

MINISTRY OF EDUCATION AND TRAINING

FPT UNIVERSITY

-----



**FPT UNIVERSITY**

**GRADUATION THESIS**

**ENHANCE EDGE COMPUTING  
SURVEILLANCE SYSTEM USING  
IP CAMERAS AND EDGE DEVICES**

MAJOR: SOFTWARE ENGINEERING

**COUNCIL:            COMPUTER ENGINEERING**

**INSTRUCTOR 1: Dr. LE TRONG NHAN**

**INSTRUCTOR 2: Dr. HO HAI VAN**

Student: Ho Ngoc Dinh    dinh23mse23104

Ho Chi Minh City, Dec 2024

## **COMMITMENT**

We pledge that this project is based on our supervisors' ideas and knowledge. Not all studies and data have been published. The references, numbers and statistics are reliable and honest. The group completed the thesis requirements set by the Computer Science and Engineering faculty - department of Computer Engineering.

Sincerely,

**Ho Ngoc Dinh**

## **Abstract**

Edge computing has become a popular trend in processing data at the source, minimizing latency, and improving system efficiency. This research presents an approach to develop an enhanced edge computing surveillance system, integrating with IP cameras and edge devices to provide real-time monitoring and doing analysis. This proposed system integrates with Internet of Things (IoT) devices to manage and interact with IP cameras, actively events detect directly on edge devices. By processing data locally, the system is quicker in response times and also reducing need for cloud-based services.

The architecture combines edge devices (IoT devices) for monitoring with RabbitMQ service for high-throughput detected events streaming, and MinIO for images and videos storage. Additionally, it helps to solve challenges like difficult to customize detecting events, privacy concerns, and scalability, making it suitable for variety applications, including industrial monitoring, large areas surveillance and home security.

This solution not only significantly reduces latency in events detection but also allows to customize algorithms for detecting events in surveillance systems. This work demonstrates to the growing field of edge computing by presenting an efficient, scalable, flexible and privacy enhanced framework for modern surveillance needs.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Background . . . . .	5
1.1.1	History of surveillance . . . . .	5
1.1.2	The current state of modern surveillance system . . . . .	6
1.2	Research Question . . . . .	7
1.3	Problem Definition . . . . .	7
1.3.1	Current Issues . . . . .	7
1.3.2	Research Objectives . . . . .	8
<b>2</b>	<b>Literature Review</b>	<b>10</b>
2.1	Overview of Existing IoT Frameworks in Surveillance . . . . .	10
2.1.1	Kyungroul Framework . . . . .	10
2.1.2	Mei Kuan Lim et al.'s Framework . . . . .	12
2.1.3	YOLO Version 8 (YOLOv8) . . . . .	14
<b>3</b>	<b>Objectives and Research Goals</b>	<b>17</b>
3.1	Primary Objectives . . . . .	17
3.1.1	Scalable And Secured System . . . . .	17
3.1.2	Stable And Real-time Monitoring System . . . . .	18
3.1.3	Multi-types Data Management . . . . .	19
3.2	Technical Requirements . . . . .	20
3.2.1	Stable And Real-time Performance Metrics . . . . .	20
3.2.2	Scalability And Security Requirements . . . . .	20
3.2.3	Multi-types Data Management . . . . .	21
3.3	Success Criteria . . . . .	21
<b>4</b>	<b>Technology Evaluation and Comparison</b>	<b>23</b>
4.1	Edge Computing for Surveillance Task . . . . .	23
4.1.1	Edge Computing Architecture . . . . .	23
4.1.2	Cloud Computing Architectures . . . . .	24
4.1.3	Factors to Consider . . . . .	24

4.1.4	Edge Computing Technologies . . . . .	25
4.1.5	Edge-Cloud Approach . . . . .	26
4.1.6	Solution for IoT Surveillance Monitoring System . . . . .	26
4.2	Message Queue: RabbitMQ vs. Apache Kafka . . . . .	26
4.2.1	RabbitMQ . . . . .	26
4.2.2	Apache Kafka . . . . .	27
4.2.3	Factors to Consider . . . . .	27
4.2.4	RabbitMQ for IoT Surveillance Monitoring System . . . . .	28
4.2.5	Other Message Queue Technologies . . . . .	29
4.3	RDBMS Databases: MySQL . . . . .	29
4.3.1	MySQL . . . . .	29
4.3.2	Factors to Consider . . . . .	30
4.3.3	MySQL for IoT Surveillance Monitoring System . . . . .	30
4.3.4	RDBMS Database Alternatives . . . . .	31
4.4	Object Storage: MinIO vs. AWS S3 . . . . .	32
4.4.1	MinIO . . . . .	32
4.4.2	AWS S3 . . . . .	33
4.4.3	Factors to Consider . . . . .	33
4.4.4	MinIO for IoT Surveillance Monitoring System . . . . .	34
<b>5</b>	<b>Methodology</b>	<b>35</b>
5.1	System Architecture Design . . . . .	35
5.1.1	IoT Hardware . . . . .	35
5.2	Message Queue . . . . .	36
5.2.1	RabbitMQ Implementation . . . . .	36
5.3	MinIO - Object Storage . . . . .	36
5.4	Solution System Architecture . . . . .	37
5.5	Workflow Testing Methodology . . . . .	38
5.6	Solution System Workflow . . . . .	39
5.7	Customzied Algorithms . . . . .	40
<b>6</b>	<b>Implementation</b>	<b>41</b>
6.1	IoT Device Setup . . . . .	41
6.1.1	Hardware Configuration . . . . .	41
6.1.2	Detection Software . . . . .	41
6.2	Message Queue and Object Storage Configuration . . . . .	44
6.2.1	RabbitMQ Setup . . . . .	44
6.3	Central Service Implementation . . . . .	45
6.3.1	MySQL database . . . . .	45

6.3.2	Tables Schema . . . . .	45
6.3.3	Handle Detections from IoT Devices . . . . .	47
6.3.4	Handle Alert Signal from IoT Devices . . . . .	47
<b>7</b>	<b>Results and Analysis</b>	<b>49</b>
7.1	Algorithm Performance Metrics . . . . .	49
7.1.1	Algorithm Models Performance . . . . .	49
7.2	Workflow Testing . . . . .	50
7.2.1	Authentication and Authorization . . . . .	50
7.2.2	Add trained AI models . . . . .	51
7.2.3	Add customized algorithms . . . . .	52
7.2.4	Add IP cameras . . . . .	53
7.2.5	Add IoT devices . . . . .	54
7.2.6	Deploy/Stop Model . . . . .	55
7.2.7	Receive data from IoT devices . . . . .	55
7.3	Key Findings . . . . .	56
<b>8</b>	<b>Discussion</b>	<b>57</b>
8.1	Results Analysis . . . . .	57
8.1.1	Architectural Analysis . . . . .	57
8.1.2	Performance of Customized Models . . . . .	57
8.2	Comparative Analysis with Existing Solutions . . . . .	58
8.2.1	Architectural Advantages . . . . .	58
8.3	Lessons Learned . . . . .	59
8.3.1	Implementation Challenges . . . . .	59
8.4	Future Improvements . . . . .	59
8.4.1	Technical Enhancements . . . . .	59
8.4.2	Architectural Improvements . . . . .	60
8.5	Research Limitations . . . . .	62
8.5.1	Technical Limitations . . . . .	62
8.5.2	Methodology Constraints . . . . .	62
<b>9</b>	<b>Conclusion</b>	<b>63</b>
9.1	Summary of Key Achievements . . . . .	63
9.2	Research Impact . . . . .	64
9.2.1	IoT Surveillance Monitoring Implications . . . . .	64
9.3	Future Research Directions . . . . .	65
9.3.1	Technical Enhancements . . . . .	65
9.3.2	Recommended Research Areas . . . . .	66
9.4	Final Remarks . . . . .	67

9.4.1	Closing Recommendations . . . . .	67
<b>10</b>	<b>Improvements</b>	<b>69</b>
10.1	Enhanced Scalability with Kubernetes . . . . .	69
10.2	Hybrid Architecture for Surveillance Monitoring . . . . .	69
10.3	Enhance Security with Keycloak . . . . .	70
10.4	Stream Processing with Apache Spark . . . . .	70
10.5	System Logs with Signoz . . . . .	71
10.6	Manage Time-Series Data with ClickHouse . . . . .	71
10.7	CI/CD Pipeline with Gitlab CI . . . . .	71
<b>A</b>	<b>Technical Specifications</b>	<b>6</b>
A.1	Hardware Components . . . . .	6
A.1.1	IP Camera Specifications . . . . .	6
A.1.2	Computing Hardware . . . . .	6
<b>B</b>	<b>Software Configuration</b>	<b>7</b>
B.1	System Software . . . . .	7
B.1.1	Operating System Configuration . . . . .	7
B.1.2	Development Environment . . . . .	7
B.2	Central Service Configuration . . . . .	8
B.2.1	Server Settings . . . . .	8
<b>C</b>	<b>Code Samples</b>	<b>9</b>
C.1	Detect with Customized Algorithms Scripts . . . . .	9
C.2	Performance Testing . . . . .	17
<b>D</b>	<b>Technologies Used</b>	<b>20</b>
D.1	Hardware . . . . .	20
D.2	Software . . . . .	20
D.3	Services Communication . . . . .	21
D.4	Object Storage . . . . .	21
D.5	Databases . . . . .	22
<b>E</b>	<b>Project Timeline</b>	<b>23</b>
E.1	Develop Outline (05/10/2024 - 20/10/2024) . . . . .	23
E.2	Design (21/10/2024 - 31/10/2024) . . . . .	24
E.3	Development (01/11/2024 - 03/12/2024) . . . . .	24
E.4	Testing, Maintenance (04/12/2024 - 20/12/2024) . . . . .	25
E.5	Reporting (21/12/2024 - 31/12/2024) . . . . .	25
E.6	Gantt Chart . . . . .	26

# Chapter 1

## Introduction

### 1.1 Background

#### 1.1.1 History of surveillance

Nowadays, security in general is always a dilemma for society, especially in the surveillance domain which protects people, assets, etc. Without this exception, public places such as airports, construction sites, stations, etc. also have these concerns in the safety management process.

Evidence indicates a possible origin of surveillance technology in the Soviet Union around 1927, during the Stalin era. The Russian physicist Léon Theremin, best known for inventing the theremin (an early electronic musical instrument), is believed to have potentially created a closed-circuit system using a camera and television. However, this technology was reportedly classified by the Kremlin. A further fifteen years passed before the established development of a functional surveillance camera. Publicly available CCTV security cameras first emerged in 1949. These early models, produced by the American company Vericon, lacked recording capabilities and necessitated continuous monitoring.

Takes airport surveillance system as an example, throughout the history of aviation, airports have often faced complex security challenges requiring advanced surveillance and management systems. Aviation security has evolved from minimal measures in the pre-1960s, almost focused on preventing simple crimes, to more advanced cases due to rising



threats like hijackings in the 1960s and 1970s. During the 1970s, there was a significant improvement with the introduction of metal detectors and X-ray machines. In 2010, full-body scanners were introduced to detect non-metallic threats. More and more advanced equipment, procedures, etc. were invented to improve the quality of the security inside and outside of airports.

### **1.1.2 The current state of modern surveillance system**

Yet, the criminal complexity also developed over time, which requires sophisticated technological solutions. Lots of security systems still rely on a combination of human surveillance, physical checkpoints, and various monitoring technologies. This might prevent the ability to provide real-time, overall situations across the entire society.

Internet Protocol (IP) cameras are 1 of the most common equipment used in surveillance tasks due to their numerous advantages over traditional systems such as integrating with network infrastructure seamlessly, scalability and flexibility in deployment, also ability to remote access and control, etc. With those advancements, most of the common areas are using IP cameras in security management, typical is Changi Airport with over 2,000 CCTV implemented all over the airport.

As mentioned above, there are complex criminal cases that need complex solutions. Combined with IP cameras, the development of custom detection algorithms is crucial for addressing the specific security needs of social spaces. Some of the algorithms could be:

- Suspicious behavior patterns
- Abandoned objects
- Unauthorized access to restricted areas
- Crowd density and flow analysis
- License plate recognition for vehicle tracking

The combination of computer vision algorithms and IP cameras has shown promising results in developing a robust surveillance system. [1] With complex algorithms, IP cameras

alone could not cope with because of the heavy calculation with limited hardware capability. With the development of technology, the “Edge Computing” idea was born to solve this problem. By using some special devices or cloud services in the middle between the IP cameras and the security servers, the computation is much lighter and reduces the weight of the network and system.

However, to guard against hacking through cloud services, many organizations choose for closed-circuit infrastructure, making external attacks more difficult. This is why they prefer edge devices and wired connections over cloud services or wireless connections.

On a small scale with just a few dozen IP cameras and edge devices, integration is straightforward. But at the large-scale level, where devices number in the hundreds or even thousands, each potentially running specific algorithms, management becomes a massive challenge for the entire system.

Consequently, a system that can seamlessly integrate devices with the central hub is a logical requirement for medium to large open spaces. Our proposed solution aims to address this issue: An enhanced edge computing surveillance system (using cameras) with an auto-deployment algorithm system integrated into edge devices.

## **1.2 Research Question**

What could be done to develop a system that makes use of real-time awareness, integrates surveillance equipment like IP cameras, and permits the deployment of customized detection algorithms more effectively, improving supervisor and management efficiency in the open spaces?

## **1.3 Problem Definition**

### **1.3.1 Current Issues**

Public places are currently facing surveillance efficiency issues due to:

- Processing latency: The time to transmit and process data from cameras to cloud

servers can cause delays, reducing the ability to respond promptly.

- **Bandwidth load:** The large amount of data from IP cameras requires high bandwidth, putting pressure on network infrastructure.
- **Customization capability:** Current systems often lack flexibility in deploying and updating new detection algorithms.

### **1.3.2 Research Objectives**

#### **Enhance Surveillance Security and Safety through Advanced Video Analytics**

- Implement an IP camera-based surveillance system with edge computing capabilities.
- Create a real-time alert system for immediate response to potential security and safety incidents.
- Implement facial recognition and object detection capabilities to identify suspicious activities such as people loitering in restricted areas, unattended baggage, etc.

#### **Improve Operational Efficiency via Real-time Monitoring and Analysis**

- Implement automated alert and tracking of detected objects to improve real-time awareness and provide timely decisions.
- Monitor restricted areas to check if any person or vehicles enter these areas.

#### **Implement a secure, real-time data transmission and processing infrastructure**

- Develop strong encryption protocols for all data transmissions between cameras, edge devices, and the central system.
- Optimize data streaming and processing techniques to minimize latency and ensure real-time performance.

- Design a scalable server architecture to handle incoming alerts and data from multiple edge devices smoothly.

Develop a central system for management tasks

- Deploying and updating custom detection algorithms on edge devices.
- Receive alerts from devices and monitor statistics from alerts to have a better insight into operations to create a secure, safety and effective environment.

# Chapter 2

## Literature Review

### 2.1 Overview of Existing IoT Frameworks in Surveillance

#### 2.1.1 Kyungroul Framework

Building an effective video surveillance system requires a coordinated combination of a centralized management system (CMS), strict access control mechanisms, and a flexible, scalable system architecture [1].

