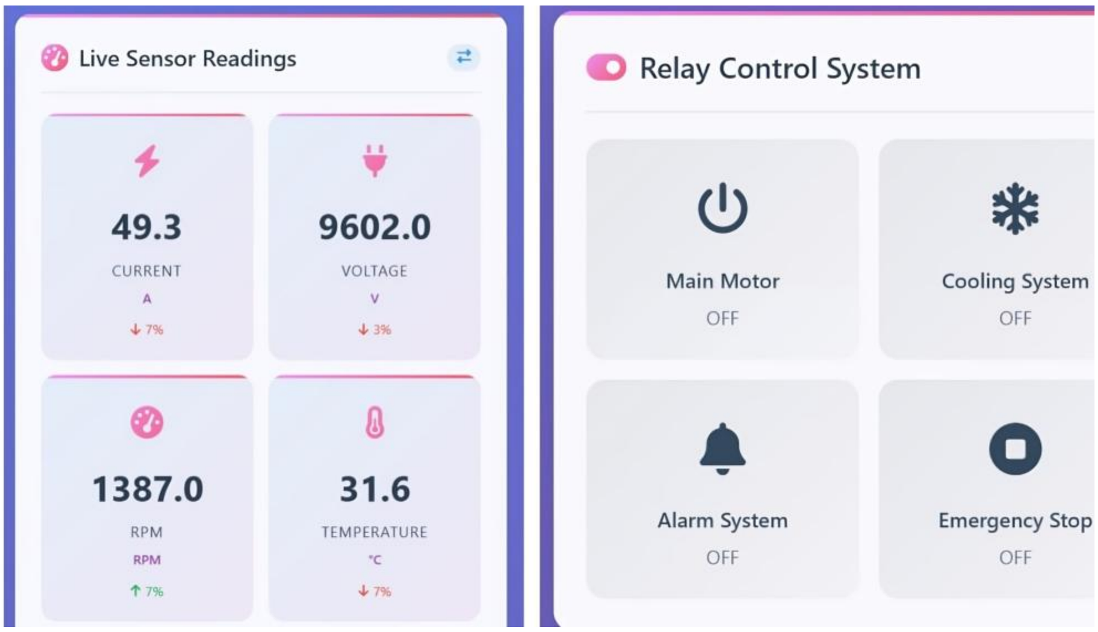


Figure 5.2: Flask dashboard displaying live sensor values and manual control buttons

5.3 AI Model Output and Alerts

The AI-based health scoring and predictive maintenance module was tested using both real-time sensor data and historical logs. The system consistently calculated a motor health score based on parameters like temperature, current, voltage, and RPM, and the dashboard visualization (shown in Figure 5.3) reflected these calculations in real time.

The output health score accurately tracked the operating condition of the motor and dynamically updated as sensor values changed.

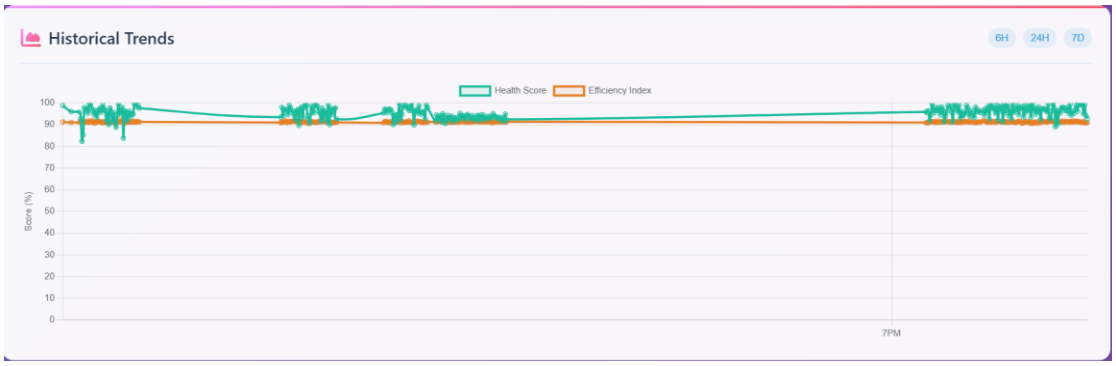


Figure 5.3: Health score trend on dashboard and AI-predicted alert notification

In scenarios simulating abnormal conditions—such as artificially raising the motor temperature or reducing RPM—the AI module successfully detected risk and generated predictive alerts. These alerts triggered corresponding actions, such as activating the cooling fans or sounding the buzzer through the PLC control. The health score and alert status were also displayed on the dashboard, providing the user with timely warnings.

