1. Abstract

As industrial electrical systems become increasingly complex, industrial workplaces are inherently susceptible to fire hazards due to the presence of heavy machinery, combustible materials or potential electrical equipment malfunctions. Fire incidents in such environments can result in catastrophic consequences, including property damage, loss of life, production downtime, and even environmental impacts.

To address potential fire hazards, our team developed a device that integrates multiple sensors. The device uses a machine learning core to reliably and accurately monitor and detect potential fires. Once the likelihood of a fire is confirmed, the device will sound an alarm and automatically activate the sprinkler system, extinguishing the flames at the earliest possible moment.

2. Introduction

2.1. Motivation

Fire detection systems are essential for protecting life, property, and the environment. However, existing solutions often lack the flexibility and integration required to cope with the complexity of modern fire hazards. Current systems are often limited to standalone hardware (such as smoke detectors) or expensive industrial-grade installations, which makes them inaccessible to small facilities or residential users. In addition, many such systems rely on single-sensor mechanisms, leading to delayed responses or false alarms due to the inability to analyze data from multiple sources.

The challenge of existing systems is the lack of seamless integration between sensors, real-time data processing, and active response mechanisms. For example, many fire detection systems cannot combine advanced technologies such as image recognition or machine learning to detect anomalies or effectively classify hazards. Another limitation is that existing systems are often limited by the environment, and specific systems can only serve specific environments and are difficult to apply to a variety of scenarios. This

functional limitation makes the types of systems on the market chaotic and increases the difficulty of customer purchasing decisions.

2.2. Problem Statement

To address this series of problems with existing systems, PyroShield integrates multimodal sensing, machine learning, and real-time response mechanisms into a cohesive platform to detect, analyze, and respond to fire hazards in real-time. By using multiple sensors, PyroShield improves the monitoring of various fire hazards. In the multi-sensor integration mode, users can choose to use only a single sensor or several sensors, so that the system is not restricted by the environment and can be applied in different scenarios. At the same time, the multi-sensor mechanism can analyze data from multiple sources at the same time, organize and analyze the data collected by each sensor, and monitor factors that may cause fire hazards, greatly improving the certainty of detection.

2.3. Solution Objectives

The core goal of PyroShield is to monitor potential fire risks using multiple sensors such as gas detectors (for detecting flammable gases), temperature sensors (for detecting thermal anomalies), and image recognition systems (standard cameras). At the same time, in order to solve the accuracy issues of existing systems on the market, PyroShield uses machine learning models, specifically autoencoders for temperature anomaly detection and logistic regression for gas sensor analysis, to identify abnormal conditions that may indicate the presence of fire or other fire-related hazards. By processing these sensor data in real-time, the system can detect early signs of fire and classify the severity of the threat, improving the accuracy of fire monitoring.

After detecting the presence of a fire hazard, the system will automatically respond to the source of the fire and immediately send an alert to the user through the application. When a fire is detected, PyroShield automatically triggers the sprinkler system to suppress the source of the fire in the first place to prevent the fire from spreading. By

implementing an alarm system, users can receive instant notifications via SMS, email, or app alerts detailing the severity and location of the danger. The system also allows users to manually intervene and control safety measures remotely, thereby fully controlling the functions of the system, including activating/deactivating alarms, adjusting sensor thresholds, or viewing sensor data in real-time, thereby improving user experience and operability.

3. Details of approach

3.1. System Architecture Diagram

PyroShield system is designed as an IoT-based fire hazard detection and management solution. It integrates sensor nodes, a gateway, and a cloud (emulated using a laptop) to ensure reliable and efficient hazard detection, monitoring, and response. The system architecture comprises three main components: sensor nodes powered by ESP32 microcontrollers, a gateway, and a web-based user interface. Below is the system architecture diagram:

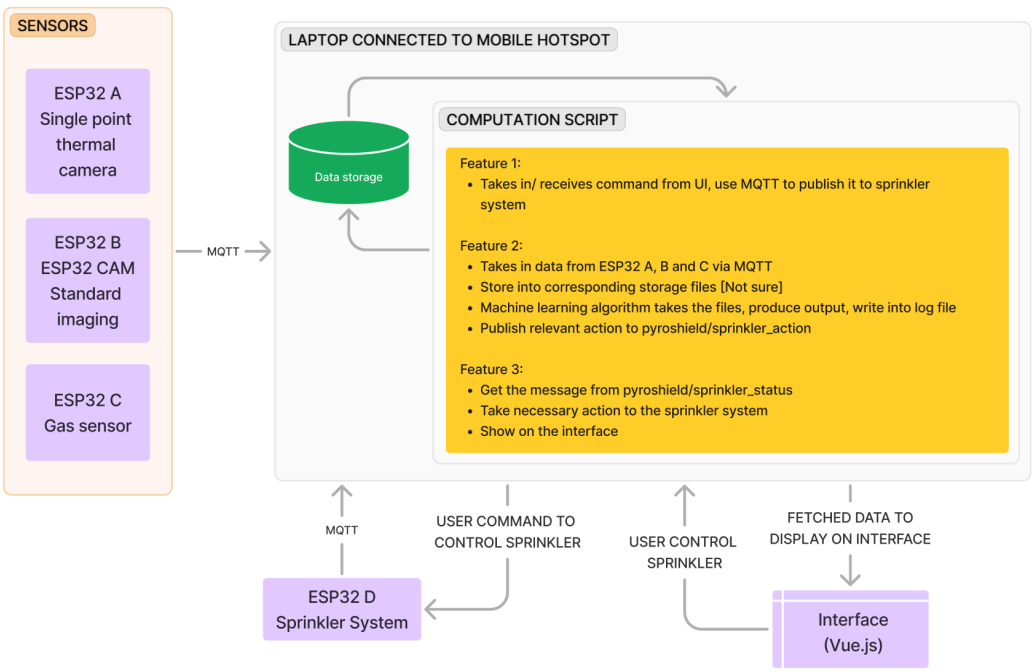

Figure 1: System architecture diagram

3.2. Data Flow and Communication

PyroShield system uses MQTT as its primary communication protocol for data exchange between the sensor nodes, gateway and cloud. MQTT was selected for its lightweight protocol design, ideal for low-power devices like ESP32. Its publish/subscribe model ensures efficient communication between multiple nodes and the gateway. MQTT provides reliable and efficient communication even over low-bandwidth networks, making it ideal for IoT applications. Topics are used to organize and distribute messages. (Details in Section 4.1.1 MQTT topics and communication logic) The MQTT used is hosted on a Mosquitto MQTT broker running on the laptop, which connects to the network via a mobile phone hotspot. ESP32 nodes use different topics to communicate with the Python script running on the laptop.