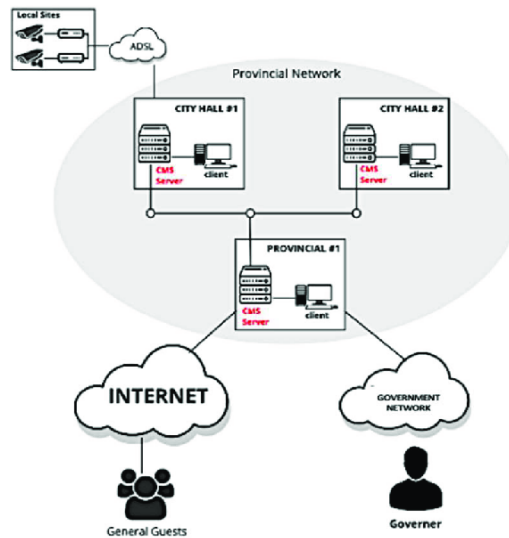


Figure 2.1: Scenario for a large-scale public surveillance video network

The solution proposed by Kyungroul Lee et al. [2] is to build a centralized management system (CMS) to integrate and distribute video streams from many different types of cam-

eras. Separate connection and decoding modules manage connections from many cameras, and process and distribute video streams to many different customers. Multi-level access management to protect video information.

Advantages:

- Centralized Management System (CMS): Effectively manage multiple types of cameras in the same system. Enhance remote management capabilities and comprehensive control of video from multiple sources.
- Separate connection and decoding modules: Reduces complexity in system development, making it easy to integrate cameras from different manufacturers. Increases flexibility, allowing the system to be easily expanded and upgraded.
- Flexible server architecture: Supports easy system expansion, allowing the system to handle multiple connections and distribute video efficiently. Bandwidth management helps maintain video quality even when many users are accessing it.
- Multi-level access management: Better privacy protection with different levels of access based on the sensitivity of the information.
- Encryption key management: Ensures the safety of video streams and sensitive information, that only authorized users can access. Encryption systems provide better protection against potential attacks.
- Privacy recovery mechanism: It helps protect sensitive information and allows flexible masking and recovery of privacy areas in the video. Useful in criminal investigations or incidents that require high security.

Disadvantages:

- Centralized Management System (CMS): May require large and complex system resources when scaling to larger systems
- Separate connection and decoding modules: Requires separate development and maintenance of connection and decoding modules, leading to increased complexity in deployment and maintenance.

- Flexible server architecture: Complex architecture can be expensive and require powerful hardware. Reliance on efficient management of bandwidth and resources can cause problems if not designed properly.
- Multi-level access management: Authorization can be complex, requiring a lot of effort to set up and maintain. It can be difficult to update and manage when the number of users is large.
- Encryption key management: Key management is complex, especially in large systems with many cameras and users. If the key is stolen or exposed, the system's security can be compromised.
- Privacy recovery mechanism: The encryption and recovery mechanism can add latency and increase processing requirements when decoding masked areas. Developing and deploying this mechanism requires more complexity than in a conventional surveillance system.

### 2.1.2 Mei Kuan Lim et al.'s Framework

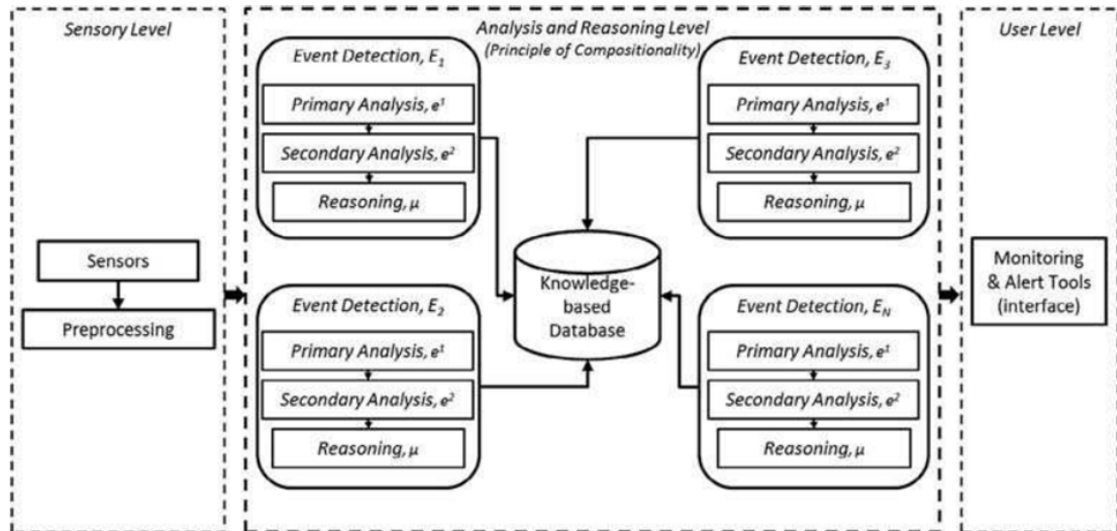


Figure 2.2: iSurveillance framework

This framework was named iSurveillance [4] and was designed in 2014 to detect multiple events in different ROIs within a video scene. While traditional systems are usually lim-

ited to single-task detection, iSurveillance utilizes compositional modeling to simplify the surveillance problem, allowing flexibility and a greater understanding of complex activities in real scenarios [1].

This framework was named iSurveillance [3] and was designed in 2014 to detect multiple events in different ROIs within a video scene. While traditional systems are usually limited to single-task detection, iSurveillance utilizes compositional modeling to simplify the surveillance problem, allowing flexibility and a greater understanding of complex activities in real scenarios [1].

Advantages:

- **Comprehensive Surveillance:** iSurveillance is designed to detect multiple events simultaneously within different ROIs in a single video scene. This capability is crucial in environments where diverse types of security threats may occur at the same time.
- **Enhanced Scene Understanding:** By modularizing the surveillance problem into various components (e.g., objects, activities), iSurveillance provides a more integrated and broader understanding of complex scenes, improving the accuracy and effectiveness of the surveillance system.
- **Compositional Modeling:** The framework breaks down complex surveillance tasks into smaller, manageable components, making it easier to adapt the system to different environments and scenarios without extensive reconfiguration.
- **Reduced Redundancy:** iSurveillance minimizes the need for redundant low-level processing across different events, streamlining the overall surveillance process and optimizing resource utilization.
- **Timely Alerts:** The system is capable of real-time analysis and can trigger alerts immediately upon detecting events.
- **High Accuracy and Performance:** Experimental results have shown a prominent level of accuracy and robustness in various scenarios. This makes it a reliable choice for complex surveillance environments.



Disadvantages:

- **Infrastructure Requirements:** Implementing requires significant infrastructure, particularly in large-scale environments. The need for extensive hardware, such as multiple cameras and edge devices, can be a substantial barrier to adoption.
- **Management Overhead:** The system requires maintaining the network of cameras, managing data processing at the edge, and ensuring detection algorithms are updated and deployed efficiently so all components function together seamlessly.
- **Fine-Tuning Requirements:** While iSurveillance is adaptable to different environments, it still requires fine-tuning to enhance performance for specific scenarios.
- **Resource Demands:** High processing demands at both the edge and central servers under high-load conditions could lead to performance bottlenecks.
- **Challenges with Large-Scale Deployment:** The sheer volume of data and events that need to be monitored can strain the system's capacity, especially in a large-scale environment.

### 2.1.3 YOLO Version 8 (YOLOv8)

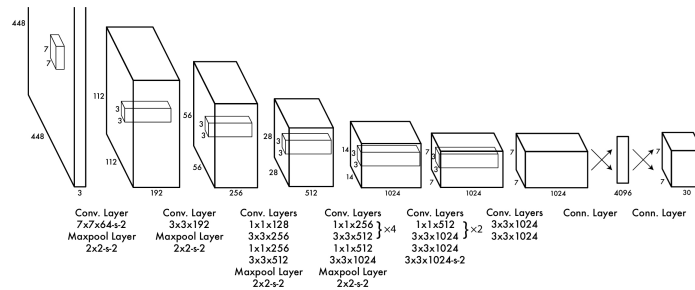


Figure 2.3: Original YOLO architecture

Developed by Joseph Redmon and Ali Farhadi at the University of Washington, YOLO (You Only Look Once) is a widely recognized model for object detection and image segmentation. Since its introduction in 2015, YOLO has become popular due to its speed and accuracy. Below figure is the original architecture of YOLO which is inspired by GooLeNet model for image classification [4].

Some significant milestones in the development of this model:

- **YOLO (2015):** The initial release, establishing the foundation for fast and accurate object detection.
- **YOLOv2 (2016):** Introduced batch normalization, anchor boxes, and dimension clusters for improved performance.
- **YOLOv3 (2018):** Further enhanced performance with a more efficient backbone network, multiple anchors, and spatial pyramid pooling.
- **YOLOv4 (2020):** Introduced innovations such as Mosaic data augmentation, a new anchor-free detection head, and a new loss function.
- **YOLOv5:** Improved performance and added features like hyperparameter optimization, integrated experiment tracking, and automatic export to popular formats.
- **YOLOv6 (2022):** Open-sourced by Meituan and deployed in their autonomous delivery robots.
- **YOLOv8 (2023):** Released by Ultralytics, focusing on enhanced performance, flexibility, and efficiency, supporting a comprehensive range of vision AI tasks.

Advantages:

- **Real-time performance:** This is a core design principle of YOLO. YOLO processes images at 45 frames per second, significantly faster than other detectors at 2016 [4].
- **High accuracy:** While early versions lagged in accuracy, later versions made significant improvements.[5] [6].
- **Generalization:** YOLO is known for its ability to generalize well to new objects and datasets. It learns to recognize objects based on their overall features rather than just specific examples, making it more robust in diverse environments [4].
- **Simplicity and ease of use:** Compared to some other complex object detection models, YOLO's architecture is relatively simpler, making it easier to understand, implement, and train.

Disadvantages:

- Difficulty with small objects: Due to its grid-based approach, YOLO can struggle with detecting small objects, especially those that appear in close proximity to each other [7].
- Localization errors: While YOLO is generally accurate, it can sometimes have issues with precise localization, meaning the bounding boxes around detected objects might not be perfectly tight [8].
- Challenges with unusual aspect ratios: YOLO's performance can be affected by objects with unusual or extreme aspect ratios.
- Training complexity: Training YOLO models, especially the more complex versions, can require substantial computational resources and large datasets.

# Chapter 3

## Objectives and Research Goals

### 3.1 Primary Objectives

This research seeks to design an stable, realtime, flexible and secured IoT platform for surveillance tasks, focusing on the following key objectives:

#### 3.1.1 Scalable And Secured System

This objective aims for creating a central system to manage and control IoT devices effectively, also able to process large data load from IoT devices when working monitoring tasks.

Additionally, the research also cover the scalability of the system when the workflow is overload, especailly with large monitoring system like airport or construction site surveillance. This means that the system monitors the resource consumption while running and give out the infrastructure to be scale up or down corresponding to the situation, optimizing the power and resource usage.

To ensured the security of the system, this research proposes a self-hosted solution that operates independently of 3<sup>rd</sup> party cloud systems, minimizing the possibility of being exposed to outsiders and the risk of remote attacks.

### 3.1.2 Stable And Real-time Monitoring System

In surveillance tasks such as monitoring, real-time recognition is essential to complete the work. This object tive is to keep the latency of the processing pipeline as low as possible. This latency could be breakdown into this formular:

$$L_{system} = L_{inference} + L_{transmission} + L_{processing} \quad (3.1)$$

Where:

- $L_{system}$  = Latency of the system
- $L_{inference}$  = Latency of the inference process in IoT device
- $L_{transmission}$  = Latency of the data transmission to other services
- $L_{processing}$  = Latency of the data processing of central server after receiving detection

By reducing above latencies, the system makes sure that the images from cameras is recorded, analyzed, transmitted and processed within reasonable time for surveillance purpose, allowing system's users have real-time overview of the situation and give instantaneous and precise decisions.

In the inference process in IoT devices, the latency could happen when the inference time in average for each frame in a video and the time needed to transmit the data to other services is taking longer than the time between 2 frames in the video. This latency per frame could be breakdown to following formular:

$$t_{latency} = t_{inference} + t_{transmission} - \frac{1}{FPS} \quad (3.2)$$

Where:

- $t_{latency}$  = Latency per frame in inference process (in seconds)
- $t_{inference}$  = Latency of the inference process (in seconds)
- $t_{transmission}$  = Latency of the data transmission to other services (in seconds)

- $FPS$  = Number of frames per second

If this latency value exceeds 0, then after each frame, the time between capturing frames from the cameras and getting the result of detecting objects will be accumulated, causing significantly delays in sending alerts in the system over time. Therefore, it is crucial to keep this value below 0 to make sure the system could perform in real-time.

The chosen of algorithms for objects detection also an important key point when we want to lower not only the inference time but also the power and resource consumption on IoT devices. Suitable models could enhance the performance of the workflow but not significantly sacrifice the accuracy of the detection.

The system also needs to be stable, which means the reliability and availability have to be ensured. The AWS's five 9s rank system [9] is one of the reliable metrics for benchmarking the downtime based on minutes of the system per year. Here is the example calculated downtime based on five 9s system:

Availability	Downtime / Year	Downtime / Month	Downtime / Day
<b>99.999%</b>	5.256 Minutes	0.438 Minutes	0.014 Minutes
<b>99.995%</b>	26.28 Minutes	2.19 Minutes	0.072 Minutes
<b>99.990%</b>	52.56 Minutes	4.38 Minutes	0.144 Minutes
<b>99.900%</b>	8.76 Hours	43.8 Minutes	1.44 Minutes
<b>99.500%</b>	43.8 Hours	3.65 Hours	7.2 Minutes
<b>99.000%</b>	87.6 Hours	7.3 Hours	14.4 Minutes

### 3.1.3 Multi-types Data Management

Adaptive system with various data types is necessary when working in a surveillance ecosystem. Not only we have to store the detection records but also the evidence like images, videos which only a normal database could not handle efficiently. This research suggests 3 types of data:

- Real-time message, effectively manage and routing data with high reliability.

- Structural data storage, give the ability to quickly access the all historical records of detections.
- Binary data storage, where to keep all media data and others which could not be stored in regular database.

## **3.2 Technical Requirements**

### **3.2.1 Stable And Real-time Performance Metrics**

- Inference and Transmission Latency: The latency in inference process and transmission to other services must remain below 0 to ensure the system operates in real-time.
- Success Message Received Rate: The platform should efficiently handle a high detection messages load, approximately 2,000 to 3,000 messages per 3 seconds, with a failure rate of 0.
- Data Correctness: The system must maintain data consistency with high accuracy to ensure reliable and accurate data handling.
- System Availability: High system availability is required, using five 9s standard to evaluate the maximum reliability.

### **3.2.2 Scalability And Security Requirements**

- Auto-Scaling Capability: The system must be able to scale resources horizontally, automatically scale resources respond to current resources consumption demand.
- Seamless Scaling Operations: The system must guarantee zero downtime during scaling process to ensure service always be available.
- Self-host Capability: The system must be deployable on-premise, and operate seamlessly without requiring an internet connection.

### 3.2.3 Multi-types Data Management

- Efficient Real-Time Data Processing: The system is required to handle data streams from queue service with minimum latency to ensure real-time performance.
- Data retention policy:
  - System data: Critical data which is required for the system to operate seamlessly, so the service could always be available.
  - Message queue data: Data from the queue service should be retained for at least 1 day to account for unexpected unavailability of the consumer service.
  - Binary data: Data for other purpose such as media like images, video, etc. need to be stored permanently. This is not only for serving report, monitoring tasks but also as a valuable resource for ongoing training for supervised models.

## 3.3 Success Criteria

### 1. Successful deployment of customized algorithms on IoT devices:

- Customized algorithms support: The system allows users to upload and config customized algorithms for specific requirements.
- Models deployment capability: The system provides ability to deploy customized models from central service to IoT devices without failure.
- IoT device processing control: The system could control the process running detection in IoT devices.

### 2. Security protected:

- Self-host capability: The system is able to support on-premise deployment, not depend on any 3<sup>rd</sup> party cloud service.
- Offline mode: The system must have ability to work without connection to the internet to isolate the system from external threats.