Table 5.2: Sample Health Score and AI Prediction Outcomes

Condition Simulated	**Health Score**	**AI Prediction**	**System Action**
Normal operation	85 - 100	Healthy	None
Rising temperature	60 - 85	Warning	Fans activated
High current + low RPM	<45	Critical Fault	Buzzer, Fans ON

5.4 Overall System Behavior

The integrated system was observed to perform reliably across multiple testing sessions. Sensor readings were accurate and stable, actuator responses were immediate, and communication between devices remained consistent without data loss. The AI module provided timely health assessments and predictive alerts, allowing preventive action before reaching critical states.

The system exhibited excellent coordination between the PLC's deterministic control, Arduino's auxiliary sensing, ESP32's communication handling, and the Python server's data management. Latency between sensor data capture and dashboard display was minimal—generally under one second. There were no significant system crashes or communication failures during extended test runs.

Table 5.3: Final Observations on System Performance

Aspect Tested	**Observation**	**Result**
Sensor Accuracy	Readings within acceptable limits	Satisfactory
PLC Control Response	Instantaneous on input triggers	Satisfactory
Communication Delay	Under 1 second (typical)	Acceptable
Dashboard Update	Real-time with minimal lag	Acceptable
AI Prediction Accuracy	Aligned with expected outcomes	Satisfactory

Chapter 6:
CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

The development and implementation of the AI-Enabled Industrial Motor Health & Environment Monitoring System successfully demonstrated the integration of industrial automation with modern IoT and AI technologies. The hybrid architecture, which combined a Mitsubishi iQ-R PLC for critical control with an Arduino and ESP32 for auxiliary sensing, proved effective in achieving reliable motor monitoring and control. The system was able to monitor essential parameters such as motor temperature, voltage, load current, RPM, and ambient conditions in real time.

The Python-based server provided seamless data acquisition, storage, and AI-driven analysis, while the Flask dashboard offered a user-friendly interface for live monitoring, manual control, and predictive alerts. Testing validated that both hardware and software components functioned cohesively, with accurate sensor readings, prompt actuator response, and reliable communication between all devices.

By incorporating a predictive maintenance model, the system moved beyond traditional monitoring to provide proactive insights into motor health, helping to reduce downtime and prevent potential failures. The project met its objectives of creating a scalable, cost-effective, and intelligent monitoring solution suitable for industrial environments.

6.2 Future Scope

While the current system has achieved its intended functionality, there are several areas identified for future enhancement. One key improvement is the integration of cloud-based data storage and remote dashboard access, enabling real-time monitoring and control from any location. Implementing advanced AI algorithms, such as deep learning models for more accurate health prediction, would further increase system reliability and predictive capability.

The addition of industrial-grade sensors for RPM and current measurement could improve durability and measurement accuracy in harsh environments. Expanding the system to support Modbus or OPC UA communication would also enhance compatibility with existing industrial networks.

Finally, automating the retraining process for the AI model using continuously collected data can make the system self-adaptive, ensuring it remains effective as operating conditions evolve. These future enhancements would elevate the system into a more robust Industrial IoT solution, aligning with Industry 4.0 objectives.

REFERENCES

[1] Sabo, S., Umaru, A.M. and Yusuf, L.A., 2025, “Smart IoT-Based Broiler Room Controller: Design, Implementation, Performance Evaluation, and Optimization”, Journal of Science and Technology, Volume Number 30, Issue Number 2, pp. 180-190.

[2] Dong, D. and Feng, H., 2024, “Design and use of a wireless temperature measurement network system integrating artificial intelligence and blockchain in electrical power engineering”, PLoS One, Volume Number 19, Issue Number 1.

[3] Jiang, Jia, S. and Guo, H., 2024, “IoT-enabled framework for a sustainable vaccine cold chain management system”, Heliyon, Volume Number 10, Issue Number 7.

[4] Sukkam, N., Jitpakdee, P. and Saenkhum, S., 2024, “Machine Learning Prediction of a Battery’s Thermal-Related Health Factor in a Battery Electric Vehicle Using Real-World Driving Data”, Information, Volume Number 15, Issue Number 9, pp. 553.

[5] Ghahramani, A., Wang, K., Becerik-Gerber, B. and Kavulya, M., 2020, “Artificial Intelligence for Efficient Thermal Comfort Systems: Requirements, Current Applications and Future Directions”, Frontiers in Built Environment, Volume Number 6, pp. 49.

[6] Rahaman, M.H. and Iqbal, M.T., 2020, “A Remote Thermostat Control and Temperature Monitoring System of a Single-Family House Using openHAB and MQTT”, European Journal of Electrical Engineering and Computer Science, Volume Number 4, Issue Number 5, pp. 1-6.

[7] Rattanapoka, C., Wongchaisuwat, N. and Thongprapha, N., 2019, “An MQTT-based IoT Cloud Platform with Flow Design by Node-RED”, Proceedings of the IEEE Conference on IoT, Kaohsiung, Taiwan, pp. 123-128.