The Python script processes the incoming data and facilitates communication between the MQTT broker and the Flask backend. The Flask application, chosen for its lightweight nature and simplicity in creating RESTful APIs, acts as the system's central API and interacts with the Vue.js frontend using HTTP requests via Axios. HTTP and Axios are ideal for this setup due to their widespread support, reliability, and ease of implementation, ensuring a smooth exchange of data. (Details in Section 4.1.2 Flask backend and Vue.js frontend setup) This setup allows the Vue.js interface to dynamically fetch data, display real-time system updates, and send user commands to the backend for further processing, creating a seamless and responsive system.

3.3. Hazard detection methodology

Our hazard detection methodology makes use of two separate systems, the light intensity and temperature system and the gas sensor system.

For the light and temperature system, we made use of the *Seeed Studio XIAO ESP32S3 Sense* which captures still images of the mock-up room every 5 seconds for detecting abnormally bright objects. This ESP32 board has a OV2640 camera module attached and stores the images locally in its micro SD card. The second module used is the *Seeed Studio MLX90614 Thermal Imaging Camera*. This is a single point infrared

camera that detects the surface temperature of any object that it is pointed at. The module was chosen because a fully fledged infrared camera would be much more expensive. Instead, the module was mounted on a servo that swings it around to point at objects in order to measure their temperature. The combination of these two modules allowed for an economical alternative to using a thermal camera with high resolution.

For the gas detection system, we chose to use the MQ-2 gas sensor in the system based on the following analysis. First, MQ-2 can detect a variety of flammable gases, including common fire hazard gases such as LPG, Smoke, Alcohol, Propane, Hydrogen, Methane and Carbon Monoxide [1], it can detect fire hazards in different environments. Secondly, MQ-2 supports fast response and real-time degree functions, and supports seamless connection with microcontrollers such as ESP32, which makes it efficient in data acquisition and processing, which is essential for the ever-changing fire scene warning. In addition, MQ-2 has high sensitivity and can detect gas concentrations in the range of 200 ppm to 10,000 ppm [1], making it able to detect potential fire risks more sensitively and in a timely manner. From the market perspective, MQ-2 is low-priced, which can effectively reduce the overall cost-effectiveness of the project, thereby reducing the system selling price and improving PyroShield's market competitiveness. In general, PyroShield's multi-gas detection, efficient data processing, high sensitivity and low price make PyroShield applicable to different scenarios, breaking the industrial-level limitations of existing systems and making it also suitable for daily use, such as for small facilities or residential users.

With the above sensor nodes and machine learning model, the system is capable of identifying fire hazards and sending alerts to a Python script. The script processes the incoming data and determines whether to activate the sprinkler system based on predefined criteria (Figure 2) by sending a message via MQTT to control the sprinkler system.

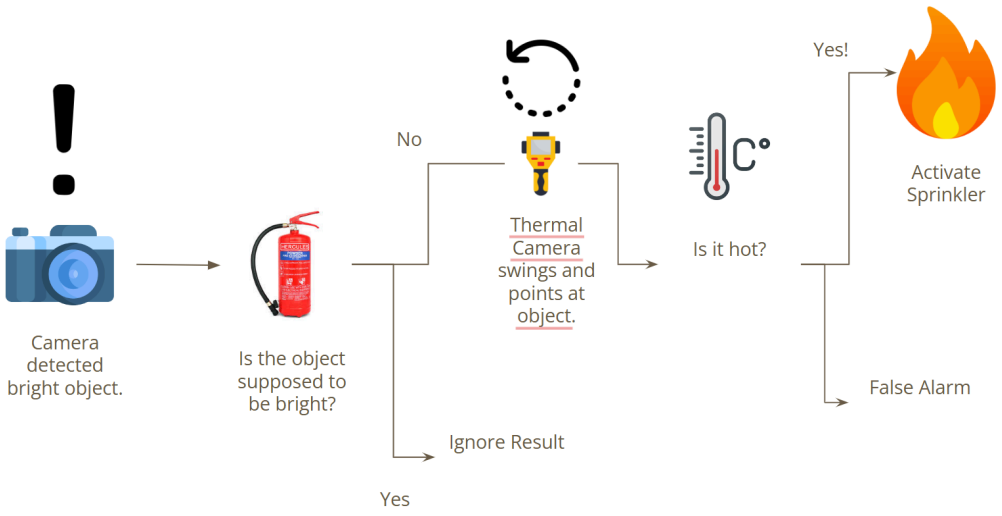

Figure 2: Decision chart combining sensor nodes and machine learning algorithm

The system employs a dual-sensor logic to minimize false alarms. When the camera identifies an object that exhibits unusual brightness, and the thermal camera confirms that the object's temperature exceeds the preset threshold, the sprinkler is triggered. Note that the thermal camera is only activated when unusually bright objects are detected. This layered logic ensures that responses are activated only when both the light intensity and temperature indicate a credible threat, significantly reducing the likelihood of false alarms. The table below describes the scenarios that differentiate false alarms from real fire situations, emphasizing the importance of integrated sensor data:

	Socket	Fire Extinguisher	Microwave	Fan	Heater
High light intensity	Y	Y	Y	Y	Y
High temperature	Y	N	N	Y	Y
Case	Fire	Normal	Normal	Fire	Fire

Table 1: Different scenarios of false alarms and real fire situations

The gas sensor node acts as an independent indicator separately, accounting for scenarios where there is no high light intensity and/or high temperature yet, but only the presence of flammable gases.

To address the challenge of simulating actual fire conditions, we incorporated images of commonly used objects used in industrial settings and used a controlled light source to simulate fire hazards. The selected objects were categorized as safe (e.g. fire extinguisher, microwave) and unsafe (e.g. heater, socket). Our approach combines object detection with environmental monitoring to assess the risk level of the scene.

For object detection, the YOLOv8 model, known for its high accuracy and speed, was selected due to its ability to perform efficient and precise object detection, making it suitable for real-time fire hazard identification. Its robustness in detecting diverse objects ensures adaptability to various scenarios. Additionally, OpenCV complements this by providing brightness measurements of detected objects, enabling us to simulate fire-like conditions effectively. This integration of object detection with environmental analysis forms the basis of our machine learning approach.

3.4. Alerts and User Interface

The sprinkler system is controlled via an ESP32 node. (Details in Section 4.2 Hardware Setup) A simple micro-submersible water pump is being used to simulate a real sprinkler, with half a plastic bottle containing water as the water reservoir. This system is alone on an ESP32 node to isolate it from the rest of the detection system as a standalone component to be built separately from the rest of the system to prevent water damage to the sensors.

The status of the sprinkler system, along with other system information can be viewed via the UI. (Details in Section 4.1 Software configuration) The user interface displays information such as detection system status, sprinkler system status as well as historical gas level trends. Therefore, the user can view alerts of activation history with relevant information such as the temperature and gas level at the point of activation. It also provides a button for the sprinkler system to be activated manually if the automation system fails.