3. Handle 3,000 event detection messages with:

- Successful message delivery: The system must guarantee that at least 99.9% of all messages are delivered successfully to their routed consumers.
- End-to-end latency: The system must maintain an end-to-end latency to be no more than 3 second, from message generation to being delivered, to ensure real-time connection.

4. Ensure system reliability through:

- Minumum downtime: The system need to be available 99.9% of all time during operation to provide reliability to the users.
- Fault tolerance: The system should have mechanisms to get it recovered after being failed, ensuring uninterrupted operation.
- Data integrity: The system has to ensure data preservation even when the system fails, preventing data loss accross all usecases.

# Chapter 4

## Technology Evaluation and Comparison

### 4.1 Edge Computing for Surveillance Task

#### 4.1.1 Edge Computing Architecture

Edge Computing is an architecture which is designed for scenarios where it is more advantageous when processing data closer to where it is generated, rather than sending data to a centralized service [10]. Moreover, edge computing allows for handling detection tasks locally without unnecessarily sending large video streams to other services for processing. Besides, due to the advantages of edge computing architecture, the workload on centralized services is reduced significantly, helping avoid increasing costs of scaling operations, and also reducing bandwidth and storage usage costs. With this architecture, the centralized service only needs to receive events from IoT devices and aggregate them to show on the user interface. Also, because of the reduction in transmission processes (not going through a cloud service for processing), the received events are more likely to be real-time, which makes it suitable for surveillance monitoring tasks.

Nonetheless, this Edge Computing architecture also comes with trade-offs. While real-time processing is a superior advantage, there are some limitations in hardware such as the processing power of edge devices or environmental impacts. Because most IoT devices have minimal size, which may limit their processing capabilities compared to normal servers [10]. This factor needs to be considered when implementing complicated algorithms for

streaming analytics. The environment where the devices are deployed is also a dilemma since these edge devices are quite sensitive to temperature, power supply, or connectivity.

### 4.1.2 Cloud Computing Architectures

Cloud Computing services are quite varied; for example, in Infrastructure as a Service (IaaS), there are EC2 from Amazon Web Services (AWS), Google Cloud Platform, or Microsoft Azure Virtual Machines that offer great options for virtual machines needed for computational operations. With these infrastructures, data is processed in remote data centers provided by cloud providers. The system is highly scalable with resources easily provided according to demand [11]. Also, unlike the limitations in computational operations on edge devices, Cloud Computing services provide powerful inference capabilities which help to detect anomalous events faster and are less sensitive to environmental conditions. Cloud providers like Google, Amazon, Microsoft, etc. offer redundancy and fault tolerance in their services, but they also raise a concern that the system now relies on a single connection with the providers. Moreover, the cost of the service is usually calculated on a pay-as-you-go model based on storage, processing power, and bandwidth usage. With this fee model, the cost normally would be higher than a self-hosted system. And security is also a problem since the data is stored and processed in the providers' infrastructure. Nonetheless, instead of processing locally, the data now has another transmission step from the local to the cloud services and back, which definitely results in higher latency.

### 4.1.3 Factors to Consider

Between Edge Computing and Cloud Computing architectures for IoT surveillance monitoring tasks, these factors should be evaluated:

- **Latency:** With this factor, Edge Computing architecture is much preferred due to the ability to process locally and reduce stopping points in the data transmission process.
- **Workload Tolerance:** With higher computational power, Cloud Computing services are a good solution to process high-throughput data loads simultaneously. Although

Edge Computing models can overcome this difficulty by scaling out, meaning increasing the number of edge devices to process more data at a time.

- **Infrastructure Costing:** In the short term, Cloud providers offer a great deal on the cost to hire machines for processing, but in the long term, with the pay-as-you-go model, the costs would accumulate and eventually become higher than the cost for Edge Computing infrastructure, which would be quite expensive in setup steps.
- **Data Privacy:** For this evaluation point, Edge Computing architecture provides more privacy for the data than Cloud services because all data is stored and processed locally compared to remote servers which belong to the providers.

#### 4.1.4 Edge Computing Technologies

Edge Computing involves a range of technologies which help with data processing and storage on-site. Here are some key technologies used for Edge Computing:

- **Hardware Infrastructure:** There are plenty of device models, ranging from small, single-board computers to larger, rack-mounted servers. Some well-known models are Raspberry Pi, Asus Tinker Board, NVIDIA Jetson Nano, Arduino, etc.
- **Software and Platforms:** With constrained hardware, lightweight operating systems are much preferable, such as Raspberry Pi OS, Ubuntu, Armbian, etc. There are also wide options for programming languages and frameworks like Python, C/C++, NodeJS, Java.
- **Data Management:** For most use cases, lightweight databases which are designed for edge devices such as SQLite [12] or InfluxDB [13] enable local data storage and retrieval.

Each of these technologies has its own strengths and weaknesses; based on the circumstances, environment, constraints, etc., we can decide which are the best options for the project.

### **4.1.5 Edge-Cloud Approach**

As the two architectures demonstrated above, a hybrid model is also a solution in multiple scenarios [14]. While Edge Computing can resolve latency-sensitive requirements through local data processing, filtering, and basic analytics, Cloud Computing handles intensive computational operations like complex analytics tasks, large-scale data storage, and machine learning model training. This approach brings users real-time responsiveness and comprehensive data insights in applications.

### **4.1.6 Solution for IoT Surveillance Monitoring System**

The decision between Edge Computing and Cloud Computing architectures depends on many aspects of an IoT surveillance monitoring system. For real-time awareness and on-premise systems, Edge Computing architecture is more favorable with its capability to work seamlessly without internet connection. On the other hand, Cloud Computing is best suited for robust complex computational tasks and high scalability. For IoT surveillance tasks, real-time responsiveness is more critical, so Edge Computing architecture is more suitable in this use case.

## **4.2 Message Queue: RabbitMQ vs. Apache Kafka**

### **4.2.1 RabbitMQ**

RabbitMQ is a well-known open-source message broker that supports communication between applications. It uses the Advanced Message Queuing Protocol (AMQP) for transmission and also supports many other messaging protocols, accommodating various types of applications [15].

RabbitMQ provides flexible message routing capabilities, enabling message routing based on diverse parameters such as headers, exchange types (direct, topic, fanout, headers), and binding keys [16]. This helps the system deliver messages to specific consumers with complex criteria. With its user-friendly management interface and command-line support

tools, the setup and configuration steps are simplified, allowing for rapid development and deployment. RabbitMQ also has various plugins [17] and extensions for advanced functionality, including message tracing, delayed message delivery, and management of message queues. The service is available on many operating systems and can be deployed in clusters for high availability and fault tolerance. One of the valuable advantages of RabbitMQ is its high reliability, with features like message persistence, acknowledgments, and publisher confirms.

It is suitable for building distributed systems, microservices architectures, and applications that require reliable message delivery with complex constraints.

### **4.2.2 Apache Kafka**

Apache Kafka is a distributed streaming platform designed for high-throughput, fault-tolerant, and real-time data streaming applications [18]. It follows a pub-sub model, allowing producers to publish messages to topics and consumers to subscribe to those topics for receiving messages.

Kafka has the ability to handle millions of messages per second, making it suitable for large-scale data integration and processing. Its mechanism for fault tolerance works through data replication across multiple partitions, ensuring data availability if any partition fails. Kafka keeps messages on disk, which ensures durability and allows historical data analysis. Many applications support stream processing through Kafka, such as Apache Flink [19] and Apache Spark [20], enabling real-time data transformation and analysis. Kafka's scalability is also remarkable for its ability to increase throughput by adding more brokers to the cluster.

It is ideal for real-time data pipelines, streaming analytics, etc. with heavy workload.

### **4.2.3 Factors to Consider**

These factors should be considered when give a decision for the technology of message queue:

- **Scalability and Performance:** Evaluate the message throughput and latency for the best fit for the application.
- **Architecture and Data Model:** The architecture of the message queue system plays an important role in deciding which technologies will be the best for specific use cases.
- **Reliability and Message Persistence:** The confidence in reaching the destination is highly considered because it ensures the application can work seamlessly with minimal chance of misinformation.
- **Ecosystem and Integration:** The compatibility of the technologies with other infrastructure is an important factor since it makes it easier to integrate the system with other applications.
- **Resource Requirements:** Resource optimization is taken into account because of the cost for the facilities of the service.

When making a final decision, we should evaluate the project's specific demands based on these criteria. Kafka is often the better choice for high-throughput event streaming and data pipeline scenarios, while RabbitMQ is typically more suitable for traditional enterprise messaging patterns and when message routing flexibility is required.

#### 4.2.4 RabbitMQ for IoT Surveillance Monitoring System

RabbitMQ has distinct advantages for IoT surveillance monitoring systems through its robust messaging capability and flexible architecture [21]. Moreover, with its built-in clustering and federation features, RabbitMQ ensures reliable message delivery, reducing the chance of data loss in transaction processes. The platform also supports multiple messaging protocols, such as MQTT, making it suitable for IoT devices that typically use lightweight protocols for data transmission.

For IoT surveillance projects, RabbitMQ provides reliable message delivery, flexible and complex routing capabilities, and security features that are suitable for building a scal-

able monitoring system. Especially in complex message routing scenarios, RabbitMQ is a strong candidate, combined with its highly reliable message delivery.

#### **4.2.5 Other Message Queue Technologies**

Beyond popular choices like Apache Kafka and RabbitMQ, several other message queue technologies offer compelling features for today's distributed systems. For example, Amazon Simple Queue Service (SQS) is a fully managed service that works smoothly with other AWS offerings [22]. It provides both standard and FIFO (First-In, First-Out) queues, ensuring reliable message delivery and processing. Microsoft Azure Event Hubs is a powerful platform built for large-scale data streaming and event intake. It can handle millions of events per second and includes tools for time-based data analysis [23]. Redis Pub/Sub provides a simpler, lightweight option for applications that demand extremely low latency, although its message delivery guarantees are different from traditional message queues [24]. Projects with sophisticated messaging requirements or legacy system integration needs often turn to Apache ActiveMQ, which supports an extensive array of protocols and enterprise patterns. This makes it a good fit for organizations with complex messaging needs or those needing to connect with older systems. Each of these alternatives has unique advantages for different situations and system designs. This allows organizations to choose the technology that best meets their specific needs in terms of scalability, reliability, and integration.

### **4.3 RDBMS Databases: MySQL**

#### **4.3.1 MySQL**

MySQL is one of the most popular open-source relational database. It uses SQL, which is a standard language, to perform operations such as creating, retrieving, updating, and deleting data from the database. MySQL is usually used in web applications, data analysis, etc. due to its reliability, scalability, and good performance. Because of its standard protocols in connection and query language, MySQL can easily integrate with various programming



languages like Python, Java, NodeJS, etc. MySQL also offers useful features, including ACID compliance (which ensures data consistency), data replication, etc. This makes it a common choice for developers and organizations [25].

### 4.3.2 Factors to Consider

For giving decision to use which database for IoT surveillance monitoring task, we could use these following factors:

- **Scalability and Availability:** As the number of IoT devices grows over time, the database must have the capability to scale. Also, high availability is important for the system to work seamlessly.
- **Data Retention and Archiving:** The ability to decide how long data will be kept, and for data that is no longer in use, the database should have an efficient mechanism to archive and retrieve data.
- **Security:** Surveillance data is sensitive. Strong access controls, encryption, and audit trails are important database features.
- **Cost:** This considers licensing fees, hardware costs, and maintenance expenses for the database.

### 4.3.3 MySQL for IoT Surveillance Monitoring System

MySQL is suitable for IoT Surveillance Monitoring systems due to many reasons, such as being mature and widely used, which provides a large support community with extensive documentation, and its cost effectiveness, not only because it is an open-source product but also because it requires less cost in operations for smaller or medium-sized deployments. This also leads to MySQL being able to integrate with various tools and technologies, making it easier to adapt to a wide range of application tech stacks. MySQL also offers scalability features, for example, asynchronous replication, group replication, sharding, etc., which enhance availability, disaster recovery, and extensibility. With security features

like user authentication, access control, encryption at both rest and transit, and auditing, MySQL provides assurance for privacy of surveillance data.

In summary, MySQL provides a solid foundation for IoT surveillance monitoring, especially for deployments that prioritize cost-effectiveness, data integrity, flexibility, and strong community support.

#### 4.3.4 RDBMS Database Alternatives

Beyond MySQL, there are some other options for RDBMS to implement into IoT Surveillance project:

**PostgreSQL:** It is known for its robustness, reliability, and strong adherence to SQL standards. It also supports other data type like JSON, which might be beneficial for handling semi-structured data from IoT devices. PostgreSQL is also highly scalable and performs well with large datasets [26].

**MariaDB:** As a fork of MySQL, it aims to remain open-source and provides improved performance and some additional features compared to MySQL. Since it inherit from MySQL, the migration from MySQL to MariaDB is quite easy [27].

**Microsoft SQL Server:** Microsoft's commercial RDBMS product with various set of features and tools. It offers excellent performance, scalability and security. SQL Server is often used in enterprise environments and can be a good option if the existed technologies are using Microsoft's ecosystem [28].

**Oracle Database:** Another commercial product from Oracle, which is known as a wonderful software ecosystem for enterprises. This database is well-known for its robustness and scalability. Organizations usually use it in large-scale applications with high transaction throughput. Oracle database also have advanced features like partitioning, etc. to increase availability [29].

**ClickHouse:** This is an open-source column-oriented database management system designed for online analytical processing (OLAP). It well handle massive data and able to execute complex analytical queries with high performance. In IoT scenarios, it is well-suited because there are lots of usecases which need to analyze large amounts of data from

multiple IoT devices [30].

It's crucial to carefully evaluate the project's requirements and compare the features and capabilities of different RDBMS options before making a decision. The scale of the project, data types, performance requirements, resource constraints or budgets could be the factors for considering the most suitable database for the application.

## 4.4 Object Storage: MinIO vs. AWS S3

### 4.4.1 MinIO

MinIO is a high-performance, Kubernetes-native object storage server compatible with the Amazon S3 API. This compatibility is a key advantage, allowing applications designed for S3 to work with MinIO with minimal or no code changes [31]. MinIO focuses on simplicity, performance, and scalability, making it suitable for various use cases, including:

- **On-Premise and Hybrid Cloud Deployments:** Unlike S3, which is cloud-exclusive, MinIO can be deployed on any infrastructure, including on-premises servers, private clouds, and public clouds. This flexibility makes it ideal for hybrid cloud strategies and edge computing scenarios.
- **High Performance:** MinIO is designed for high throughput and low latency, making it suitable for performance-sensitive applications like AI/ML, big data analytics, and high-performance computing.
- **Kubernetes Native:** MinIO is designed to run seamlessly on Kubernetes, making it easy to deploy and manage in containerized environments.
- **Cost-Effectiveness:** For on-premise deployments, MinIO have extremely low costs associated with cloud storage services. This can be a significant advantage for organizations with large data storage needs.
- **Simplified Management:** MinIO focus on simplicity translates to easier deployment and management compared to complex distributed storage systems.

- **Open Source and Community Support:** MinIO benefits from a large and active community, contributing to its continuous development and improvement.

#### 4.4.2 AWS S3

AWS S3 is a fully managed, scalable, and highly available object storage service offered by Amazon. It provides virtually unlimited storage capacity, high durability, and low latency access to data [32]. S3 is well-known for its great feature set, including:

- **Scalability and Durability:** S3 is designed for 99.999999999% (11 nines) durability and can handle massive amounts of data.
- **Security:** S3 offers various security features, such as access control lists (ACLs), bucket policies, and encryption.
- **Integration with AWS Services:** S3 integrates seamlessly with other AWS ecosystem, such as EC2, Lambda, and EMR.
- **Global Availability:** S3 is available in multiple AWS regions worldwide.