[8] Ding, S. and Li, W., 2013, “Temperature Monitoring System Based on PLC”, TELKOMNIKA, Volume Number 11, Issue Number 12, pp. 7251-7258.

[9] Ruiz-Garcia, L., Lunadei, L., Barreiro, P. and Robla, J.I., 2009, “A Review of

Wireless Sensor Technologies and Applications in Agriculture and Food Industry: State of the Art and Current Trends", Sensors, Volume Number 9, Issue Number 6, pp. 4728-4750.

APPENDIX

Appendix A

ARDUINO NANO CODE

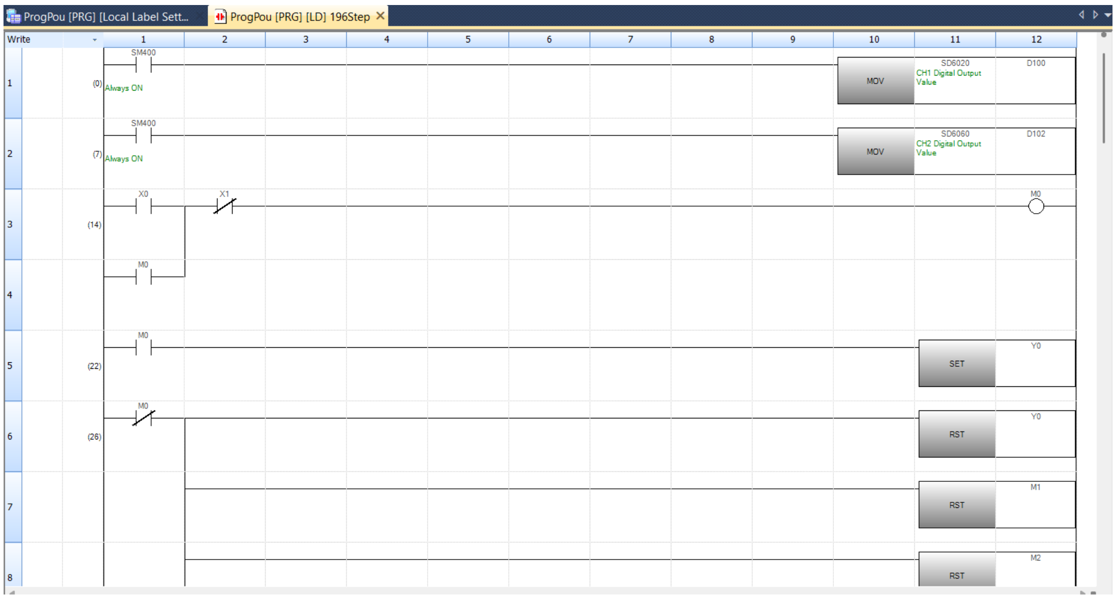
```
#include <Arduino.h>
#include <SoftwareSerial.h>   // For communication with ESP-01 or other
serial devices

// --- SoftwareSerial Configuration ---
// Connect Arduino Nano Digital Pin 3 to ESP-01 RX
// Connect Arduino Nano Digital Pin 4 to ESP-01 TX
SoftwareSerial esp(3, 4);

// --- DHT Sensor Libraries ---
// Removed Adafruit_Sensor.h and DHT_U.h as they are not needed for the
standard DHT.h library
#include <DHT.h>              // Standard DHT library (includes
computeHeatIndex)

#define DHTPIN 5              // Digital pin connected to the DHT
sensor
#define DHTTYPE DHT22         // DHT 11 sensor type

// DHT sensor instance (using the simpler DHT library for
computeHeatIndex)
DHT dht(DHTPIN, DHTTYPE);

// --- Relay Pin Definitions ---
const int RLY1 = 10; // Digital Pin 10
const int RLY2 = 11; // Digital Pin 11
const int RLY3 = 12; // Digital Pin 12

// --- Sensor Pin Definitions ---
const int CURRENT_SENSOR_PIN = A0; // Analog pin for current sensor
const int VOLTAGE_SENSOR_PIN = A1; // Analog pin for voltage sensor
const int SPEED_SENSOR_PIN = 2;    // Digital pin for speed sensor
(External Interrupt 0)

// --- RPM Measurement Variables ---
volatile int pulseCount = 0;      // Use volatile for interrupt-driven
counting
unsigned long startTime = 0;
const unsigned long MEASUREMENT_DURATION_MS = 10000; // Measure speed
over 10 seconds (10000 milliseconds)
```

```
int rpm = 0;  // Current calculated RPM
int prpm = 0; // Previous calculated RPM (for averaging)

// --- Sensor Reading Variables ---
unsigned int Sensor1_Val = 0; // Stores Current Sensor value
unsigned int Sensor2_Val = 0; // Stores Voltage Sensor value
unsigned int Sensor3_Val = 0; // Stores RPM value
unsigned int Sensor4_Val = 0; // Stores DHT Temperature (Celsius)
unsigned int Sensor5_Val = 0; // Stores DHT Humidity
unsigned int Sensor6_Val = 0; // Stores DHT Temperature (Fahrenheit)
unsigned int Sensor7_Val = 0; // Stores DHT Heat Index (Celsius)
unsigned int Sensor8_Val = 0; // Stores DHT Heat Index (Fahrenheit)

// --- Relay Status Variables ---
unsigned int RLY1_STATUS = 0;
unsigned int RLY2_STATUS = 0;
unsigned int RLY3_STATUS = 0;

// --- Control Flags/Delays ---
unsigned int ADC_send_delay = 0; // Keeping for consistency, though not
actively used in this loop structure
unsigned int ADC_send = 1;       // Set to 1 to ensure initial sensor
read and send

// --- Sensor Thresholds ---
#define CURRENT_NORMAL 230 // Analog value: Below this is considered
normal
#define CURRENT_ALARM  700  // Analog value: Above this is considered
an alarm
#define CURRENT_BUZZER 800 // Analog value: Above this for buzzer (if
used)

#define VOLTAGE_NORMAL 10000  // Analog value: Below this is considered
normal
#define VOLTAGE_ALARM 20000   // Analog value: Above this is considered
an alarm
#define VOLTAGE_BUZZER 12000  // Analog value: Above this for buzzer
(if used)

#define DHT11_NORMAL 30    // Temperature threshold in Celsius
#define DHT11_ALARM 30     // Temperature alarm threshold
#define DHT11_BUZZER 35    // Temperature buzzer threshold

// --- Buffers for sprintf (for sending data via SoftwareSerial) ---
char foo[10]; // For Sensor1_Val (CURRENT)
char bar[10]; // For Sensor2_Val (VOLTAGE)
char goo[10]; // For Sensor3_Val (RPM)
char har[10]; // For Sensor4_Val (DHT Temp C)
char ioo[10]; // For Sensor5_Val (DHT Humidity)
char jar[10]; // For Sensor6_Val (DHT Temp F)
char loo[10]; // For Sensor7_Val (DHT Heat Index C)
char mar[10]; // For Sensor8_Val (DHT Heat Index F)
char nob[10]; // For RLY1_STATUS ("ON"/"OFF")
char tot[10]; // For RLY2_STATUS ("ON"/"OFF")
char zar[10]; // For RLY3_STATUS ("ON"/"OFF")
char yar[10]; // For Combined Relay Status ("NOR"/"ALM"/"BUZ")