To build the user interface, Flask is chosen to host the RESTful APIs for the system, enabling data retrieval and user interaction. This is because Flask provides a lightweight

and flexible framework for developing the backend of the web application. To run the web application's frontend which allows users to monitor and interact with the system in real-time, Vue.js was chosen for its simplicity and ability to build dynamic, responsive user interface.

4. Implementation Details

4.1. Software Configuration

4.1.1. MQTT topics and communication logic

MQTT is used to facilitate communication between devices, ensuring a reliable and efficient exchange of data. The system's subscribers and publishers include:

1. **Laptop A**: Emulates the cloud and hosts the main logic Python script responsible for the system's functionality. It processes sensor data, evaluates fire hazards, and publishes commands to other devices, such as the sprinkler system.
2. **Laptop B**: Emulates another cloud node running the machine learning script.

In the intended setup, these two roles would typically be combined into a single cloud server or machine to centralize logic and ML processing for simplicity and improved system performance. However, for this project, the roles were split across two laptops so that team members could work on the scripts separately.

Multiple topics are defined to separate message channels and ensure a streamlined data flow. The table below describes all the MQTT topics used in this system and their corresponding usage:

Topic	Usage (Message)	From	To
pyroshield/sprinkler_action	Turn on or off the sprinkler (ON or OFF)	Laptop A (Main logic)	ESP32
pyroshield/sprinkler_status	State of sprinkler (Sprinkler activated or Sprinkler	ESP32	Laptop A (Main logic)

	deactivated)		
pyroshield/danger	Safety status based on ML output (without considering gas sensor) True or False	Laptop B (ML)	Laptop A (Main logic)
pyroshield/temp	Thermocamera reading 23.97	ESP32 Thermocamera	Both Laptop A (Main logic) and B (ML)
pyroshield/gasLevel	Gas sensor reading 82	ESP32 Gas sensor	Laptop A (Main logic)
pyroshield/cam_image	Image captured from camera	ESP32 Camera	Both Laptop A (Main logic) and B (ML)

Table 2: MQTT topics and the corresponding usage

4.1.2. Flask backend and Vue.js frontend setup

The Python script used for the backend is responsible for processing data received via MQTT. It also uses Flask to create routes and manage REST API endpoints to retrieve sensor data and handle requests from the Vue.js frontend (e.g., activating the sprinkler). Axios in Vue.js is used to send HTTP requests to Flask. (Figure 3) Cross-Origin Resource Sharing (CORS) is configured in Flask to allow communication between the backend and the Vue.js interface hosted on a different port or domain.

The Flask API provides several endpoints for the Vue.js frontend:

- **/api/status**: Returns the current sprinkler status.
- **/api/activate**: Manual activation or deactivation of the sprinkler via MQTT.
- **/api/messages**: Fetches the last 10 sprinkler state change messages for display.
- **/api/detection**: Returns the current safety status, temperature and gas levels.
- **/api/gas_history**: Retrieves historical gas level data from the log file.

The Vue.js frontend serves as the primary interface for monitoring and controlling the PyroShield system. (Figure 4) Users can view real-time data on temperature, gas levels, and system status. The latest 10 alert notifications are also displayed. The UI also includes control options for overriding automation, such as manually activating or deactivating the sprinkler system using the button which sends HTTP requests to the

Flask backend. This ensures that users remain in control while benefiting from the system's automation capabilities. If the user presses the button to manually toggle the sprinkler status, a confirmation dialogue will pop up in case of accidental press.

4.1.3. Data Storage and Logging

The Python script logs data to various files with entries timestamped for easy retrieval. For example, it appends gas level data to a file named gas_level.log every time a new data point is received. Flask provides endpoints to query historical data for visualization using the Vue.js frontend. Vue.js fetches historical data and displays it in a real-time chart, providing insights into historical trends.

The script sets up loggers for systematic tracking of sprinkler status, safety state, temperature, gas levels, and captured images. This ensures that data from all MQTT topics is logged for debugging or historical analysis. (Figure 5)

4.2. Hardware Setup

The following section details the source code used for each of the components of our system.

ESP32 Camera

On setup, the ESP32S3 connects to wifi and the MQTT broker. (Figure 6) After which it enters a simple loop which takes a picture with the camera every 5 seconds and saves it in its microSD card. (Figure 7)

Each time an image is taken and saved locally, it also transmits the jpeg image as a base64 string to the MQTT client through the pyroshield/cam_image channel. (Figure 8)

Infrared Camera Turret

This Arduino code controls the servo and the infrared camera as part of the temperature detecting turret. The code listens for a message through command/esp2, which tells the location at which the camera should point. After which, it swings the servo to the

specified position and triggers the camera to get a single reading. This reading is encoded and sent to the MQTT client through the pyroshield/temp channel. (Figure 9)

MQTT Client

The MQTT Client listens for two topics, pyroshield/cam_image and pyroshield/temp. If it receives a camera image, it decodes the message first as a utf-8 string. This string is then decoded again using the base64 library into a jpeg image. This is then saved with the corresponding time the image was taken. After which, the machine learning algorithm is used to check for brightness on the image saved. (Figure 10)

If the machine learning algorithm detects an abnormally bright object, the client will send a message through command/esp2 to the infrared camera turret. The message contains an integer that tells the turret which object to point at. (Figure 11)

At this point, the client is expecting a returning message from the infrared camera which consists of the temperature reading of the object. If it is beyond a certain threshold, the client will finally broadcast a “True” on the pyroshield/danger channel, which will be picked up by Laptop A in order to trigger the sprinkler system. In reality, the threshold should be 100 to 130 degree celsius [2].

Gas Sensor

The underlying logic of the gas sensor is to set a threshold. When the detected gas level exceeds the threshold, the alarm information is printed to the monitor, and the communication is carried out via Wi-Fi and the data is transmitted via MQTT. In PyroShield, to ensure the accuracy of the detected data, the average value is calculated after multiple readings are set in the loop() function. If the average gas level exceeds the predefined threshold, an alarm message will be displayed on the serial monitor, indicating that the gas level is too high and may cause danger. (Figure 12)

Sprinkler System

When the sprinkler system initially connects to Wi-Fi, necessary for establishing an MQTT connection, LED 1 will light up to indicate a successful connection. The ESP32

node connected to the sprinkler system subscribes to the MQTT topic, “pyroshield/sprinkler_action”. Upon MQTT connection, the system will listen to it for incoming control messages ("ON" or "OFF"), which trigger the corresponding activation or deactivation functions. (Figure 13) When it receives a command, it will toggle the state of the sprinkler. Then, it will publish a message ("Sprinkler activated" or "Sprinkler deactivated") to the MQTT topic, “pyroshield/sprinkler_status”, and update LED 2 to reflect the activation status of the sprinkler. (Figure 14)