#### 4.4.3 Factors to Consider

When choosing which Object Storage technology to use, consider the following factors:

- **Deployment Model:** Which application model does the project is deployed on? Fully cloud or on-premise or hybrid, etc.
- **Performance Requirements:** The ability to handle high wordload operations related to data storage.
- **Cost Considerations:** Evaluate the total cost of ownership (TCO), considering storage capacity, data transfer costs, and management overhead.
- **Integration with Existing Infrastructure:** Consider how well each option integrates with your existing infrastructure and applications.

#### **4.4.4 MinIO for IoT Surveillance Monitoring System**

MinIO is a strong choice for IoT surveillance, mirroring MySQL's role for structured data but focusing on media data like video, images. Its deployment flexibility, from on-premise to the cloud, is crucial for distributed surveillance setups, enabling low-latency data capture and reduced bandwidth. Like MySQL's cost-effectiveness, MinIO offers significant cost benefits, especially for on-premise and hybrid deployments, by reducing reliance on expensive cloud storage.

MinIO's S3 compatibility supports integration with existing surveillance tools and platforms. Its high performance and scalability are essential for real-time video processing and handling increasing data volumes. Robust security features, including encryption and access control, ensure data integrity [33].

# Chapter 5

## Methodology

### 5.1 System Architecture Design

#### 5.1.1 IoT Hardware

For monitoring simulation, there are various options for IP cameras. In this research scope, we are using a body camera from Dahua model MPT220 as an IP camera because of its ability to stream video from the device using Real-Time Streaming Protocol (RTSP). The camera supports 1080p@30 fps recording with H.264 and H.265 high compression coding, which makes it suitable for testing surveillance tasks. Additionally, the project also involves a camera from Hikvision for testing on multiple models of cameras.

A Raspberry Pi 4 Model B is selected to be the IoT device for edge computing operations because of its P/P (Price over Performance). With 4GB of RAM and a 64-bit quad-core Arm Cortex-A72 processor, the device has enough resources and power to effectively process streaming data from IP cameras. This processing capability allows for real-time video analytics and reduces bandwidth requirements by performing analysis locally before transmitting only relevant data.

## 5.2 Message Queue

### 5.2.1 RabbitMQ Implementation

RabbitMQ offers a flexible and organized approach to message queuing, enabling efficient communication between producers and consumers. Exchanges in RabbitMQ are designed to facilitate the routing of messages, with a structure that allows for precise control over message delivery.

To simplify the use case scenario, the project uses the default exchange and only uses queue names as the method to separate the message destinations. There are 3 types of messages produced by the IoT devices:

- **detection:** Sends events detected by the algorithms deployed on IoT devices.
- **status:** Sends signals to check if the IP cameras are still active or not.
- **snapshots:** Sends snapshots of the IP cameras at an interval of time.

With these queues configuration, majority of the necessary usecases which is needed for an IoT system works seamlessly are covered.

## 5.3 MinIO - Object Storage

Implementing MinIO involves a structured methodology to ensure successful deployment and management, especially in contexts like IoT surveillance.

- **Storage Needs:** Depends to deployed device's capacity.
- **Performance Requirements:** Can handle approximately 20-50 read/upload operations at a time.
- **Security:** Have access key and secret key to make sure only application services can access to the storage.
- **Deployment Model:** On-premise model.

This streamlined methodology ensures a robust, performant, and secure MinIO deployment, crucial for applications like IoT surveillance where data integrity and availability are paramount.

## 5.4 Solution System Architecture

Building on the methodology outlined in the previous points, this research proposes a solution using the mentioned technologies within an Edge Computing Architecture:

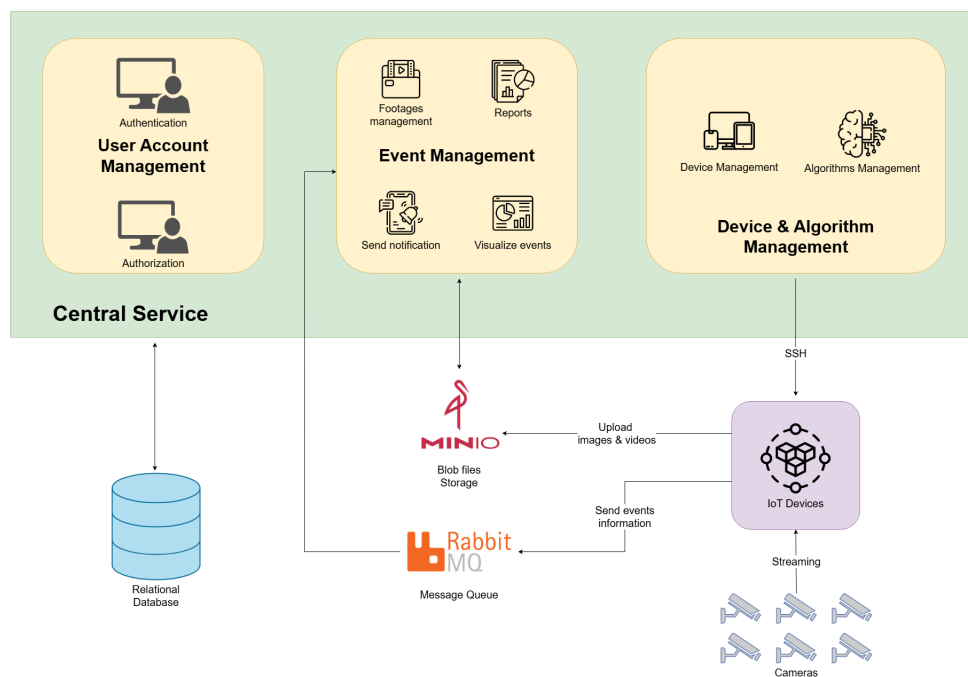


Figure 5.1: Solution architecture for surveillance monitoring task

Architecture Components:

- **Centralized Management System (CMS)**
  - User Account Management: Handles authentication and authorization for users.
  - Event Management: Manages detected events from IoT devices, retrieves detection data for reporting, and sends notifications for new events.
  - Device and Algorithm Management: Oversees device management and deployment of customized algorithms.



- **IoT Devices:** Responsible for receiving streaming signals from IP cameras, performing edge processing using customized models deployed by the CMS, sending event signals back to the CMS, and uploading detected footage to MinIO.
- **RabbitMQ:** Serves as a message queue system for data transmission between IoT devices and the CMS.
- **MinIO:** A storage system for unstructured data, such as media files like images, videos, etc.
- **Relational Database (MySQL):** Stores metadata related to detected events, devices, user accounts, and other structured information.

## 5.5 Workflow Testing Methodology

The workflow testing includes several test cases to test if the system works as expected under different scenarios:

- **Authentication and Authorization:** The system is able to use username and password to check if user is authorized in the application.
- **Add trained AI models:** The system is able to add customized algorithm files to the system.
- **Add customized algorithms:** The system is able to add customized algorithm files to the system.
- **Add IP cameras:** The system is able to add IP cameras' information which will be deployed on IoT devices.
- **Add IoT devices:** The system is able to add IoT devices' information to make connections for deployment.
- **Deploy/Stop Model:** The system is able to deploy customized models on IoT devices and run/stop detection processes on the devices.

- **Receive data from IoT devices:** The system is able to receive data including detected events, status signals, and snapshots of defined IP cameras from IoT devices.

These scenarios ensure the system is working seamlessly with IoT devices and IP cameras.

## 5.6 Solution System Workflow

The following diagram demonstrates the proposed workflow for deploying services from the central system to IoT devices and receiving detection signals from these devices:

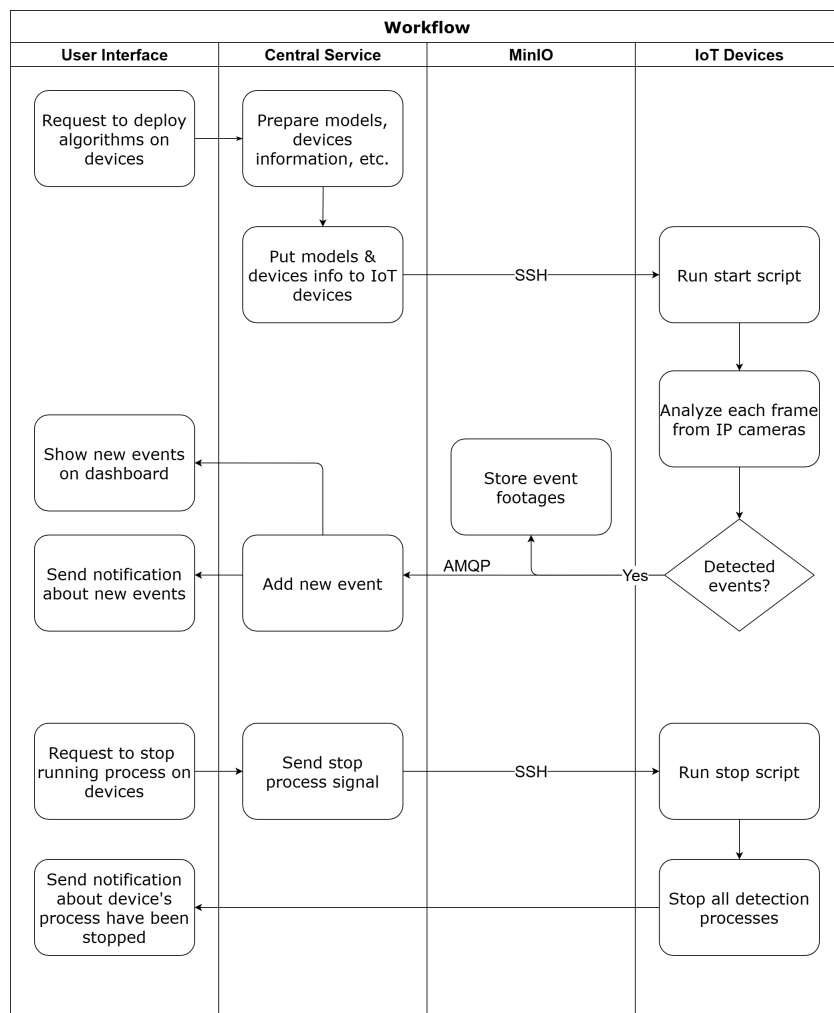


Figure 5.2: System workflow for IoT deployment

2 Main Deployment Workflows:

- **Deploy and Start Process:** Deploy customized algorithm models to IoT devices via SFTP and initiate the detection process to monitor events from IP cameras.
- **Stop Process:** Use an SSH command to execute a script which stop all active detection processes.

## 5.7 Customzied Algorithms

All customized models in this research are trained on the YOLO v8 model, which is pre-trained on COCO datasets with 80 object classifications. Using this base model, training continues with the following datasets:

- **Vehicle detection:** Detects multiple vehicle types such as cars, buses, motorbikes, etc.
- **People detection:** Detects if a person is in the picture
- **Safety assets detection:** Detects if a person is wearing safety assets like helmets, jackets, etc.
- **Trash bin detection:** Detects trash bins, overflowing trash, and trash outside the bin.

With these customized models, the system can prove its flexibility working with various surveillance scenarios in different environments.

# Chapter 6

## Implementation

### 6.1 IoT Device Setup

#### 6.1.1 Hardware Configuration

Listing 6.1: IoT Device Initial Setup

```
1 # Install NodeJS on Raspberry Pi
2 sudo apt-get install -y nodejs
3 node -v
4 # Install NPM on Raspberry Pi
5 sudo apt install npm
6 npm -v
7 # Install required libraries
8 npm install pm2 -g && pm2 update
9 sudo apt-get install -y python3-pip
```

#### 6.1.2 Detection Software

Listing 6.2: Detect Events from IP cameras Script

```
1 def inference_camera(url, camera_id, algorithms, detect_mode):
2     # Get camera and set its frame size
3     cap = cv2.VideoCapture(url)
4     frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
5     frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
6
7     cap.set(cv2.CAP_PROP_FRAME_WIDTH, frame_width)
8     cap.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_height)
9
```

```

10     print(f"CAMERA_URL: {url}\nFrame width: {frame_width}\nFrame height: {frame_height}\nFrame rate: {cap.get(cv2.CAP_PROP_FPS)}")
11
12     if detect_mode == 'tracking':
13         Tracker().inference(url, camera_id, algorithms, cap)
14     if detect_mode == 'alert':
15         Alert().inference(url, camera_id, algorithms, cap)

```

Listing 6.3: Send messages Script

```

1 def publish_detection(queue_name=RMQ_QUEUE, camera="",
2     algorithm="", image_url="", video_url=""):
3     try:
4         # Establish a connection to RabbitMQ
5         connection = pika.BlockingConnection(pika.
6             ConnectionParameters(RMQ_HOST, port=RMQ_PORT))
7         channel = connection.channel()
8
9         # Declare the queue
10        channel.queue_declare(queue=queue_name, durable=True)
11
12        # Create the detection data
13        detection_data = {
14            "timestamp": datetime.datetime.now().isoformat(),
15            "camera": camera,
16            "algorithm": algorithm,
17            "image_url": image_url,
18            "video_url": video_url,
19        }
20
21        # Publish the message
22        channel.basic_publish(exchange='', routing_key=
23            queue_name, body=json.dumps(detection_data))
24
25        print(f" [x] Sent {detection_data}")
26
27        # Close the connection
28        connection.close()
29
30        return detection_data
31    except:
32        print('Something wrong happened')

```

Listing 6.4: Upload media files Script

```

1 def get_image_url(bucket_name=MINIO_BUCKET, file_name=""):
2     if not file_name:
3         return ""
4     image_url = f"{MINIO_HOST}/{bucket_name}/{file_name}"
5     return image_url
6

```

```

7 def upload_file(bucket_name=MINIO_BUCKET, file_path="",
8   file_name=""):
9     try:
10         # The file to upload, change this path if needed
11         source_file = file_path
12
13         # The destination bucket and filename on the MinIO
14         # server
15         destination_file = file_name
16
17         # Make the bucket if it doesn't exist.
18         found = client.bucket_exists(bucket_name)
19         if not found:
20             client.make_bucket(bucket_name)
21             print("Created bucket", bucket_name)
22
23         # Upload the file, renaming it in the process
24         client.fput_object(
25             bucket_name,
26             destination_file,
27             source_file,
28         )
29
30         print(source_file, "successfully uploaded as object",
31               destination_file, "to bucket", bucket_name)
32
33         os.remove(file_path)
34
35     except Exception as e:
36         print("Something wrong happened when upload image")
37         print(f"An error occurred during upload: {e}")
38         return ""

```

Listing 6.5: Check IP cameras status Script

```

1 def check_camera_status(camera_id, camera_url):
2     try:
3         cap = cv2.VideoCapture(camera_url)
4         start_time = time.time()
5         while time.time() - start_time < 5:
6             if cap.isOpened():
7                 cap.release()
8
9                 # Publish message to notify status
10                publish_status(camera_id=camera_id, status="
11                               ACTIVE")
12
13                return True
14                time.sleep(0.1)
15            cap.release()

```

```

16         # Publish message to notify status
17         publish_status(camera_id=camera_id)
18         return False
19     except Exception as e:
20         return False

```

## 6.2 Message Queue and Object Storage Configuration

### 6.2.1 RabbitMQ Setup

Listing 6.6: RabbitMQ and MinIO Docker Compose Setup

```

1  version: '3.8'
2
3  name: message-queue-s3
4  services:
5      rabbitmq:
6          image: rabbitmq:4.0-management
7          container_name: IoTMQ
8          hostname: rabbitmq
9          ports:
10             - "5672:5672"
11             - "15672:15672"
12          networks:
13             - my_network
14
15      minio:
16          image: quay.io/minio/minio
17          container_name: minio-storage-s3
18          ports:
19             - "9000:9000"
20             - "9001:9001"
21          environment:
22             MINIO_ROOT_USER: minio_username
23             MINIO_ROOT_PASSWORD: minio_password
24          volumes:
25             - path/to/store/data/on/host/machine
26          command: server /data --console-address ":9001"
27          networks:
28             - my_network
29
30  networks:
31      my_network:
32          driver: bridge