char uar[10]; // For "RSV" (Reserved)

// --- Function Prototypes (Declarations) ---
void countPulse();        // ISR for RPM sensor
void Cal_RPM();           // Calculates RPM
void ReadSensor_values(); // Reads Current, Voltage, and DHT sensors
void RelayOperation();    // Controls relays based on sensor values
void DHT11_init();        // Initializes DHT sensor
void DHT11_read();        // Reads DHT sensor data

// -----------------------------------------------------------------------
------------------------------------------------
// Setup Function: Runs once when you press reset or power the board
// -----------------------------------------------------------------------
------------------------------------------------
void setup() {
  Serial.begin(9600); // Initialize hardware serial for debugging 
output
  esp.begin(9600);    // Initialize software serial for communication 
with ESP-01

  // Initialize Relay Pins as OUTPUTs
  pinMode(RLY1, OUTPUT);
  pinMode(RLY2, OUTPUT);
  pinMode(RLY3, OUTPUT);

  // Set all relays OFF initially (assuming HIGH means OFF for your 
relays)
  digitalWrite(RLY1, HIGH);
  digitalWrite(RLY2, HIGH);
  digitalWrite(RLY3, HIGH);

  // Initialize Analog Sensor Input Pins
  pinMode(CURRENT_SENSOR_PIN, INPUT); // Analog input
  pinMode(VOLTAGE_SENSOR_PIN, INPUT); // Analog input

  // Initialize DHT11 Sensor
  DHT11_init();

  // Configure the Speed Sensor input pin
  pinMode(SPEED_SENSOR_PIN, INPUT_PULLUP); // Enable internal pull-up
resistor

  // Attach the interrupt for the speed sensor
  // Digital Pin 2 corresponds to External Interrupt 0 (INT0) on
Arduino Nano.
  // RISING means the interrupt triggers when the signal goes from LOW
to HIGH.
  attachInterrupt(digitalPinToInterrupt(SPEED_SENSOR_PIN), countPulse,
RISING);

  // Initialize startTime for the first measurement window
  startTime = millis() + MEASUREMENT_DURATION_MS;

  Serial.println("\nArduino Nano Setup Complete.");
  Serial.println("Waiting for sensor readings and RPM pulses...");
}

// -----------------------------------------------------------------------
-----------------------------------------------
// Loop Function: Runs over and over again forever
// -----------------------------------------------------------------------
-----------------------------------------------
void loop() {
  // Check if the measurement duration has passed
  if (millis() - startTime >= MEASUREMENT_DURATION_MS) {
    // Perform all readings and operations
    Cal_RPM();           // Calculate RPM
    ReadSensor_values(); // Read Current, Voltage, and DHT sensors
    RelayOperation();    // Control relays based on sensor values

    // Populate char arrays with sensor status/values for sending via
SoftwareSerial
    sprintf(foo, "%d", Sensor1_Val);   // CURRENT
    sprintf(bar, "%d", Sensor2_Val);   // VOLTAGE
    sprintf(goo, "%d", Sensor3_Val);   // RPM
    sprintf(har, "%d", Sensor4_Val);   // DHT Temp C
    sprintf(ioo, "%d", Sensor5_Val);   // DHT Humidity
    sprintf(jar, "%d", Sensor6_Val);   // DHT Temp F
    sprintf(loo, "%d", Sensor7_Val);   // DHT Heat Index C
    sprintf(mar, "%d", Sensor8_Val);   // DHT Heat Index F

    // Relay Status strings
    if (RLY1_STATUS == 0) { sprintf(nob, "%s", "OFF"); } else {
sprintf(nob, "%s", "ON"); }
    if (RLY2_STATUS == 0) { sprintf(tot, "%s", "OFF"); } else {
sprintf(tot, "%s", "ON"); }
```

```
    if (RLY3_STATUS == 0) { sprintf(zar, "%s", "OFF"); } else {
sprintf(zar, "%s", "ON"); }

    /*// Combined Relay Status (based on your logic)
    if ((RLY1_STATUS == 1) && (RLY2_STATUS == 0) && (RLY3_STATUS == 0))
{
      sprintf(yar, "%s", "NOR"); // Normal
    } else if ((RLY1_STATUS == 0) && (RLY2_STATUS == 1) && (RLY3_STATUS
== 0)) {
      sprintf(yar, "%s", "ALM"); // Alarm
    } else if ((RLY1_STATUS == 0) && (RLY2_STATUS == 0) && (RLY3_STATUS
== 1)) {
      sprintf(yar, "%s", "BUZ"); // Buzzer/Shutdown
    } else {
      sprintf(yar, "%s", "INV"); // Invalid State (or another default)
    }*/
    if ((RLY1_STATUS == 0) && (RLY2_STATUS == 0) && (RLY3_STATUS == 0))
{
      sprintf(yar, "%s", "NOR"); // Normal
    } else if ((RLY1_STATUS == 1) || (RLY2_STATUS == 1) || (RLY3_STATUS