The micro submersible pump requires connection to a relay and an external battery as the pump operates at 5V. The relay acts as a switch to safely control the pump using the low-power GPIO signals from the ESP32. At the same time, the external battery supplies the necessary power for the pump's operation. (Figure 15)

Machine Learning Implementation

To implement the object detection system, we utilized the YOLOv8 model for object classification and brightness analysis. For Dataset Preparation, a total of 300 images containing objects such as sockets, fire extinguishers, microwaves, fans, and heaters were manually labeled to create annotations for training. In addition, the ‘light’ object is created to identify the light source and perform brightness calculations. The dataset was split into 80% for training, 10% for validation, and 10% for testing. The YOLOv8 model “yolov8n.pt” was trained for 150 epochs to achieve high accuracy. The resulting trained model was exported in the ONNX format for deployment. (Figure 16)

In the separated python file, the trained YOLO model was loaded with task='detect'. Safe and unsafe item categories were defined. The safe items are fan, fire extinguisher and microwave. The unsafe items are socket and heater. In the object detection process, the YOLO model detects objects in the image and outputs bounding boxes for each detected object.

In object detection and image processing, if any danger items are not detected in the image, it will be flagged as missing item and listed as potential fire hazards. When missing danger items are found and a light source is detected, the system calculates

the average brightness in the light's region by converting the region of interest to grayscale and computing the mean pixel value. A red bounding box highlights the region in which the object is on fire. A green bounding box is drawn in the region where the object is safe, their class names and coordinates are stored.

In the terminal, the detected items and unsafe items, the notification for fire detection are displayed if the conditions (danger items and high brightness) are met. In addition, the coordinates of fire and the brightness value will be displayed. (Figure 17) The processed image with bounding boxes is displayed using OpenCV. (Figure 18)

4.3. Power Management

In the early stages of the project, each component was powered directly from a laptop computer via a USB connection because each sensor was set up separately. The advantage of this power supply method is that it is simple and direct to use and can ensure stable power supply for the components, but the disadvantage is that it is less flexible.

In the final product, as each component was integrated, the PyroShield system gradually matured into a deployable product. Its battery management was also upgraded accordingly. We changed the power supply method from direct power supply to portable battery power supply, using a rechargeable power bank to power the components. This power supply mode greatly improves the flexibility of the device and is very helpful for remote and adaptable deployment in various practical scenarios. At the same time, in order to optimize the system power, the components use a threshold-based activation method to keep the unused sensors in a low-power state. It will only be fully activated when the data indicates that there is a potential fire, ensuring that energy is only used when necessary. This optimization of battery management ensures that PyroShield can operate autonomously for a long time, making it suitable for applications in various scenarios such as residential and industrial fire protection.

5. Experimental Evaluation

5.1. Accuracy of Machine Learning Model

The performance of the pretrained YOLOv8 model was evaluated based on its ability to accurately detect and classify objects as safe or unsafe. During the validation phase, the model was tested on 30 images containing 180 object instances. (Figure 19)

Based on the overall performance of the model, the precision is 0.979, recall is 0.992, Mean Average Precision at 50% IoU (Intersection over Union) (mAP@50) is 0.995, Mean Average Precision at 50-95% IoU (mAP@50-95) is 0.775. (Figure 20)

The model demonstrates good precision, recall, and high mAP values, indicating reliable detection and classification of objects. The model correctly identifies the object instances from validation data, as shown in the confusion matrix (Figure 21). The results confirm the model's robustness and suitability for real-time fire hazard analysis, with precise classification contributing to effective decision-making in simulated fire scenarios.

5.2. Power consumption

Considering that our system is a real-time detection system within an enclosed environment, our camera would need a constant power supply to stay active. However, power consumption is still reduced by the slower rate of capturing images, once every 5 seconds, and the infrared camera is only activated when abnormally bright objects are detected. Such rate is only for demonstration purpose. In actual real life deployment, longer intervals will be set which further reduces the power consumption.

In PyroShield, each key component is powered by a dedicated 5000mAh power bank to ensure reliable operation and adequate energy resources.

For the MQ-2 gas sensor, it consumes approximately 800mW during active operation [1]. With a dedicated 5000mAh (18.5Wh) power bank, the estimated battery life for it is 18.5Wh / 0.8W ≈ 23 hours.

For the Seeed Studio XIAO ESP32S3 Sense, it consumes approximately 240mW during active operation and WIFI transmission and consumes approximately 70mA during idle operation [3]. Since the camera captures images every 5 seconds, it pimples a short burst of high power (active) then followed by a low power (idle). Assume to capture the images for 0.5 seconds and idle for the remaining 4.5 seconds, the Average Power = [(0.5 × 240) + (4.5 × 240)] / 5 = 87mA. With a dedicated 5000mAh power bank, the estimated battery life for it is 5000mAh / 87mA ≈ 57 hours.

For the OV2640 camera, it consumes between 120-150 mW during image capture [4]. With a dedicated 5000mAh (18.5Wh) power bank, the estimated battery life for it is 18.5Wh / ((0.12W + 0.15W) / 2) ≈ 137 hours.

For Seeed Studio MLX90614 Thermal Imaging Camera, it consumes approximately 3mW during active operation [5]. Given its low power requirements, the battery life is expected to be very long, so we can ignore its battery consumption.

6. Challenges Faced

Designing the PyroShield system

Designing the PyroShield system was a complex process that began with conceptualizing how various sensors would integrate to detect hazards and trigger appropriate actions. As we purchased a single-point thermocamera instead of the initially planned thermocamera that is capable of capturing thermal images, we had to redesign our system. This involved redesigning it to use a combination of light intensity and single-point temperature readings to achieve the same functionality.

Handling Imbalanced Classes in Image Dataset

One significant challenge in creating a machine learning model is addressing the issue of imbalanced classes within the dataset. This arises when certain object classes are overrepresented compared to others. For example, the dataset may contain a larger number of images of fans and fewer images of critical objects like heaters or sockets. The ML model trained on imbalanced datasets may develop a bias toward predicting the more frequent classes. This can cause higher detection accuracy for overrepresented objects but lower detection accuracy for underrepresented ones. Hence, we tried to ensure that all the objects in the image dataset are equally represented.

Slow transfer of images over MQTT

Another challenge that we faced was the slow rate at which the images were being transferred over through MQTT. At higher resolutions, it would take too long for the images to be transferred. To resolve this, we made it so that the images would only be taken at an interval of 5 seconds, and we used a lower resolution image of QVGA (320x240) as the mock up was small enough to fit within this frame. In reality, the resolution would need to be larger to account for a larger room size. In this case, it would be preferable to transfer photos taken over wire, or given additional computation power, to run the machine learning scripts directly on the microcontroller.