```

## 6.3 Central Service Implementation

### 6.3.1 MySQL database

Listing 6.7: MySQL Docker Compose Setup

```
1  services:
2  mysql:
3      image: mysql:8.0
4      container_name: fsb_sql_db
5      environment:
6          MYSQL_ROOT_PASSWORD: admin_pwd
7          MYSQL_DATABASE: fsb
8          MYSQL_USER: fsb
9          MYSQL_PASSWORD: fsb_pwd
10     ports:
11         - "3306:3306"
12     volumes:
13         - mysql_data:/var/lib/mysql
14     healthcheck:
15         test: [ "CMD", "mysqladmin", "ping", "-h", "localhost" ]
16         timeout: 20s
17         retries: 10
18
19     flyway:
20         container_name: fsb_flyway_db_migration
21         image: flyway/flyway
22         command: -url=jdbc:mysql://mysql/fsb -user=fsb -password=
23             fsb_pwd -connectRetries=60 migrate
24         volumes:
25             - ./sql:/flyway/sql
26         depends_on:
27             mysql:
28                 condition: service_healthy
29
30 volumes:
31     mysql_data:
```

### 6.3.2 Tables Schema

Listing 6.8: MySQL Tables Initialize Scripts

```
1  CREATE TABLE
2  cameras (
3      id CHAR(36) PRIMARY KEY,
4      name VARCHAR(255) NOT NULL,
5      uri TEXT NOT NULL,
```



```

6   lat DOUBLE NOT NULL,
7   'long' DOUBLE NOT NULL,
8   algorithms TEXT,
9   status VARCHAR(50) NOT NULL,
10  zones TEXT,
11  snapshot TEXT,
12  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
13  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
14 );
15
16 CREATE TABLE
17 algorithms (
18   id CHAR(255) PRIMARY KEY,
19   name VARCHAR(255) NOT NULL,
20   description TEXT NOT NULL,
21   configs TEXT NOT NULL,
22   threshold INT NOT NULL,
23   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
24   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
25 );
26
27 CREATE TABLE
28 iot_devices (
29   id VARCHAR(255) PRIMARY KEY,
30   name VARCHAR(255) NOT NULL,
31   cameras VARCHAR(255) NOT NULL,
32   ip VARCHAR(20) NOT NULL,
33   port INT NOT NULL,
34   username VARCHAR(255) NOT NULL,
35   password VARCHAR(255) NOT NULL,
36   deployment_path TEXT NOT NULL,
37   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
38   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
39 );
40
41 CREATE TABLE
42 detections (
43   id VARCHAR(255) PRIMARY KEY,
44   timestamp DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
45   camera VARCHAR(255) NOT NULL,
46   algorithm VARCHAR(255) NOT NULL,
47   image_url VARCHAR(255) NOT NULL,
48   video_url VARCHAR(255),
49   approved TINYINT DEFAULT 0 NOT NULL,
50   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
51   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
52 );
53
54 CREATE TABLE
55 ai_models (
56   id VARCHAR(255) PRIMARY KEY,

```

```

57     name VARCHAR(255) NOT NULL,
58     description TEXT NOT NULL,
59     model_type ENUM ('default', 'onnx', 'ncnn') DEFAULT 'default'
        NOT NULL,
60     model_name VARCHAR(255) NOT NULL,
61     model_size FLOAT NOT NULL,
62     class_id TEXT NOT NULL,
63     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
64     updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
65 );

```

### 6.3.3 Handle Detections from IoT Devices

Listing 6.9: Process New Detection Logic

```

1  @Public()
2  @RabbitSubscribe({
3      queue: 'detections',
4      errorHandler: (channel, msg, error) => {
5          console.error('Error processing message:', error);
6          channel.nack(msg);
7      },
8  })
9  async handleDetectionMessage(msg: any) {
10     try {
11         this.create({ ...msg, timestamp: new Date(msg.timestamp)
12             });
13         return;
14     } catch (error) {
15         throw error;
16     }
17 }

```

### 6.3.4 Handle Alert Signal from IoT Devices

Listing 6.10: Process Alert Signal Logic

```

1  @Public()
2  @RabbitSubscribe({
3      queue: 'alert',
4      errorHandler: (channel, msg, error) => {
5          console.error('Error processing message:', error);
6          channel.nack(msg);
7      },
8  })
9  async handleAlertMessage(msg: any) {

```

```
10  try {
11      const cameraId = msg.camera;
12      const notificationConfig: Notification = await
13          Notification.findOne({
14              where: { camera_id: cameraId },
15          });
16      const camera = await Camera.findByPk(cameraId);
17      if (!notificationConfig || !camera) return;
18
19      const message = `New alert for ${msg.algorithm} at camera
20          ${camera.name}`;
21      this.notificationService.sendNotification(message,
22          notificationConfig);
23  } catch (error) {
24      throw error;
25  }
```

# Chapter 7

## Results and Analysis

### 7.1 Algorithm Performance Metrics

#### 7.1.1 Algorithm Models Performance

Model name	Datasets	Input size	Epochs	Precision	mAP50(B)	Model size
yolov8s-person.pt	17,401	800	20	82%	0.77	21.4 MB
yolov8s-vehicle.pt	17,210	800	20	69%	0.78	21.4 MB
yolov8n-vehicle.pt	17,210	800	20	67%	0.76	5.96 MB
yolov8n-safety.pt	20,280	800	40	60%	0.61	21.4 MB
yolov8s-safety.pt	20,280	800	20	60%	0.64	5.96 MB
yolov8n-trash.pt	6,885	800	40	89%	0.91	21.4 MB
yolov8s-trash.pt	6,885	800	40	86%	0.90	5.96 MB

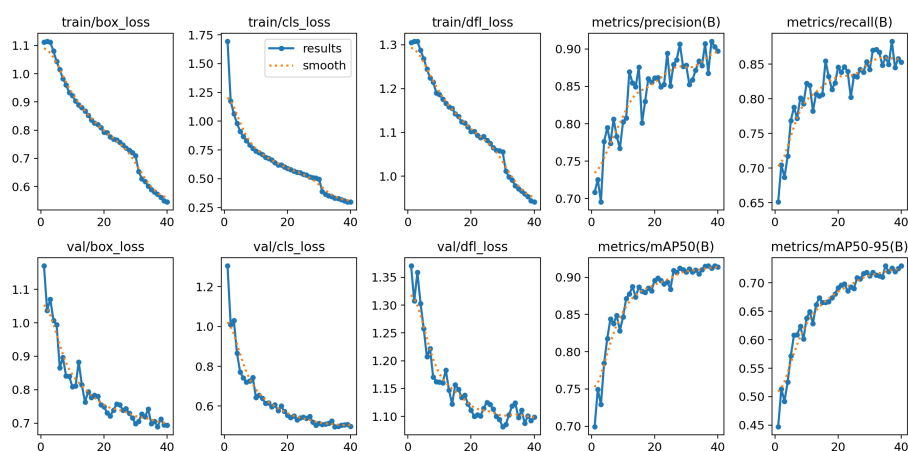


Figure 7.1: Training result of model YOLO v8 nano with Trash Bin datasets

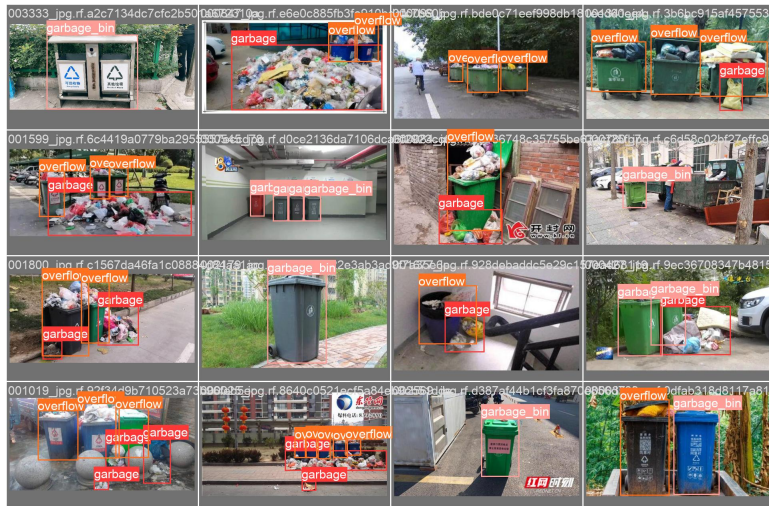


Figure 7.2: Validation Label of model YOLO v8 nano with Trash Bin datasets

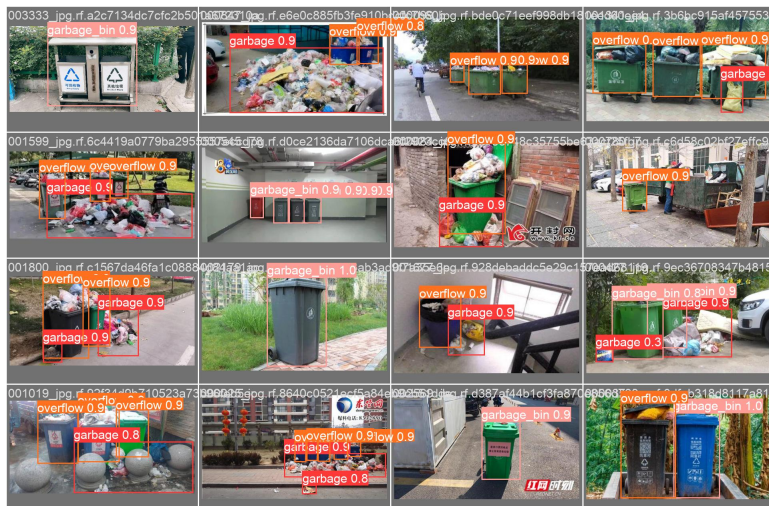


Figure 7.3: Validation Predict of model YOLO v8 nano with Trash Bin datasets

## 7.2 Workflow Testing

### 7.2.1 Authentication and Authorization

### Listing 7.1: Request To Login

```
1 curl --location 'localhost:3000/auth/login' \  
2 --header 'Content-Type: application/x-www-form-urlencoded' \  
3 --data-urlencode 'username=username' \  
4 --data-urlencode 'password=password'
```

### Listing 7.2: Login Success Response Body

```
1 {
2     "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
3 }
```

Listing 7.3: Login Fail Response Body

```
1 {  
2   "message": "Unauthorized",  
3   "statusCode": 401,  
4   "timestamp": "date_time_ISO_format"  
5 }
```

Listing 7.4: Request To Get User Profile

```
1 curl --location 'localhost:3000/auth/profile' \  
2 --header 'Authorization: Bearer  
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'
```

Listing 7.5: Get User Profile Success Response Body

```
1 {  
2   "sub": 7,  
3   "username": "admin",  
4   "role": "admin",  
5   "iat": 1735431340,  
6   "exp": 1735517740  
7 }
```

## 7.2.2 Add trained AI models

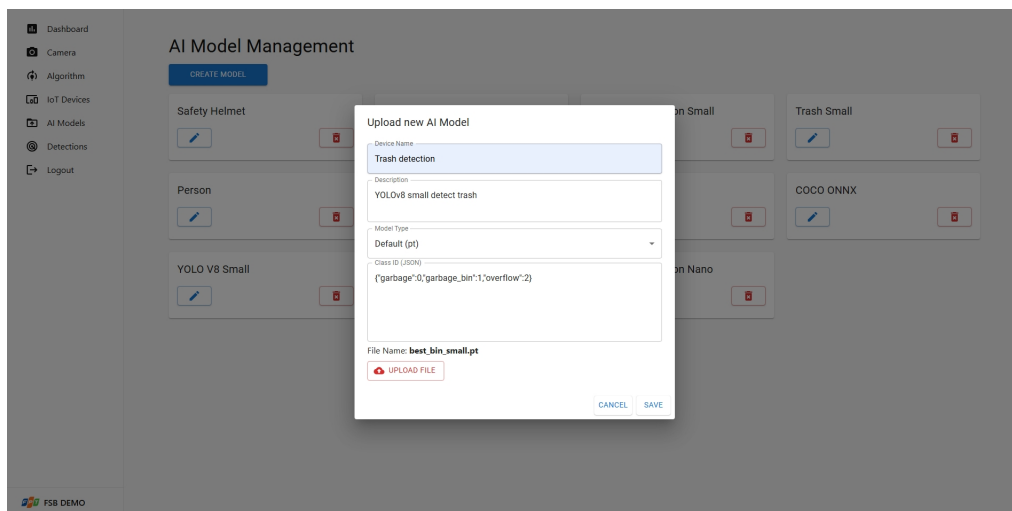


Figure 7.4: Upload new AI model

Listing 7.6: Request To Add AI Model

```
1 curl --location 'localhost:3000/ai-models' \  
2 --header 'Authorization: Bearer  
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9... ' \  
3 --form 'name="YOLO V8 Nano" ' \  
4 --form 'description="YOLO V8 Nano" ' \  
5 --form 'model_type="Default (pt)" ' \  
6 --form 'class_id="{\"garbage\":0,\"garbage_bin\":1,\"overflow\":2}" ' \  
7 --form 'file=@best_bin_small.pt'
```

```

4 --form 'description="YOLO V8 Nano"' \
5 --form 'model_type="default"' \
6 --form 'class_id="{\"person\":0,\"bicycle\":1,\"car\":2,\"
  motorcycle\":3,...}\"' \
7 --form 'file=@"/path/to/file/yolov8n.pt"'

```

Listing 7.7: Get User Profile Success Response Body

```

1 {
2   "class_id": {
3     "person": 0,
4     "bicycle": 1,
5     "car": 2,
6     "motorcycle": 3,
7     ...
8   },
9   "id": "0080169b-8dd5-4b97-aca5-c3d9be268a79",
10  "name": "YOLO V8 Nano",
11  "description": "YOLO V8 Nano",
12  "model_type": "default",
13  "model_name": "yolov8n_1735431919153.pt",
14  "updated_at": "2024-12-29T00:25:19.239Z",
15  "created_at": "2024-12-29T00:25:19.239Z"
16 }