== 1)) {
      sprintf(yar, "%s", "BUZ"); // Alarm
    }
    sprintf(uar, "%s", "RSV"); // Reserved

    // Construct the full serial string to send to ESP-01
    char str[200]; // Increased buffer size for the concatenated string
    strcpy(str, "ADU_TEXT&");
    strcat(str, foo); // CURRENT
    strcat(str, "&");
    strcat(str, bar); // VOLTAGE
    strcat(str, "&");
    strcat(str, goo); // RPM
    strcat(str, "&");
    strcat(str, har); // DHT Temp C
    strcat(str, "&");
    strcat(str, ioo); // DHT Humidity
    strcat(str, "&");
    strcat(str, jar); // DHT Temp F
    strcat(str, "&");
    strcat(str, loo); // DHT Heat Index C
    strcat(str, "&");
    strcat(str, mar); // DHT Heat Index F
    strcat(str, "&");
    strcat(str, nob); // RLY1_STATUS
    strcat(str, "&");
    strcat(str, tot); // RLY2_STATUS
    strcat(str, "&");
    strcat(str, zar); // RLY3_STATUS
    strcat(str, "&");
    strcat(str, yar); // Combined Relay Status
    strcat(str, "&");
    strcat(str, uar); // Reserved

    Serial.print("\nSending to ESP: ");
    Serial.println(str);
    esp.print(str); // Send data to ESP-01 via SoftwareSerial

    // Reset startTime for the next measurement window
    startTime = millis();
  }

  // Small delay for stability
  delay(10);
}

// -----------------------------------------------------------------------
------------------------------------------------
// Helper Functions (Definitions)
// -----------------------------------------------------------------------
------------------------------------------------

/**
 * @brief Interrupt Service Routine (ISR) for the speed sensor.
 * Increments pulseCount on each rising edge.
 */
void countPulse() {
  pulseCount++;
}

/**
 * @brief Calculates RPM based on pulseCount over the measurement
duration.
 * Resets pulseCount and restarts the measurement window.
 */
void Cal_RPM() {
  //pulseCount = 0 ;
  // Temporarily disable interrupts to safely read and reset pulseCount
  detachInterrupt(digitalPinToInterrupt(SPEED_SENSOR_PIN));

  Serial.print("\nRaw Pulse Count: ");
  Serial.print(pulseCount);

  // Your original RPM calculation logic was commented out,
  // and you have a new custom calculation based on pulseCount/400.
  // I'm using your new logic directly.
  if (pulseCount == 0) {
    prpm = 0;
  } else {
    if (prpm == 0) {
      prpm = pulseCount * 3; // Initial calculation
    } else {
      prpm = (prpm + (pulseCount *3)) / 2; // Averaging
    }
  }

  Serial.print("\tCalculated RPM (prpm): ");
  Serial.println(prpm);
  Sensor3_Val = prpm; // Store calculated RPM in Sensor3_Val
  Serial.print("Sensor3_Val (RPM): ");
  Serial.println(Sensor3_Val);

  pulseCount = 0; // Reset pulse counter
  // startTime is reset in loop() after all operations for the current
window.
  // Re-enable interrupts to continue counting pulses for the next
window
  attachInterrupt(digitalPinToInterrupt(SPEED_SENSOR_PIN), countPulse,
RISING);
}

/**
 * @brief Reads values from Current, Voltage, and DHT sensors.
 */
void ReadSensor_values() {
  Serial.println("\n--------------------------------");
  Sensor1_Val = 0 ;
  Sensor2_Val = 0 ;

  // Read Current Sensor (Analog)
  Sensor1_Val = analogRead(CURRENT_SENSOR_PIN);
  Serial.print("Current Sensor Value (raw): ");
  Serial.println(Sensor1_Val);
  // Apply your scaling factor (multiplied by 6)
  if( Sensor1_Val < 20)
    { Sensor1_Val = 0 ; }
  else
    { Sensor1_Val = Sensor1_Val /3.4; }
  Serial.print("Current Sensor Value (Sensor1_Val): ");
  Serial.println(Sensor1_Val);
  delay(10); // Small delay after reading analog

  // Read Voltage Sensor (Analog)
  Sensor2_Val = analogRead(VOLTAGE_SENSOR_PIN);
  Serial.print("Voltage Sensor Value (raw): ");
  Serial.println(Sensor2_Val);
  // Apply your scaling factor (divided by 45, then multiplied by 1000)
  Sensor2_Val = (unsigned int)((float)Sensor2_Val / 42.8 * 1000.0);
  if(Sensor2_Val <= 1000)
     {  Sensor2_Val = 0 ; }
  Serial.print("Voltage Sensor Value (Sensor2_Val): ");
  Serial.println(Sensor2_Val);
  delay(10); // Small delay after reading analog

  // Read DHT sensor
  DHT11_read();
  delay(10); // Small delay after DHT read