Creating the user interface

The UI must provide a clear, intuitive interface to display real-time data and effectively control the sprinkler system. Designing a responsive UI that adapts to different devices and screen sizes while presenting complex data, such as gas trends and hazard status, is challenging, especially without prior background in building a web-based interface. It requires significant time and effort to research and select a suitable tech stack, learn the necessary coding skills, set up the required configurations, and ensure everything works seamlessly together while connecting various components given the limited time.

7. Limitations of Solution

Machine Learning Model

The model's performance is closely tied to the dataset used for training and validation. If the dataset lacks diversity, such as variations in object appearances (e.g. colour, shape) or environmental conditions (e.g., lighting), the model might struggle to generalize unseen scenarios. For example, if the training dataset contains only a specific type of socket (e.g., white wall sockets), the model might fail to detect or misclassify sockets of different colours or designs in the real world. This limits the model's robustness and adaptability to real-world scenarios where object appearances can vary significantly.

The YOLOv8 model performance can degrade in non-ideal scenarios, particularly in dynamic or uncontrolled environments. Detection accuracy can be influenced by lighting conditions. Extremely bright environments can cause glare, while dim settings can obscure object features. Shadows or uneven lighting can distort object boundaries and textures, making it difficult for the model to classify objects correctly. For example, if a socket is in a shadowed corner, it might lead to incorrect or missed detections. In addition, the model is optimized for objects in standard orientations, such as upright or frontal views. Objects viewed from unusual angles or partially obscured by other objects may pose challenges. For example, if the socket is partially hidden or tilted at an angle, it may not be detected with the same accuracy as objects in standard orientations.

Camera Sensors

The attached OV2640 used has a limited resolution of 1600x1200 and field-of-view. In practice, the OV5640 could be used instead, being compatible with the ESP32S3, offering up to 2592x1944 resolution and a wider field of view.

In practice, a more complicated servo and turret would be required that could offer 360-degree rotation such that the thermal camera can point at any location in the room,

rather than a fixed range of detection. Two servos could be used in tandem to achieve such a turret. (Figure 22)

With the implementation of this turret, we could develop a system to extract the coordinates of objects from the images taken and be able to independently point the thermal camera at said object anywhere within the room.

8. Future Extension

To enhance the functionality and utility of the current system, several extensions can be implemented in future upgrades. These extensions can improve fire hazard detection, predictive capabilities, and user interaction.

Identify and Show Zones of Fire Hazard on the UI

The system can be extended to highlight fire hazard zones dynamically on the user interface. Detected objects, such as sockets and heaters, can be associated with specific risk levels (low, moderate, high) based on their properties like brightness, temperature or proximity to heat sources. A colour-coded heatmap overlay could be displayed on a floor plan or image, providing users with an intuitive visualization of high-risk areas. This feature will help prioritize inspection or maintenance in hazardous zones.

Predictive Analysis for Fire Risk Assessment

Incorporating historical data analysis can enable the system to predict which objects are at risk of causing a fire. Using a trained machine learning model from data such as object type, brightness levels, temperature, gas levels and past incidents, the system could provide real-time fire risk scores for each object. This proactive feature would enhance fire prevention by allowing users to address potential risks before they escalate into hazardous situations.

Feedback on UI for Sprinkler Trigger Events

A feedback function can be included to display which detected object led to the activation of the sprinkler system. When the system detects unsafe conditions, it can log

and display details such as the object class, location, and timestamp. For example, the UI could indicate "15:08:03: Sprinkler activated due to fire detected near socket in Zone A." This feature would provide users with clear and actionable information regarding fire hazard events.

Option to View Image Related to Sprinkler Activation

To provide visual evidence of a hazardous location, the system could store the image frame captured during a fire hazard event. This image, labelled with detected objects and bounding boxes, can be made accessible through the user interface. An image gallery could also be implemented to maintain a historical record of sprinkler activations, aiding in post-incident analysis and decision-making.

9. Conclusion

PyroShield combines multiple ESP32 microcontrollers, multiple sensors, machine learning models, and IoT technologies to create an accurate, flexible, and adaptable intelligent automated fire protection system. The system monitors multiple parameters in real-time, detects potential fire hazards more effectively through thermal imagers, gas sensors, and advanced detection algorithms, and can respond quickly by activating the sprinkler system, highlighting its potential as a comprehensive safety system.

Throughout the development process, although PyroShield met these achievements, challenges such as ESP32 memory limitations, limited camera field of view, and UI complexity also provided opportunities for future improvements. PyroShield plans to upgrade the sensor system and use more diverse data sets in the future. With the continuous research and iteration of the system, PyroShield will improve the robustness and adaptability of the system, and ultimately more effectively protect property and life in various environments.

References

[1] "MQ2 Gas Sensor with Arduino: A Complete Guide", Last Minute Engineers, [Online]. Available: https://lastminuteengineers.com/mq2-gas-senser-arduino-tutorial/. [Accessed: Nov 20, 2024]

[2] "Fire Prevention and Overheating Protection Systems", Streamer Electric, [Online]. Available: https://streamer-electric.com/products/fipres-fire-overheating-prevention/. [Accessed: Nov 20, 2024]

[3] “Getting Started with Seeed Studio XIAO ESP32S3”, Seeed Studio, [Online]. Available: https://wiki.seeedstudio.com/xiao_esp32s3_getting_started/. [Accessed: Nov 20, 2024]

[4] "OV2640 Camera Module Specifications," Arducam, [Online]. Available: https://www.arducam.com/ov2640/. [Accessed: Nov 20, 2024]

[5] Melexis, "MLX90614 Infrared Thermometer for Non-Contact Temperature Measurements," Datasheet, 2024. [Online]. Available: https://www.mouser.sg/datasheet/2/744/MLX90614_Datasheet_Melexis-1891825.pdf. [Accessed: Nov 20, 2024]

Appendices

```vue
<script>
import axios from "axios";
import DetectionInfo from "./DetectionSystem/DetectionInfo.vue";

export default {
  components: {
    DetectionInfo,
  },
  data() {
    return {
      detectionStatus: "Unknown",
      temperature: null,
      gasLevel: null,
    };
  },
  methods: {
    fetchDetectionData() {
      axios
        .get("http://localhost:5000/api/detection")
        .then((response) => {
          this.detectionStatus = response.data.danger;
          this.temperature = response.data.temperature;
          this.gasLevel = response.data.gasLevel;
        })
        .catch((error) =>
          console.error("Error fetching detection data:", error)
        );
    },
  },
  mounted() {
    this.fetchDetectionData();
    setInterval(this.fetchDetectionData, 1000);
  },
};
</script>
```

Figure 3: Using Axios in DetectionSystem.vue to fetch data by sending HTTP request