```

### 7.2.3 Add customized algorithms

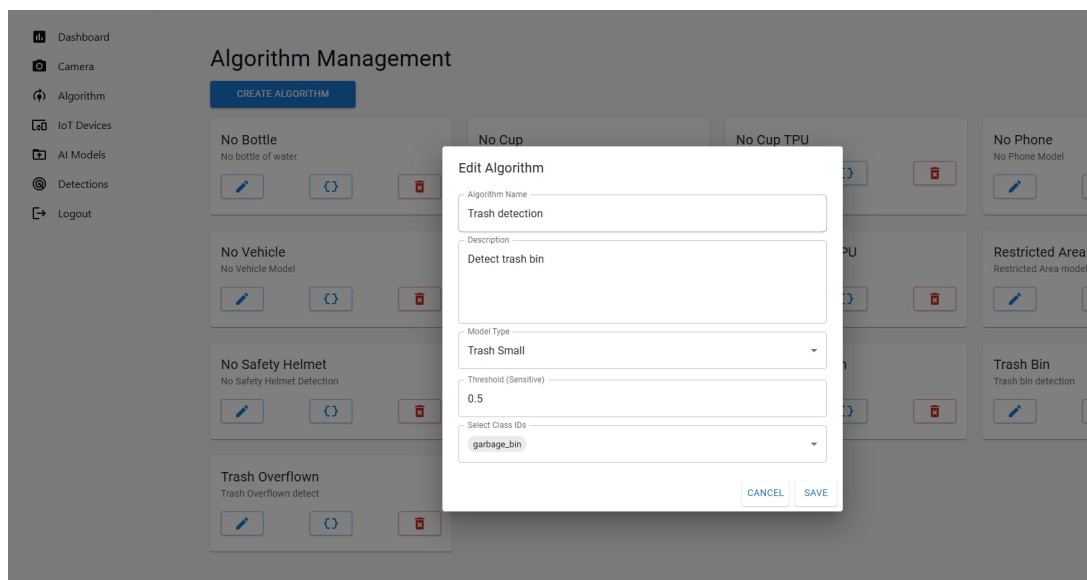


Figure 7.5: Upload new AI model

Listing 7.8: Request To Add Algorithm

```

1 curl --location 'localhost:3000/algorithms' \
2 --header 'Content-Type: application/x-www-form-urlencoded' \
3 --header 'Authorization: Bearer
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...' \
4 --data-urlencode 'id=restricted_area' \
5 --data-urlencode 'name=Restricted Area' \
6 --data-urlencode 'description=Restricted Area' \
7 --data-urlencode 'configs={"model":"yolov8n_1735431919153.pt
   ","threshold":0.8,"class_id":[1]}'

```

Listing 7.9: Add Algorithm Success Response Body

```

1 {
2   "configs": {
3     "model": "yolov8n_1735431919153.pt",
4     "threshold": 0.8,
5     "class_id": [
6       1
7     ]
8   },
9   "id": "restricted_area",
10  "name": "Restricted Area",
11  "description": "Restricted Area",
12  "updated_at": "date_time_ISO_format",
13  "created_at": "date_time_ISO_format"
14 }

```

## 7.2.4 Add IP cameras

Listing 7.10: Request To Add IP Camera

```

1 curl --location 'localhost:3000/cameras' \
2 --header 'Content-Type: application/x-www-form-urlencoded' \
3 --header 'Authorization: Bearer
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...' \
4 --data-urlencode 'name=Construction 2' \
5 --data-urlencode 'uri=rtsp://uri_ip_camera' \
6 --data-urlencode 'mode=alert' \
7 --data-urlencode 'lat=camera_latitude' \
8 --data-urlencode 'long=camera_longitude' \
9 --data-urlencode 'algorithms=["restricted_area"]' \
10 --data-urlencode 'status=INACTIVE' \
11 --data-urlencode 'zones=[]' \
12 --data-urlencode 'snapshot='

```

Listing 7.11: Add IP Camera Success Response Body

```

1 {
2   "algorithms": [

```



```

3         "restricted_area"
4     ],
5     "zones": [],
6     "id": "1d62e5d9-c8ba-4c74-b065-2458b47c3a71",
7     "name": "Construction 2",
8     "uri": "rtsp://uri_ip_camera",
9     "mode": "alert",
10    "lat": camera_latitude,
11    "long": camera_longitude,
12    "status": "INACTIVE",
13    "snapshot": "minio_host/snapshots/snapshot.png",
14    "created_at": "date_time_ISO_format",
15    "updated_at": "date_time_ISO_format"
16 },

```

## 7.2.5 Add IoT devices

Listing 7.12: Request To Add IoT Device

```

1 curl --location 'localhost:3000/iot-devices' \
2 --header 'Content-Type: application/json' \
3 --header 'Authorization: Bearer
4     eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9... ' \
5 --data '{
6     "name": "Raspberry Pi 4B",
7     "cameras": "[\"1d62e5d9-c8ba-4c74-b065-2458b47c3a71\"]",
8     "ip": "device_ip",
9     "port": device_port,
10    "username": "ssh_username",
11    "password": "ssh_password",
12    "deployment_path": "path/to/deploy/ai_models"
13 }',

```

Listing 7.13: Add IoT Device Success Response Body

```

1 {
2     "cameras": [
3         "1d62e5d9-c8ba-4c74-b065-2458b47c3a71"
4     ],
5     "id": "dc62bdc8-1f4b-4436-9ea3-232b837cefef",
6     "name": "Raspberry Pi 4B",
7     "os": "linux",
8     "ip": "device_ip",
9     "port": device_port,
10    "username": "ssh_username",
11    "password": "ssh_password",
12    "deployment_path": "path/to/deploy/ai_models",
13    "created_at": "date_time_ISO_format",
14    "updated_at": "date_time_ISO_format"

```

15 }

## 7.2.6 Deploy/Stop Model

Listing 7.14: Request To Deploy Algorithms on IoT Device

```
1 curl --location --request POST 'localhost:3000/iot-devices/  
  deploy/dc62bdc8-1f4b-4436-9ea3-232b837cefeb' \  
2 --header 'Authorization: Bearer  
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'
```

Listing 7.15: Request To Stop Process on IoT Device

```
1 curl --location --request POST 'localhost:3000/iot-devices/  
  stop/dc62bdc8-1f4b-4436-9ea3-232b837cefeb' \  
2 --header 'Authorization: Bearer  
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'
```

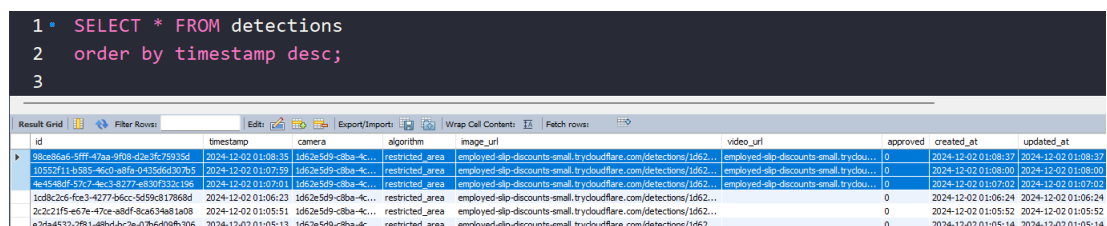
Listing 7.16: Deploy Algorithms/Stop Process on IoT Device Success Response Body

```
1 { result: 'success' }
```

Listing 7.17: Deploy Algorithms/Stop Process on IoT Device Fail Response Body

```
1 {  
2   "message": "Bad Request Exception",  
3   "statusCode": 400,  
4   "timestamp": "date_time_ISO_format"  
5 }
```

## 7.2.7 Receive data from IoT devices



The screenshot shows a SQL query editor with the following query:

```
1 SELECT * FROM detections  
2 order by timestamp desc;  
3
```

Below the query editor is a table with the following columns: id, timestamp, camera, algorithm, image\_url, video\_url, approved, created\_at, and updated\_at. The table contains several rows of data, with the first row highlighted in blue.

id	timestamp	camera	algorithm	image_url	video_url	approved	created_at	updated_at
98ce8a6-9ff4-7aa-9f08-d5e3fc759354	2024-12-02 01:08:35	1d62e5d9-c8ba-4c...	restricted_area	employed-slp-discounts-small.trycloudflare.com/detections/1d62...	employed-slp-discounts-small.trydou...	0	2024-12-02 01:08:37	2024-12-02 01:08:37
10552f11-b585-46c0-a8fa-0435d6d30705	2024-12-02 01:07:59	1d62e5d9-c8ba-4c...	restricted_area	employed-slp-discounts-small.trycloudflare.com/detections/1d62...	employed-slp-discounts-small.trydou...	0	2024-12-02 01:08:00	2024-12-02 01:08:00
4e4548df-57c7-4ec3-8277-e830f332196	2024-12-02 01:07:01	1d62e5d9-c8ba-4c...	restricted_area	employed-slp-discounts-small.trycloudflare.com/detections/1d62...	employed-slp-discounts-small.trydou...	0	2024-12-02 01:07:02	2024-12-02 01:07:02
1cd82c6-fce3-4277-b6cc-5d59c817868d	2024-12-02 01:06:23	1d62e5d9-c8ba-4c...	restricted_area	employed-slp-discounts-small.trycloudflare.com/detections/1d62...		0	2024-12-02 01:06:24	2024-12-02 01:06:24
2c2c21f5-e67e-47ce-a8df-8ca634a81a08	2024-12-02 01:05:51	1d62e5d9-c8ba-4c...	restricted_area	employed-slp-discounts-small.trycloudflare.com/detections/1d62...		0	2024-12-02 01:05:52	2024-12-02 01:05:52
e2da4532-2f81-48bd-bc2e-07b6d09fb306	2024-12-02 01:05:13	1d62e5d9-c8ba-4c...	restricted_area	employed-slp-discounts-small.trycloudflare.com/detections/1d62...		0	2024-12-02 01:05:14	2024-12-02 01:05:14

Figure 7.6: New detected events added to detections table

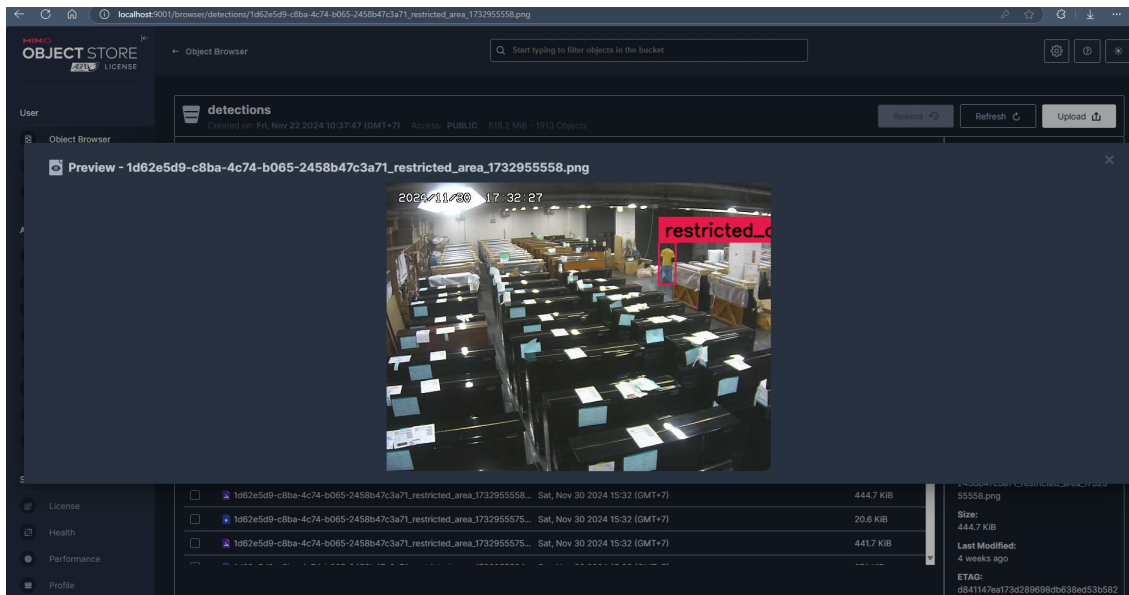


Figure 7.7: Detected events media file stored in MinIO

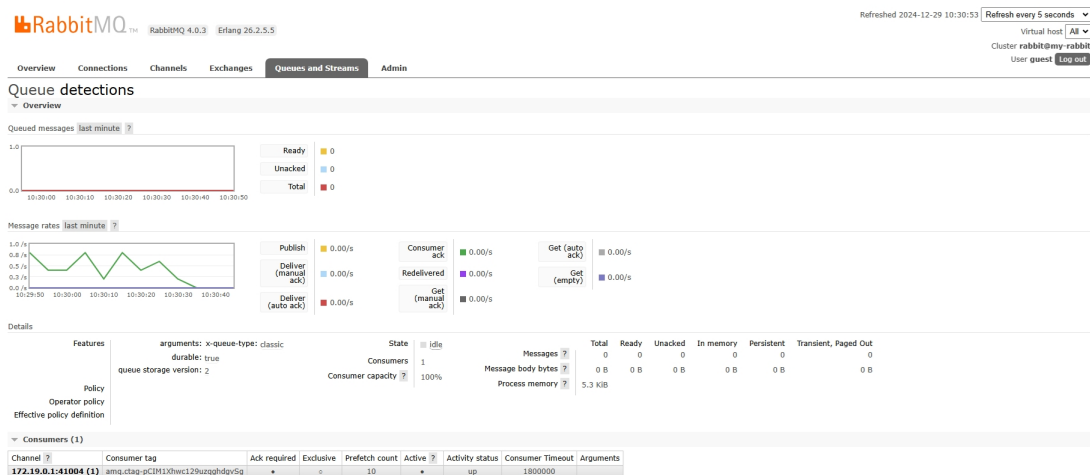


Figure 7.8: Detected events sent through RabbitMQ

## 7.3 Key Findings

- **Workflow Success:** Successfully demonstrated the workflow from adding customized AI models to receiving events detected by IoT devices.
- **Customized Models:** Illustrates some object detection models in real life scenario

# Chapter 8

## Discussion

### 8.1 Results Analysis

#### 8.1.1 Architectural Analysis

In our IoT surveillance monitoring application scenario, the Edge Computing architecture with RabbitMQ, MinIO, and MySQL has proven its robustness and efficiency. RabbitMQ handled the detected events data with high volume from IoT devices. Its reliable message queuing ensured no data loss even under peak loads and could be considered as a buffer between data producers (IoT devices) and consumers (central service). MinIO, on the other hand, served as the media storage for processed videos and images. Its scalability and cost-efficiency plays an important role in deploying surveillance systems on-premise. Finally, MySQL is used for metadata management about detected events, devices, and AI models. In this research scope, this technology is sufficient for the intensity of the workload from detection tasks of IoT devices.

#### 8.1.2 Performance of Customized Models

Besides Edge Computing architecture, the results from training YOLO models with customized datasets also have some noteworthy figures. With only about 20-40 epochs, some models have impressive precision of more than 80%. Perhaps most impressively, even with nano size models (6 MB approximately), the accuracy still remains high, which can reduce

workload on edge devices when performing detection tasks.

## 8.2 Comparative Analysis with Existing Solutions

### 8.2.1 Architectural Advantages

- **New architecture solution compared to existing surveillance:** Most of current surveillance systems follow cloud-based architecture to analysis video footages sending from IP cameras which could be an insecured monitoring system if the configuration is not set properly. With edge AI architecture, this solution already has ability to minimize to security risk by processing raw data at the source.
- **Enhanced Scalability and Flexibility:** Traditional systems use monolithic architectures with limited scalability. In contrast, our approach separates data ingestion, storage, and metadata management, enabling each component to scale independently. RabbitMQ efficiently handles data streams, MinIO offers vast storage capacity, and MySQL scales for metadata queries. This modular design also simplifies future feature and technology integration [34].
- **Improved Reliability and Data Integrity:** RabbitMQ ensures reliable delivery of surveillance data, even during network issues, preventing data loss common in direct-to-storage systems. MinIO's object storage provides high durability and redundancy, protecting valuable surveillance footage [35][36].
- **Cost-Effectiveness:** MinIO's open-source design and compatibility with commodity hardware lower storage costs compared to proprietary solutions. Its scalable architecture supports a pay-as-you-grow model, reducing upfront investment and optimizing resource use [31].
- **Integration with Existing Infrastructure:** With open-source technologies like MinIO, MySQL, and RabbitMQ, the integration between these services and existing applications is simplified, which also helps reduce the cost of integration operations.

## 8.3 Lessons Learned

### 8.3.1 Implementation Challenges

#### Challenge 1: The setup operations

- **Problem:** With multiple IoT devices, there would be scenarios having multiple operating systems (OS) installed on each device. This causes difficulties in handling step-by-step operations to install prerequisite software for the edge devices.
- **Solution:** Writing a general shell script which would work on most common OS types, the script checks the OS type and then executes the correct install commands for each type of operating system.
- **Result:** Successfully pre-installed packages for both Linux and Windows on IoT devices, which makes the application more well-integrated with other operating systems.