}

/**
 * @brief Controls relays based on Current, Voltage, and DHT
Temperature values.
 */
void RelayOperation() {
  Serial.println("\n--- Performing Relay Operations ---");

  Serial.print("Current RLY1_STATUS = "); Serial.print(RLY1_STATUS);
  Serial.print(", RLY2_STATUS = "); Serial.print(RLY2_STATUS);
  Serial.print(", RLY3_STATUS = "); Serial.print(RLY3_STATUS);

  // Logic for relay control based on sensor thresholds
  // All sensors normal
  //if ((Sensor1_Val <= CURRENT_NORMAL) && (Sensor2_Val <=
VOLTAGE_NORMAL) && (Sensor4_Val <= DHT11_NORMAL)) {
  if ((Sensor2_Val <= VOLTAGE_NORMAL) ) {
    Serial.println("\n Voltage values is normal.");
    if(RLY2_STATUS == 1)
    {
       digitalWrite(RLY2, HIGH);  // RLY1 ON
       Serial.print("RLY2 = OFF");
    }
    RLY2_STATUS = 0;

  }
  else if ((Sensor2_Val > VOLTAGE_NORMAL) ) {
    Serial.println("\nVoltage values are above normal.");
    if(RLY2_STATUS == 0)
    {
        digitalWrite(RLY2, LOW);  // RLY1 ON
        Serial.print("RLY2 = ON");
    }
    RLY2_STATUS = 1;
  }
  // Alarm condition (Current OR Voltage in alarm range)
  if (Sensor1_Val <= CURRENT_NORMAL) {
    Serial.println("\nCurrent value is normal.");
    if(RLY1_STATUS == 1)
    {
      digitalWrite(RLY1, HIGH); // RLY1 OFF
      Serial.print("RLY1 = OFF");
    }
    RLY1_STATUS = 0;

  }
  else if (Sensor1_Val > CURRENT_NORMAL) {
    Serial.println("\nCurrent value above normal\.");
    if(RLY1_STATUS == 0)
    {
       digitalWrite(RLY1, LOW); // RLY1 OFF
       Serial.print("RLY1 = ON");
    }
    RLY1_STATUS = 1;

  }

  // Shutdown condition (Current OR Voltage above alarm, OR DHT Temp
above normal)
  if ( (Sensor4_Val <= DHT11_NORMAL)) {
    Serial.println("\nTemperature is normal.");
    if(RLY3_STATUS == 1)
    {
      digitalWrite(RLY3, HIGH);  // RLY3 ON
      Serial.print("RLY3 = OFF");
    }
    RLY3_STATUS = 0;
  }
  else if ( (Sensor4_Val > DHT11_NORMAL)) {
    Serial.println("\nTemperature above normal.");
    if(RLY3_STATUS == 0)
    {
        digitalWrite(RLY3, LOW);  // RLY3 ON
        Serial.print("RLY3 = ON");
    }
    RLY3_STATUS = 1;
  }


  Serial.print("\nFinal RLY1_STATUS = "); Serial.print(RLY1_STATUS);
  Serial.print(", RLY2_STATUS = "); Serial.print(RLY2_STATUS);
  Serial.print(", RLY3_STATUS = "); Serial.println(RLY3_STATUS);
}

/**
 * @brief Initializes the DHT sensor.
 */
void DHT11_init() {
  dht.begin(); // Initialize the DHT sensor
  Serial.println(F("DHT Sensor Initialized."));
}

/**
 * @brief Reads temperature and humidity from the DHT sensor using DHT
methods.
 * Stores values in Sensor4_Val to Sensor8_Val, including calculated
Heat Index.
 */
void DHT11_read() {
  // Reading temperature or humidity takes about 250 milliseconds!
  // Sensor readings may also be up to 2 seconds 'old' (its a very slow
sensor)

  // Read humidity as percentage (the default)
  float h = dht.readHumidity();
  // Read temperature as Celsius (the default)
  float t = dht.readTemperature();
  // Read temperature as Fahrenheit (isFahrenheit = true)
  float f = dht.readTemperature(true);

  // Check if any reads failed and exit early (to try again).
  if (isnan(h) || isnan(t) || isnan(f)) {
    Serial.println(F("Failed to read from DHT sensor!"));
    return;
  }

  // Compute heat index in Fahrenheit (the default)
  float hif = dht.computeHeatIndex(f, h);
  // Compute heat index in Celsius (isFahreheit = false)
  float hic = dht.computeHeatIndex(t, h, false);

  // Store values in global Sensor variables
  Sensor4_Val = (unsigned int)t;   // Temperature in Celsius
  Sensor5_Val = (unsigned int)h;   // Humidity
  Sensor6_Val = (unsigned int)f;   // Temperature in Fahrenheit
  Sensor7_Val = (unsigned int)hic; // Heat Index in Celsius
  Sensor8_Val = (unsigned int)hif; // Heat Index in Fahrenheit

  // Print all readings to Serial Monitor for debugging
  Serial.print(F("Humidity: "));
  Serial.print(h);
  Serial.print(F("% \tTemperature: "));
  Serial.print(t);
  Serial.print(F("°C "));
  Serial.print(f);
  Serial.print(F("°F \tHeat index: "));
  Serial.print(hic);
  Serial.print(F("°C "));
  Serial.print(hif);
  Serial.println(F("°F"));
}
```