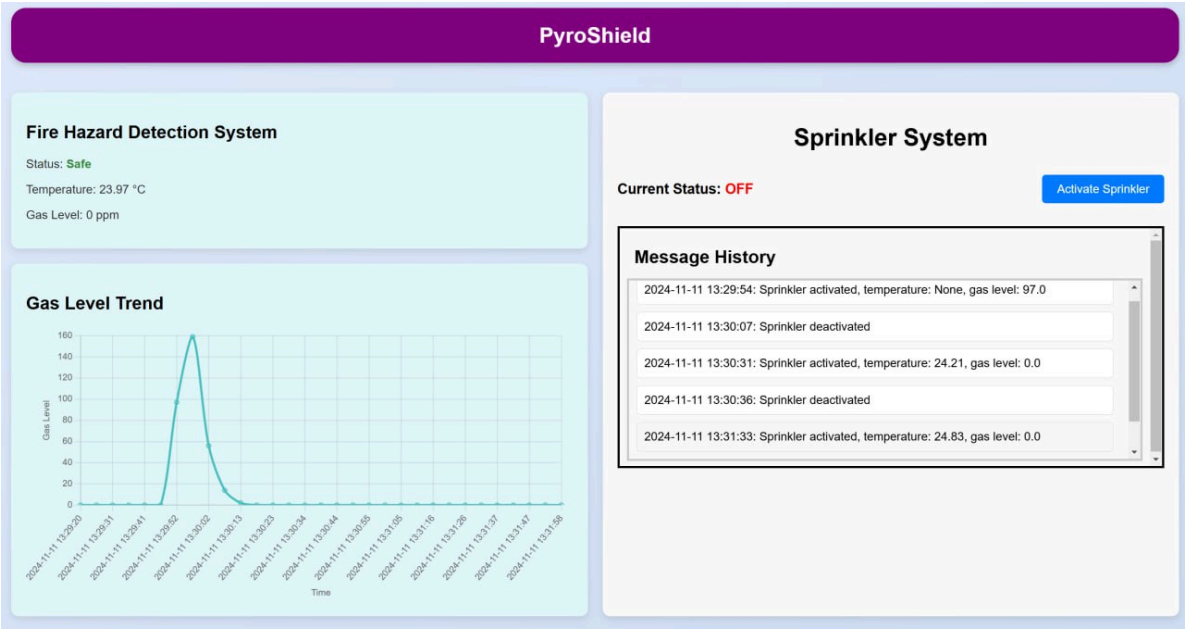

Figure 4: User interface

```python
# Set up logging for sprinkler status and detection results
logging.basicConfig(filename=os.path.join("logs", "general_logs.log"), level=logging.INFO, format='%(asctime)s - %(message)s')

# Loggers for each topic
loggers = {
    "sprinkler_status": logging.getLogger("sprinkler_status"),
    "danger": logging.getLogger("danger_status"),
    "temperature": logging.getLogger("temperature"),
    "gas_level": logging.getLogger("gas_level"),
    "captured_image": logging.getLogger("captured_image")
}

# File handlers for each logger
loggers["sprinkler_status"].addHandler(logging.FileHandler(os.path.join("logs", "sprinkler_status.log")))
loggers["danger"].addHandler(logging.FileHandler(os.path.join("logs", "danger_status.log")))
loggers["temperature"].addHandler(logging.FileHandler(os.path.join("logs", "temperature.log")))
loggers["gas_level"].addHandler(logging.FileHandler(os.path.join("logs", "gas_level.log")))
loggers["captured_image"].addHandler(logging.FileHandler(os.path.join("logs", "captured_image.log")))
```

Figure 5: Setup code for logging

```
  // MQTT
  mqttClient.enableDebuggingMessages();

  mqttClient.setURI(server);
  mqttClient.enableLastWillMessage("lwt", "I am going offline");
  mqttClient.setKeepAlive(30);

  initWifi();
  WiFi.setHostname("c3test");
  mqttClient.loopStart();
}

void initWifi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, pass);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
      Serial.print('.');
      delay(1000);
    }
    Serial.println(WiFi.localIP());
}
```

Figure 6: ESP32S3 connects to wifi and the MQTT broker

```
// Take picture
void takePicture() {
    Serial.println("Touched, take a picture");
    // Create image file name
    char imageFileName[32];
    sprintf(imageFileName, "/image%d.jpg", fileCount);

    // Take a picture
    photo_save(imageFileName);
    Serial.printf("Saving picture: %s\r\n", imageFileName);

    fileCount++;
}

void loop() {

  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }

  // Make sure the camera and MicroSD are ready
  if (camera_status && sd_status) {
    takePicture();

    delay(5000);
  }
}
```

Figure 7: Take picture every five seconds and saves in microSD card

```
// Save pictures to SD card
void photo_save(const char *fileName) {
  // Take a photo
  camera_fb_t *fb = esp_camera_fb_get();
  if (!fb) {
    Serial.println("Failed to get camera frame buffer");
    return;
  }
  // Save photo to file
  // Write file and encode as base64 string
  // Then send it off
  if (writeFile(SD, fileName, fb->buf, fb->len)) {

    String base64Img = base64::encode(fb->buf, fb->len);
    //String base64Img = "Hello";
    //Base64.encode(base64Img, fb, fb->len);

    Serial.println("Sending payload...");
    mqttClient.publish(sendTopic, (String)base64Img);
  }

  // Release image buffer
  esp_camera_fb_return(fb);

  Serial.println("Photo saved to file");
}


// SD card write file
bool writeFile(fs::FS &fs, const char *path, uint8_t *data, size_t len) {
  Serial.printf("Writing file: %s\r\n", path);

  File file = fs.open(path, FILE_WRITE);
  if (!file) {
    Serial.println("Failed to open file for writing");
    return false;
  }
  if (file.write(data, len) == len) {
    Serial.println("File written");
  } else {
    Serial.println("Write failed");
    return false;
  }
  file.close();
  return true;
}
```

Figure 8: Transmits the jpeg image as a base64 string to MQTT client

```
void swingToTarget(int target) {
  int curr = myservo.read();
  int next;
    while (true) {
      if (abs(target-curr)<=step){ // Reaching target
        myservo.write(target); // Set to target
        delay(delayTime);
        return;
      }
      // Else step to target:
      next = (target - curr) / abs(target - curr) * step; // Get direction
      //Serial.println(curr + next);
      curr = curr + next;

      myservo.write(curr);
      delay(delayTime);
    }

}

// And send payload to server
void checkTemp() {
  double ambient = mlx.readAmbientTempC();
  double temp = mlx.readObjectTempC();

  Serial.println("Ambient = " + (String)ambient + "*C");

  //String message = "Object = " + (String)temp + "*C";
  String message = (String)temp;
  Serial.println("Sending payload: " + message);

  mqttClient.publish(sendTopic, message);


}
```