## 8.4 Future Improvements

### 8.4.1 Technical Enhancements

- **Enhanced Data Consuming Speed:** Improving the ingestion performance of the data from the RabbitMQ service could be done in multiple methods, but basically 2 solutions:
  - **Scale Up:** Increasing the resources for the central service to provide more consume power for the machine to process the messages faster.
  - **Scale Out:** Separates the consuming component into a functional service. We can make use of open-source Function as a Service (FaaS) software like OpenFaaS, Knative, etc. to improve the scalability of the consume functionality component which will resolve the issue about high load of messages [37][38].
- **Fault Tolerance:**

- **Disconnect Recovery:** The system could handle to process footages from IP cameras while being disconnected to storage and message queue services, and push all failed sent data when the connection is back
- **System logging:** Every services such as Central Service, IoT devices, Message Queue, Storage, etc. have their own system logs for analysis when any fault happen to come up with better strategy when deploying services.
- **Improve data transmission process:**
  - **Optimized Protocols:** Implementing more efficient data transmission protocols such as MQTT or CoAP can reduce latency and improve the reliability of data transfer between IoT devices and the central service.
  - **Data Compression:** Utilizing data compression techniques to minimize the size of the data being transmitted can significantly reduce bandwidth usage and speed up the transmission process.
  - **Edge Data Aggregation:** Aggregating data at the edge before transmission can reduce the number of messages sent to the central service, thereby optimizing the network load and improving overall system performance.
  - **Quality of Service (QoS) Levels:** Implementing different QoS levels for data transmission can ensure that critical data is prioritized and delivered reliably, even in the presence of network congestion or failures.
  - **Adaptive Transmission Rates:** Dynamically adjusting the data transmission rates based on network conditions and system load can help maintain optimal performance and prevent network bottlenecks.

## 8.4.2 Architectural Improvements

### 1. Hybrid Computing Architecture:

- **Optimized Workload Distribution:** The edge architecture handles fast, real-time processing, while cloud services take care of slower, less urgent tasks like

long-term storage, big data analysis, and AI model training. This architecture keeps edge devices running detections smoothly while using the cloud services' power for complex work.

- **Enhanced Data Accessibility:** A hybrid setup lets you easily access data everywhere. Data processed at the edge is backed up to the cloud, providing central monitoring, reports, and overall system understanding without slowing down real-time edge performance.
- **Disaster Recovery and Redundancy:** The hybrid model makes the system more reliable by backing up data in the cloud. If hardware fails or there's a local problem, important data is still safe and available, keeping the system running and minimizing downtime.

## 2. Scalability Enhancements:

- **Containerization with Kubernetes (K8s):** Using Kubernetes to manage applications in containers across both edge and cloud makes it easy to scale resources. Containers keep application deployments consistent, while Kubernetes automatically handles distributing work, scaling up or down, and failover operations, so the system can handle changing workloads efficiently.
- **Cloud-Native Scaling:** Using cloud tools like AWS Auto Scaling, Google Kubernetes Engine (GKE), or Azure Kubernetes Service (AKS) ensures cloud resources automatically adjust to demand. This flexibility handles demanding tasks like data analysis or AI training without requiring manual intervention.
- **Dynamic Load Balancing:** Using smart traffic directors like NGINX or Traefik spreads traffic evenly between the edge and cloud. This prevents slowdowns and improves overall system performance, especially during high-volume operations.
- **Microservices Architecture:** Switching to a microservices design means each component of the system can scale independently. This modular approach



makes the system more flexible, allowing specific services to handle different workloads without affecting other components.

## **8.5 Research Limitations**

### **8.5.1 Technical Limitations**

This research had several technical limitations. A wide variety of IoT devices was not available for testing, which means it is uncertain whether the system could work across all hardware types. Also, the test environment was not a perfect replica of a real-world setup, so factors like network issues or unexpected problems could not be fully accounted for. A limited number of cameras was available for load testing, making it difficult to determine system performance under heavy load. Finally, testing on real production servers was not possible, so performance measurements might not be entirely accurate. These limitations highlight areas for future testing improvements, such as the need for more diverse devices, better simulations, and stronger testing infrastructure.

### **8.5.2 Methodology Constraints**

The research methodology had several limitations that could affect how well the results apply to real-world situations. First, the workflow tests were too simplified and did not fully represent the complexity of real applications, which often have complicated interactions and unpredictable user behavior. Second, the research relied heavily on vendor documentation for evaluating component reliability, and there was insufficient time for thorough testing. This means the actual performance of these components in various situations remains uncertain. Finally, testing did not cover large-scale deployments, so potential problems with hundreds or thousands of devices might not have been discovered. Future work needs more comprehensive testing, including more complex workflows, longer reliability tests, and large-scale simulations to better ensure the system's robustness.

# Chapter 9

## Conclusion

### 9.1 Summary of Key Achievements

This IoT Surveillance Monitoring solution has achieved some significant results:

- **Workflow Proven**

- A complete workflow for an IoT Surveillance application worked as expected, from uploading customized AI models, configuring algorithms for detection, to deploying algorithms to IoT devices and receiving data back detected from the IP cameras
- Having an on-premise object storage for keeping media files like detected images, videos, snapshots from IP cameras, etc.

- **Customized AI Models**

- Able to train YOLO v8 models with customized datasets to provide detection tasks suitable for real-life demands
- Can train models with high accuracy with limited model size

## **9.2 Research Impact**

### **9.2.1 IoT Surveillance Monitoring Implications**

The findings of this research have several key implications for future IoT Surveillance Monitoring deployments:

#### **1. On-premise Solution**

- This research has proven the possibility of deploying an IoT Surveillance system on-premise with in-house infrastructure.
- There are many open-source technologies which can adapt well for surveillance purposes, making the cost more efficient when deploying the system
- The ability to scale out the platform also provides solutions for large-scale surveillance projects

#### **2. Real-Life Use Cases**

- With useful and correct datasets, this platform can adapt to multiple real-life use cases which support many surveillance tasks
- The solution also supports real-time detection which makes the monitoring job more effective and allows supervisors to make suitable decisions when events happen

#### **3. Economic Efficiency**

- With Edge Computing Architecture, the infrastructure can make use of in-house facilities which could significantly reduce costs since organizations could avoid the pay-as-you-go payment model from cloud providers

## 9.3 Future Research Directions

### 9.3.1 Technical Enhancements

The researchs in the future could follow:

- **Hybrid Computing Architecture**

- Enhanced Scalability and Flexibility.
- Improved Reliability and Data Integrity.
- Cost-Effectiveness.
- Integration with Existing Infrastructure.

- **Enrich Machine Learning Process**

- Make use of detected images and videos to enrich the datasets, which in part will improve the models through training.
- With detected events, can analyze user behavior to better understand patterns of actions, for example, before crimes are committed.
- Unused data could be a great resource for the public, especially in the machine learning community.

- **Security Improvement**

- Data encryption is important in IoT environments, especially when devices usually connect to each other wirelessly, which could be easily interfered with.
- Design more secure workflows and architecture to prevent hackers or outsiders from interrupting transmission between devices.
- Enhance authentication and authorization to provide only enough permissions for each type of user; also, an audit service is necessary to track users' activities in the system.

### 9.3.2 Recommended Research Areas

#### 1. Integration Studies

- **Integration with existing security infrastructure:** Research could head to seamless integration with existing security systems like CCTV networks, access control systems, and alarm systems.
- **Cross-Platform compatibility for surveillance devices:** The compatibility with a wider range of surveillance devices: IP cameras (various manufacturers and resolutions), smart watches, motion detectors, smart locks, and even drones. Research should investigate standardized data formats and communication protocols.

#### 2. Scalability And Performance

- **Microservices architecture for scalability:** Researchs about microservices can resolve the problem of scaling services to adapt high load of data from IoT devices.
- **Load balancing:** Research should investigate more sophisticated distributed architectures for large-scale surveillance deployments. This includes techniques for distributing processing and storage across multiple servers or edge devices, ensuring system's performance.

#### 3. Societal and Ethical Impact

- **Privacy-Preserving Surveillance Techniques:** Research should explore privacy-enhancing technologies (PETs) like differential privacy, federated learning, and homomorphic encryption to enable surveillance while minimizing privacy risks.
- **Transparency and Accountability in Surveillance Systems:** Future research could investigate ways to log and audit access to and use of surveillance data, promoting responsible and ethical deployment of these technologies. This includes developing clear guidelines and regulations for data retention, access control, and how surveillance data is used.

## **9.4 Final Remarks**

### **9.4.1 Closing Recommendations**

The following recommendations are offered for practitioners and researchers building upon this work.:

#### **1. Enhance Monitoring Workflow**

- Implement system logs to monitor activities happening over time in the system. Not only does this help in auditing, but it also makes it easier to find issues when the system is not working as expected.
- Focus on workflow implementation. A useful application is one that can adapt to users' needs, not only one with good performance.
- Try to implement this architecture at a large scale to test its limitations and begin improving upon them.

#### **2. Technology Selection**

- Based on the project's requirements, select the most suitable technologies. It is best if these technologies have a strong support community or are being used by many large organizations.
- Always plan for long-term support. With this mindset, the project will have enough resources for future scale expansion and maintenance.

#### **3. Improvement Research**

- Research optimizing the AI model to detect faster, have a smaller model size, consume fewer resources, etc.
- Explore more real-life use cases to diversify the applicability of the surveillance system.
- Ethical investigation also needs to be prioritized because user privacy awareness has increased in recent years.

The future of IoT surveillance monitoring lies in the continued evolution of scalable, reliable, and cost-effective systems that can adapt to the growing demands of security and public safety while maintaining operational efficiency and respecting societal and ethical considerations (such as privacy).

# Chapter 10

## Improvements

Since the IoT Surveillance system will continue to grow, more advanced technologies and strategies can considerably boost performance, scalability, flexibility, and user experience. To that end, we propose the following enhancements to the current system's functionality and capabilities:

### 10.1 Enhanced Scalability with Kubernetes

Kubernetes is a powerful tool for achieving scalability in modern systems, streamlining the deployment, scaling, and management of containerized applications [39]. By orchestrating these processes, it enables applications to automatically adjust their capacity—scaling up or down in response to changing demand. This smart workload distribution across a cluster of nodes ensures resources are used efficiently, allowing the system to handle varying workloads with ease. The result is a scalable, resilient platform that adapts seamlessly to real-world demands.

### 10.2 Hybrid Architecture for Surveillance Monitoring

A hybrid architecture for surveillance monitoring leverages both on-premises and cloud resources to optimize performance, cost, and security. By processing data closer to the source at the edge, it reduces latency and bandwidth consumption, enabling real-time analysis and



faster response times. Simultaneously, the cloud provides scalable storage, centralized management, and powerful computing resources for long-term data retention, complex analytics, and machine learning, offering a balanced approach that addresses the diverse needs of modern surveillance systems. This hybrid approach allows organizations to tailor their deployments to specific requirements, balancing local processing power with the vast capabilities of the cloud.

### **10.3 Enhance Security with Keycloak**

Keycloak elevates security by offering centralized authentication and authorization services, acting as a gatekeeper to control access based on user roles and permissions [40]. It supports widely used protocols like OpenID Connect and OAuth 2.0, enabling single sign-on (SSO) and seamless integration with existing identity providers. Advanced features such as multi-factor authentication (MFA) and fine-grained access control further enhance security, safeguarding sensitive surveillance data from unauthorized access. By consolidating identity management, Keycloak not only strengthens the overall security posture but also simplifies administration, reducing the risks associated with managing authentication across diverse systems.

### **10.4 Stream Processing with Apache Spark**

Stream processing with Apache Spark [20] enables real-time analysis of high-velocity data streams generated by surveillance systems. It enables real-time operations by processing it in small batches. This allows for quick detection of anomalies and threats. Spark's scalable design handles large amounts of data efficiently, enabling proactive monitoring and faster responses to security events.

## **10.5 System Logs with Signoz**

System logs, especially when managed with a tool like Signoz [41], provide crucial insights into the operational health and performance of a surveillance system. Signoz offers a comprehensive platform for collecting, processing, and visualizing logs, metrics, and traces, enabling effective monitoring and troubleshooting. By centralizing logs from various components, including edge devices, backend services, and databases, Signoz facilitates rapid identification of errors, performance bottlenecks, and security incidents. Its advanced querying and visualization capabilities allow for in-depth analysis of system behavior over time, aiding in proactive maintenance, performance optimization, and security auditing. Detailed logging with Signoz enhances system observability, making it easier to diagnose issues, improve reliability, and ensure the integrity of surveillance data.

## **10.6 Manage Time-Series Data with ClickHouse**

Managing time-series data with ClickHouse [30] provides a robust and efficient solution for handling the high volume of time-stamped data generated by surveillance systems. ClickHouse's column-oriented database architecture and optimized data compression enable extremely fast query performance on large datasets, making it ideal for analyzing trends, patterns, and anomalies in surveillance data over time. Its ability to handle high write throughput and performant analytical queries allows for real-time monitoring, historical data analysis, and the generation of insightful reports. By effectively managing time-series data, ClickHouse empowers deeper understanding of surveillance events and facilitates data-driven decision-making.

## **10.7 CI/CD Pipeline with Gitlab CI**

A CI/CD (Continuous Integration/Continuous Deployment) pipeline with GitLab CI [42] automates the software development lifecycle, ensuring faster and more reliable releases for surveillance systems. GitLab CI enables automated building, testing, and deployment

of code changes, reducing manual effort and minimizing the risk of human error. By automating these processes, the pipeline facilitates frequent integration of code changes, early detection of bugs, and rapid delivery of new features and updates. This streamlined development process enhances agility, improves software quality, and enables faster response to evolving surveillance needs. A well-defined CI/CD pipeline with GitLab CI ensures consistent and predictable deployments, leading to a more stable and reliable surveillance system.

# Bibliography

- [1] G. F. Shidik *et al.*, “A systematic review of intelligence video surveillance: Trends, techniques, frameworks, and datasets,” *IEEE Access*, vol. 7, pp. 170 457–170 473, 2019, ISSN: 21693536. DOI: 10 . 1109 / ACCESS . 2019 . 2955387. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/8911368>.
- [2] M. M. Lee K. Yim K., “A secure framework of the surveillance video network integrating heterogeneous video formats and protocols,” vol. 63, pp. 525–535, Jan. 2012, ISSN: 08981221. DOI: 10 . 1016 / j . camwa . 2011 . 08 . 048. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0898122111007218>.
- [3] M. K. Lim *et al.*, “Isurveillance: Intelligent framework for multiple events detection in surveillance videos,” *Expert Systems with Applications*, vol. 41, pp. 4704–4715, 10 Aug. 2012, ISSN: 09574174. DOI: 10 . 1016 / j . eswa . 2014 . 02 . 003. [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/S0957417414000530>.
- [4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You only look once: Unified, real-time object detection*, 2016. arXiv: 1506.02640 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1506.02640>.
- [5] J. Redmon and A. Farhadi, *Yolov3: An incremental improvement*, 2018. arXiv: 1804.02767 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1804.02767>.
- [6] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, *Yolov4: Optimal speed and accuracy of object detection*, 2020. arXiv: 2004.10934 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2004.10934>.