Appendix B:

LADDER PROGRAM


ProgPou [PRG] [Local Label Sett...
ProgPou [PRG] [LD] 196Step
Write
1 2 3 4 5 6 7 8 9 10 11 12
(0) SM400 Always ON MOV SD6020 CH1 Digital Output Value D100
(7) SM400 Always ON MOV SD6060 CH2 Digital Output Value D102
(14) X0 X1 M0
M0
(22) M0 SET Y0
(26) M0 RST Y0
RST M1
RST M2

Appendix C:

ESP CODE

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>

// --- WiFi Credentials ---
const char* ssid = "CMF2P";
const char* password = "basharat";

// --- Server Configuration ---
// Replace with your PC's local IP and port for the Flask server
const char* serverUrl = "http://192.168.46.126:5000/send-data";

// --- Data Buffers for Serial Input ---
// Max size for each data segment, allowing for null terminator
// Using char arrays for fixed-size strings.
char data1[30] = "";  // e.g., "ADU_TEXT"
char data2[30] = "";  // e.g., IR1 status
char data3[30] = "";  // e.g., IR2 status
char data4[30] = "";  // e.g., IR3 status
char data5[30] = "";  // e.g., IR4 status
char data6[30] = "";  // e.g., FLAME status
char data7[30] = "";  // e.g., TEMP value
char data8[30] = "";  // e.g., HUMIDITY value
char data9[30] = "";  // e.g., ALARM status
char data10[30] = ""; // e.g., GATE status
char data11[30] = ""; // e.g., SMOKE status
char data12[30] = ""; // e.g., TSTS status
char data13[30] = ""; // e.g., Extra data
char data14[30] = ""; // e.g., Extra data