Figure 9: Arduino code for the servo and the infrared camera

```python
def on_message(client, userdata, message):
    if message.topic == "pyroshield/temp":
        temp_reading = message.payload.decode("utf8")
        print("Received message: " + temp_reading)

        if float(temp_reading) > 28.0:
            print("Triggering!")
            client.publish("pyroshield/danger", "True")
        else:
            print("Low temperature.")
            client.publish("pyroshield/danger", "False")

    if message.topic == "pyroshield/cam_image":
        base64_image = message.payload.decode("utf8")
        print("Received message.")

        image_data = base64.b64decode(base64_image)

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        file_name = f"image_{timestamp}.jpg"
        file_path = os.path.join(save_dir, file_name)

        # Save the image to a file
        with open(file_path, "wb") as image_file:
            image_file.write(image_data)

        print("Image was saved as " + file_name)


        print("Sending acknowledgement...")
        client.publish("command/esp", "Message received.")

        ml_algo(file_path)
```

Figure 10: MQTT Client listens for two topics

```python
    # Print missing unsafe items
    if missing_unsafe_items:
        print(f"Danger (unsafe) items: {missing_unsafe_items}")

    else:
        print("Danger (unsafe) items: null")
        client.publish("pyroshield/danger", "False")

    if 'socket' in missing_unsafe_items:
        print("Sending message.")
        client.publish("command/esp2", 5)

    if 'heater' in missing_unsafe_items:
        print("Sending message.")
        client.publish("command/esp2", 1)

    if 'fan' in missing_unsafe_items:
        print("Sending message.")
        client.publish("command/esp2", 2)

    # Print 'Detected fire' only if missing unsafe items are present
    if 'socket' in missing_unsafe_items or 'heater' in missing_unsafe_items or 'fan' in missing_unsafe_items:
        if max_light_coords:
            x1, y1, x2, y2 = max_light_coords
            print(f"Detected fire at [{x1}, {y1}, {x2}, {y2}]: average brightness {max_light_brightness:.2f}")
```

Figure 11: Send message to infrared camera turret when abnormally bright

```
void loop() {
  int sum = 0;

  // Take 10 readings and calculate the sum
  for (int i = 0; i < 5; i++) {
    int reading = analogRead(analogPin);
    sum += reading;
    delay(50);  // Small delay between readings to avoid fast sampling
  }

  // Calculate the average reading
  int averageValue = sum / 5;

  // Print the average to the Serial Monitor
  Serial.print("Average Gas Level: ");
  Serial.println(averageValue);

  // Check if the average gas level exceeds the threshold
  if (averageValue > threshold) {
    Serial.println("Alert: Average gas level is above the threshold!");
  }
```

Figure 12: Arduino code to get the average gas level

```
// Function to activate the sprinkler system
void activate() {
  if (sprinklerActivate == false) {
    digitalWrite(RELAY_PIN, HIGH);  // Turn on relay (on water pump)
    sprinklerActivate = true;
    stateChanged = true;
    Serial.println("Sprinkler ACTIVATED");
  }

}

// Function to deactivate the sprinkler system
void deactivate() {
  if (sprinklerActivate == true) {
    digitalWrite(RELAY_PIN, LOW);   // Turn off relay (off water pump)
    sprinklerActivate = false;
    stateChanged = true;
    Serial.println("Sprinkler DEACTIVATED");
  }
}

// Function that handles incoming control message
void onMqttConnect(esp_mqtt_client_handle_t client) {
  if (mqttClient.isMyTurn(client)) { // can be omitted if only one client
    mqttClient.subscribe(subscribeTopic, [](const String &payload) {
      String action = String(payload.c_str());
      (action == "ON") ? activate() : (action == "OFF") ? deactivate() : (void)0;
    });
  }
}
```

Figure 13: Arduino code to activate or deactivate sprinkler upon receiving command

```
void loop() {

  // Send message to notify sprinkler status change
  if (stateChanged) {
    String msg = sprinklerActivate ? "Sprinkler activated" : "Sprinkler deactivated";
    mqttClient.publish(publishTopic, msg, 0, false);
    Serial.println("Publishing message: " + msg + " to " + publishTopic);

    // Change LED state according to sprinkler status
    // same as digitalWrite(LED, sprinklerActivate ? HIGH : LOW);
    digitalWrite(LED, sprinklerActivate);

    stateChanged = false;
  }
}
```

Figure 14: Arduino code to send a notification when sprinkler state changed

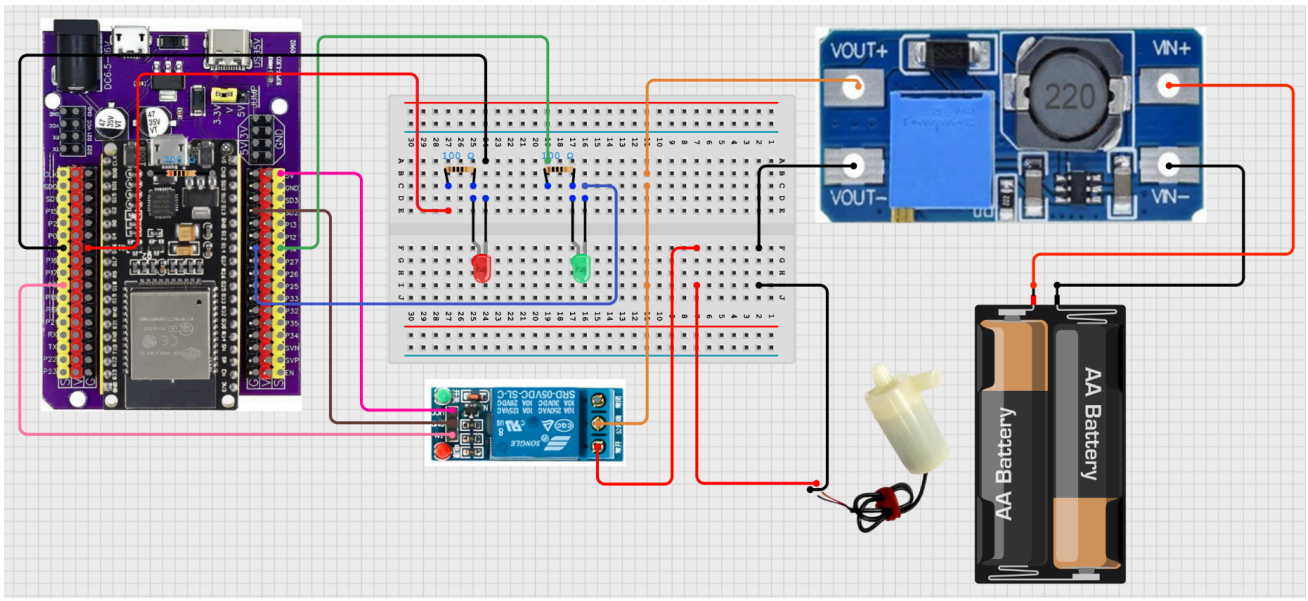

Figure 15: Circuit diagram of the sprinkler system

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
146/150         2.2G     0.5912     0.5037     0.8997          9        640: 100%|██████████| 4/4 [00:00<00:00,  5.31it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  9.57it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
147/150         2.2G     0.5899     0.5225     0.8888          9        640: 100%|██████████| 4/4 [00:00<00:00,  5.40it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00, 10.29it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
148/150         2.2G     0.6229     0.5141     0.9137          9        640: 100%|██████████| 4/4 [00:00<00:00,  5.22it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00, 15.91it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
149/150         2.2G     0.5411      0.479     0.9074         10        640: 100%|██████████| 4/4 [00:00<00:00,  4.06it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  5.05it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
150/150         2.2G     0.5676     0.4837     0.8926         10        640: 100%|██████████| 4/4 [00:01<00:00,  3.88it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  4.23it/s]
```

Figure 16: Training of YOLO model at 150 epochs

```
Speed: 15.6ms preprocess, 46.9ms inference, 15.6ms postprocess per image at shape (1, 3, 640, 640)
Detected items: ['fire extinguisher', 'microwave', 'heater', 'fan']
Danger (unsafe) items: ['socket']
Detected fire at [17, 95, 71, 146]. Average brightness: 222.29
```

Figure 17: Output showing detected and danger items

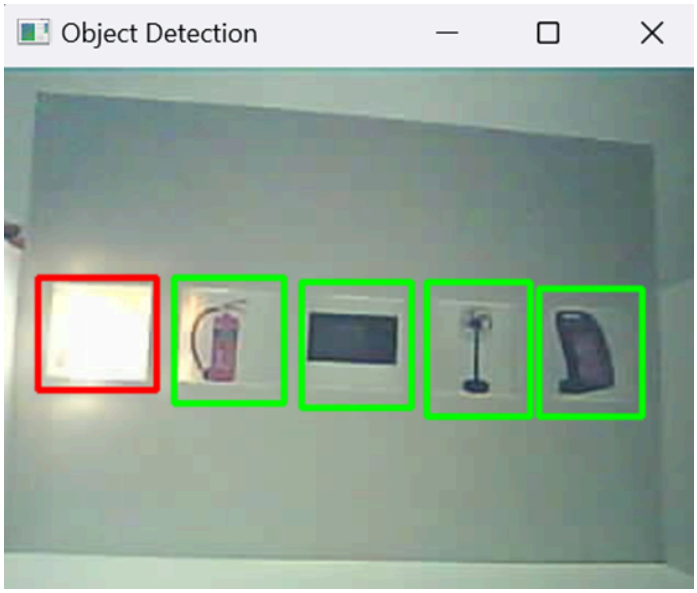

Figure 18: Corresponding processed image with bounding boxes

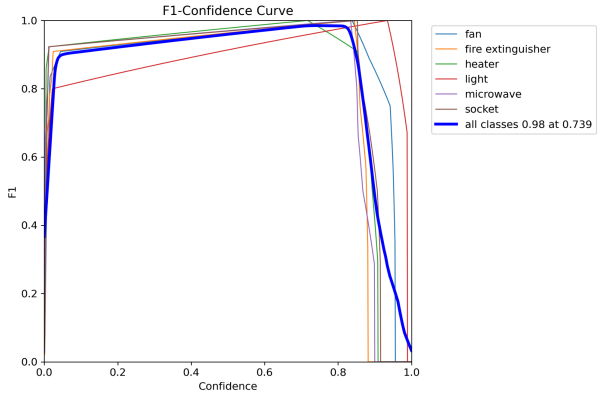

Figure 19: F1-Confidence Curve

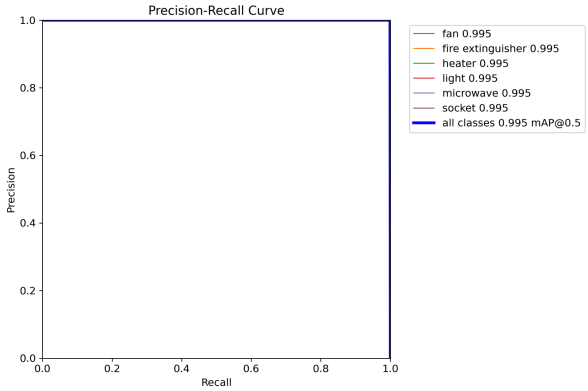

Figure 20: Precision-Recall Curve

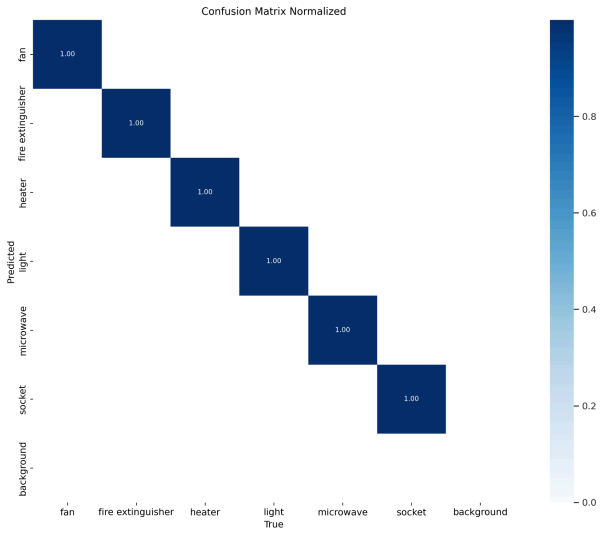

Figure 21: Confusion Matrix of Model

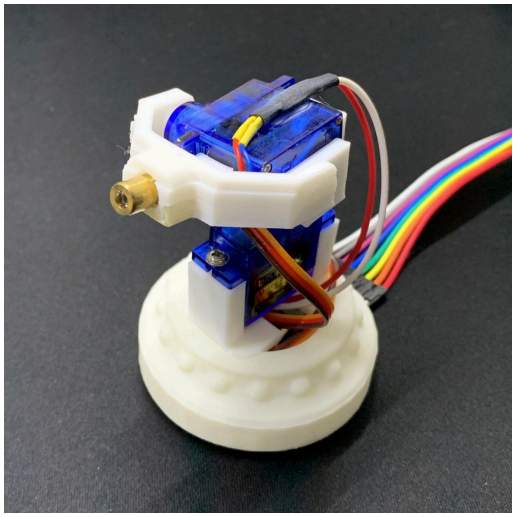

Figure 22: Implementation of turret