// --- Function Prototypes ---
void sendData();
void RelayOperation(); // Prototype for the relay control function


// --- Setup Function ---
void setup() {
  Serial.begin(9600); // Initialize serial communication

  // Connect to Wi-Fi
  Serial.print("Connecting to WiFi...");
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500); // Reduced delay for faster connection attempts
    Serial.print(".");
  }
  Serial.println("\nConnected to WiFi!");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());

}

// --- sendData Function: Handles HTTP POST Request ---
void sendData() {
  if (WiFi.status() == WL_CONNECTED) {
    WiFiClient client;
    HTTPClient http;

    Serial.println("Attempting HTTP POST...");

    // Start HTTP connection
    if (http.begin(client, serverUrl)) {
      http.addHeader("Content-Type", "application/json"); // Set
content type for JSON

      // Construct JSON data string with all 13 parsed values
      // Ensure the keys match what your Flask server expects.
      // I've used generic keys for data1 and data13.
      char jsonBuffer[300]; // Increased buffer size for more data
      sprintf(jsonBuffer, "{\"TYPE\": \"%s\", \"VAL1\": \"%s\",
\"VAL2\": \"%s\", \"VAL3\": \"%s\", \"VAL4\": \"%s\", \"VAL5\": \"%s\",
\"VAL6\": \"%s\", \"VAL7\": \"%s\", \"VAL8\": \"%s\", \"VAL9\": \"%s\",
\"VAL10\": \"%s\", \"VAL11\": \"%s\", \"VAL12\": \"%s\"}",
              data1, data2, data3, data4, data5, data6, data7, data8,
data9, data10, data11, data12, data13 );

      Serial.print("Sending JSON: ");
      Serial.println(jsonBuffer);

      // Send the POST request
      int httpResponseCode = http.POST(jsonBuffer);

      // Process HTTP response
      if (httpResponseCode > 0) {
        String response = http.getString();
        Serial.printf("HTTP Response Code: %d\n", httpResponseCode);
        Serial.println("Server Response: " + response);
      } else {
        Serial.printf("HTTP POST failed. Error code: %d - %s\n",
httpResponseCode, http.errorToString(httpResponseCode).c_str());
      }

      http.end(); // Free resources
    } else {
      Serial.println("HTTP begin failed. Check serverUrl or network.");
    }
  } else {
    Serial.println("WiFi not connected. Attempting to reconnect...");
    WiFi.begin(ssid, password); // Attempt to reconnect if WiFi is lost
  }
}

// --- Loop Function ---
void loop() {
  // Check if there's data available in the Serial buffer
  if (Serial.available() > 0) {
    char receivedBuffer[501]; // Buffer to store received serial data
    memset(receivedBuffer, 0, sizeof(receivedBuffer)); // Clear buffer

    // Read bytes until newline character or buffer is full
    // Reads up to 500 characters + null terminator
    size_t bytesRead = Serial.readBytesUntil('\n', receivedBuffer,
500);
    receivedBuffer[bytesRead] = '\0'; // Ensure null-termination

    Serial.print("Received from Serial: ");
    Serial.println(receivedBuffer);

    // Parse the received string using sscanf
    // IMPORTANT: Specify max field width to prevent buffer overflows!
    // Example: %29[^&] reads up to 29 characters into a 30-byte
buffer.
  int numParsed = sscanf(receivedBuffer,
"%29[^&]&%29[^&]&%29[^&]&%29[^&]&%29[^&]&%29[^&]&%29[^&]&%29[^&]&%29[^&
]&%29[^&]&%29[^&]&%29[^&]&%29[^&]&%29s",
                           data1, data2, data3, data4, data5, data6,
data7, data8, data9, data10, data11, data12, data13, data14);

    Serial.printf("Parsed %d data segments.\n", numParsed);
    Serial.printf("data1 (TYPE): %s\n", data1);
    Serial.printf("data2 (VAL1): %s, data3 (VAL2): %s, data4 (VAL3):
%s, data5 (VAL4): %s\n", data2, data3, data4, data5);
    Serial.printf("data6 (VAL5): %s, data7 (VAL6): %s, data8 (VAL7):
%s\n", data6, data7, data8);
    Serial.printf("data9 (VAL8): %s, data10 (VAL9): %s, data11 (VAL10):
%s\n", data9, data10, data11);
    Serial.printf("data12 (VAL11): %s, data13 (VAL12): %s, data14
(VAL13): %s\\n", data12, data13 ,data14);


    // After parsing, send the data
    sendData();
    delay(100); // Small delay to prevent watchdog timer resets
  }
}
```