- [7] Z. Huang, J. Wang, X. Fu, T. Yu, Y. Guo, and R. Wang, “Dc-spp-yolo: Dense connection and spatial pyramid pooling based yolo for object detection,” *Information Sciences*, vol. 522, pp. 241–258, Jun. 2020, ISSN: 0020-0255. DOI: 10.1016/j.ins.2020.02.067. [Online]. Available: <http://dx.doi.org/10.1016/j.ins.2020.02.067>.
- [8] J. Huang *et al.*, *Speed/accuracy trade-offs for modern convolutional object detectors*, 2017. arXiv: 1611.10012 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1611.10012>.
- [9] R. Reynold, *Achieving “five nines” in the cloud for justice and public safety*, 2020. [Online]. Available: <https://aws.amazon.com/blogs/publicsector/achieving-five-nines-cloud-justice-public-safety/>.
- [10] B. Bajic, I. Cosic, B. Katalinic, S. Moraca, M. Lazarevic, and A. Rikalovic, “Edge computing vs. cloud computing: Challenges and opportunities in industry 4.0,” in *Annals of DAAAM and Proceedings of the International DAAAM Symposium*, vol. 30, Danube Adria Association for Automation and Manufacturing, DAAAM, 2019, pp. 864–871. DOI: 10.2507/30th.daaam.proceedings.120.
- [11] I. Odun-Ayo, M. Ananya, F. Agono, and R. Goddy-Worlu, “Cloud computing architecture: A critical analysis,” in *2018 18th International Conference on Computational Science and Applications (ICCSA)*, 2018, pp. 1–7. DOI: 10.1109/ICCSA.2018.8439638.
- [12] SQLite, *Sqlite database*. [Online]. Available: <https://www.sqlite.org/>.
- [13] InfluxDB, *Influxdb: The leading platform for time series data*. [Online]. Available: <https://www.influxdata.com/>.
- [14] S. M. Alamouti, F. Arjomandi, and M. Burger, “Hybrid edge cloud: A pragmatic approach for decentralized cloud computing,” *IEEE Communications Magazine*, vol. 60, no. 9, pp. 16–29, 2022. DOI: 10.1109/MCOM.001.2200251.
- [15] RabbitMQ, *Rabbitmq: One broker to queue them all*. [Online]. Available: <https://www.rabbitmq.com/>.

- [16] CloudAMQP, *Rabbitmq exchanges, routing keys and bindings*. [Online]. Available: <https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html>.
- [17] RabbitMQ, *Rabbitmq plugins*. [Online]. Available: <https://www.rabbitmq.com/docs/plugins>.
- [18] A. Kafka, *Apache kafka*. [Online]. Available: <https://kafka.apache.org/>.
- [19] A. Flink, *Apache flink: Stateful computations over data streams*. [Online]. Available: <https://flink.apache.org/>.
- [20] A. Spark, *Apache spark: Unified engine for large-scale data analytics*. [Online]. Available: <https://spark.apache.org/>.
- [21] A. D. Pathak, "Internet of things: Quality of services of rabbitmq & kafka," *International Journal of Innovative Technology and Exploring Engineering*, 2019. DOI: 10.35940/ijitee.B7118.129219. [Online]. Available: <https://www.ijitee.org/portfolio-item/B7118129219/>.
- [22] A. W. Services, *Amazon simple queue service: Fully managed message queuing for microservices, distributed systems, and serverless applications*. [Online]. Available: <https://aws.amazon.com/sqs/>.
- [23] Microsoft, *Azure event hubs: A real-time data streaming platform with native apache kafka support*. [Online]. Available: <https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about>.
- [24] Redis, *Redis pub/sub*. [Online]. Available: <https://redis.io/docs/latest/develop/interact/pubsub/>.
- [25] MySQL, *Mysql*. [Online]. Available: <https://www.mysql.com/>.
- [26] T. P. G. D. Group, *Postgresql*. [Online]. Available: <https://www.postgresql.org/>.
- [27] M. Foundation, *Mariadb*. [Online]. Available: <https://mariadb.org/>.

- [28] Microsoft, *Microsoft sql server*. [Online]. Available: <https://www.microsoft.com/en-us/sql-server>.
- [29] Oracle, *Oracle database*. [Online]. Available: <https://www.oracle.com/database/>.
- [30] ClickHouse, *Clickhouse: What is clickhouse?* [Online]. Available: <https://clickhouse.com/docs/en/intro>.
- [31] MinIO, *Minio: Aistor*. [Online]. Available: <https://min.io/product/aistor-overview>.
- [32] A. W. Services, *Amazon s3*. [Online]. Available: <https://aws.amazon.com/s3/>.
- [33] MinIO, *Minio: Iot, iiot and storage at the edge*. [Online]. Available: <https://blog.min.io/iot-iiot-and-storage-at-the-edge/>.
- [34] goldenowl, *Monolithic vs microservices: Which is the right architecture?* [Online]. Available: <https://goldenowl.asia/blog/monolithic-vs-microservices>.
- [35] A. C. Bao, *Rabbitmq message durability and persistence*, 2024. [Online]. Available: <https://www.alibabacloud.com/tech-news/a/rabbitmq/4oc45nlwlcdrabbitmq-message-durability-and-persistence>.
- [36] MinIO, *Achieving your data strategy with minio*, 2023. [Online]. Available: <https://blog.min.io/achieving-data-strategy/>.
- [37] A. Ellis, *Openfaas: Serverless functions, made simple*. [Online]. Available: <https://www.openfaas.com/>.
- [38] Knative, *Knative: An open-source enterprise-level solution to build serverless and event driven applications*. [Online]. Available: <https://knative.dev/docs/>.
- [39] Kubernetes, *Kubernetes*. [Online]. Available: <https://kubernetes.io/>.
- [40] Keycloak, *Keycloak: Open source identity and access management*. [Online]. Available: <https://www.keycloak.org/>.
- [41] SigNoz, *Signoz: Opentelemetry-native logs, metrics and traces in a single pane*. [Online]. Available: <https://signoz.io/>.

- [42] G. CI/CD, *Get started with gitlab ci/cd*. [Online]. Available: <https://docs.gitlab.com/ee/ci/>.



# Appendix A

## Technical Specifications

### A.1 Hardware Components

#### A.1.1 IP Camera Specifications

Component	Specification	Quantity
DH-MPT220	Android 9.0 8-core 1.8 GHz CPU 2.0" touch screen 1080p@30 fps recording H.265/H.264 Support 2G/3G/4G Supports IP68	1 unit

Table A.1: Sensor Specifications

#### A.1.2 Computing Hardware

- Edge Devices

1	Raspberry Pi 4 Model B
2	- CPU: Quad-core Cortex-A72 @ 1.5GHz
3	- RAM: 4GB LPDDR4
4	- Storage: 16GB Samsung EVO+ microSD
5	- Network: Gigabit Ethernet
6	- Power: 5V/3A USB-C
7	Quantity: 1 unit

# Appendix B

## Software Configuration

### B.1 System Software

#### B.1.1 Operating System Configuration

```
1 # Raspberry Pi OS (64-bit)
2 VERSION="2024-02-01"
3 KERNEL="5.15.0-1039-raspi"
4 PYTHON_VERSION="3.9.7"
```

#### B.1.2 Development Environment

```
1 # Python Dependencies
2 ultralytics==8.0.196
3 supervision==0.2.1
4 opencv-python~=4.10.0.84
5 imutils~=0.5.4
6 python-dotenv~=1.0.1
7 numpy~=1.26.4
8 torch~=2.4.1
9 onnx
10 onnxruntime
11 pika
12 minio
```

## B.2 Central Service Configuration

### B.2.1 Server Settings

```
1 {  
2   "NodeJS version": "v20.10.0",  
3   "NPM version": "10.2.3"  
4 }
```

# Appendix C

## Code Samples

### C.1 Detect with Customized Algorithms Scripts

Listing C.1: Alert Events Script

```
1  import time
2  import cv2
3  from ultralytics import YOLO
4  import supervision as sv
5  import numpy as np
6  from collections import defaultdict
7  from message_queue import publish_detection
8  from minio_handler import get_image_url
9
10 import os
11 # importing necessary functions from dotenv library
12 from dotenv import load_dotenv
13
14 # loading variables from .env file
15 load_dotenv()
16
17
18 class Alert:
19
20     def __init__(self):
21         self.models = {}
22         self.box_annotators = {}
23
24         target_width, target_height, target_fps, _, _ = self
25             .get_demo_resolution()
26         self.target_width = target_width
27         self.target_height = target_height
28         self.target_fps = target_fps
```

```

28
29 @staticmethod
30 def get_demo_resolution():
31     return int(os.getenv("RESOLUTION_WIDTH")), int(os.
32         getenv("RESOLUTION_HEIGHT")), float(os.getenv("
33         FRAME_RATE")), int(os.getenv("SKIP_FRAMES")), int
34         (os.getenv("MIN_ACCEPT_SEC"))
35
36 @staticmethod
37 def upload_and_publish_detection(camera_id, model_id,
38     image_file_name, video_file_name):
39     image_url = get_image_url(file_name=image_file_name)
40     video_url = get_image_url(file_name=video_file_name)
41     publish_detection(camera=camera_id, algorithm=
42         model_id, image_url=image_url, video_url=
43         video_url)
44
45 def inference_with_yolo(self, url, process_frame,
46     camera_id, model_id, model, box_annotator, algorithm,
47     target_width, target_height, time):
48     result = model(process_frame, agnostic_nms=True)[0]
49     detections = sv.Detections.from_yolov8(result)
50
51     # Filter detections with correct class_id &
52     acceptable threshold
53     class_ids = algorithm["configs"]["class_id"]
54     threshold = algorithm["configs"]["threshold"]
55
56     # Create a mask for the filtering criteria
57     mask = np.isin(detections.class_id, class_ids) & (
58         detections.confidence > threshold)
59
60     # Apply the mask to both detections and track_ids
61     detections = detections[mask]
62
63     # Annotate frame with bounding boxes and labels
64     labels = [
65         f"{model_id} {confidence:0.2f}"
66         for _, confidence, class_id, _ in detections
67     ]
68     process_frame = box_annotator.annotate(
69         scene=process_frame,
70         detections=detections,
71         labels=labels
72     )
73
74     # Save image and publish
75     if detections.__len__() > 0:
76         frame_with_boxes = box_annotator.annotate(
77             scene=process_frame.copy(), # Use a copy to
78             avoid modifying the original image

```

```

68         detections=detections,
69         labels=labels
70     )
71     file_name = f'{camera_id}_{model_id}_' + time +
72         ".png"
73     image_path = 'images/' + file_name
74     cv2.imwrite(image_path, frame_with_boxes)
75
76     self.upload_and_publish_detection(camera_id,
77         model_id, file_name, "")
78
79 def inference_with_onnx(self, url, process_frame,
80     camera_id, model_id, model, algorithm, target_width,
81     target_height, time):
82     # Filter detections with correct class_id &
83     acceptable threshold
84     class_ids = algorithm["configs"]["class_id"]
85     threshold = algorithm["configs"]["threshold"]
86
87     results = model(process_frame, conf=threshold, iou
88         =0.5) [0]
89
90     # Initialize a dictionary to count objects
91     object_counts = defaultdict(int)
92
93     is_detected = False
94
95     # Process the results
96     for result in results.bboxes.data.tolist():
97         x1, y1, x2, y2, score, class_id = result
98
99         if int(class_id) in class_ids and score >
100             threshold:
101             class_name = results.names[int(class_id)]
102             object_counts[class_name] += 1
103
104             cv2.rectangle(process_frame, (int(x1), int(
105                 y1)), (int(x2), int(y2)), (0, 255, 0), 2)
106             label = f'{model_id} {score:.2f}'
107             cv2.putText(process_frame, label, (int(x1),
108                 int(y1) - 10), cv2.FONT_HERSHEY_SIMPLEX,
109                 0.5, (0, 255, 0), 2)
110
111             is_detected = True
112
113     # Annotate the frame with object counts
114     y_offset = 30
115     for class_name, count in object_counts.items():
116         cv2.putText(process_frame, f'{class_name}: {
117             count}', (10, y_offset), cv2.
118             FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255),

```

```

107         2)
108         y_offset += 30
109
110     # Save image and publish
111     if is_detected:
112         file_name = f'{camera_id}_{model_id}_' + time +
113         ".png"
114         image_path = 'images/' + file_name
115         cv2.imwrite(image_path, process_frame)
116
117         self.upload_and_publish_detection(camera_id,
118             model_id, file_name, "")
119
120     def inference(self, url, camera_id, algorithms, cap):
121         for algorithm in algorithms:
122             model_id = algorithm['id']
123
124             # Load model for the algorithm
125             self.models[model_id] = YOLO("models/" +
126                 algorithm["configs"]["model"])
127
128             # Prepare annotators
129             self.box_annotators[model_id] = sv.BoxAnnotator(
130                 thickness=2,
131                 text_thickness=2,
132                 text_scale=1
133             )
134
135     prev = 0
136     target_width, target_height, target_fps = self.
137         target_width, self.target_height, self.target_fps
138
139     while True:
140         time_elapsed = time.time() - prev
141         ret, frame = cap.read()
142
143         if (time_elapsed > 1 / target_fps):
144             prev = time.time()
145
146             if cv2.waitKey(30) == 27:
147                 break
148
149             if(frame is None):
150                 print('Cannot get frame')
151                 continue
152
153             for algorithm in algorithms:
154                 process_frame = frame.copy()
155                 model_id = algorithm['id']
156                 model = self.models[model_id]

```

```

152         box_annotator = self.box_annotators[
153             model_id]
154
155         model_name = algorithm["configs"]["model
156             "]
157         model_format = model_name.split('.')[0]
158         if '.' in model_name else ''
159
160         if model_format == 'pt':
161             self.inference_with_yolo(url,
162                                     process_frame, camera_id,
163                                     model_id, model, box_annotator,
164                                     algorithm, target_width,
165                                     target_height, str(int(prev)))
166         if model_format == 'onnx':
167             self.inference_with_onnx(url,
168                                     process_frame, camera_id,
169                                     model_id, model, algorithm,
170                                     target_width, target_height, str(
171                                     int(prev)))

```

Listing C.2: Tracking Object Script

```

1  import time
2  import cv2
3  from ultralytics import YOLO
4  import supervision as sv
5  import numpy as np
6  from message_queue import publish_detection
7  from minio_handler import get_image_url
8
9  import os
10 # importing necessary functions from dotenv library
11 from dotenv import load_dotenv
12
13 # loading variables from .env file
14 load_dotenv()
15
16
17 class Tracker:
18
19     def __init__(self):
20         self.models = {}
21         self.box_annotators = {}
22         self.tracked_objects = {}
23         self.missing_count = {}
24
25         target_width, target_height, target_fps, skip_frames
26         , minAcceptSec = self.get_demo_resolution()
27         self.target_width = target_width
28         self.target_height = target_height
29         self.target_fps = target_fps

```



```

29         self.skip_frames = skip_frames
30         self.minAcceptSec = minAcceptSec
31
32     @staticmethod
33     def get_demo_resolution():
34         return int(os.getenv("RESOLUTION_WIDTH")), int(os.
35             getenv("RESOLUTION_HEIGHT")), float(os.getenv("
36                 FRAME_RATE")), int(os.getenv("SKIP_FRAMES")), int
37             (os.getenv("MIN_ACCEPT_SEC"))
38
39     @staticmethod
40     def upload_and_publish_detection(camera_id, model_id,
41         image_file_name, video_file_name):
42         image_url = get_image_url(file_name=image_file_name)
43         video_url = get_image_url(file_name=video_file_name)
44         # publish_detection(camera=camera_id, algorithm=
45             model_id, image_url=image_url, video_url=
46             video_url)
47
48     def process_object_frames(self, camera_id, model_id,
49         track_id, time, frames, tracked_objects,
50         missing_count):
51         target_width, target_height, target_fps,
52             minAcceptSec = self.target_width, self.
53             target_height, self.target_fps, self.minAcceptSec
54
55         # Write frames to video
56         video_file_name = f'{camera_id}_{model_id}_{time}' +
57             '.mp4'
58         video_path = 'videos/' + video_file_name
59
60         # Only send video when there are more than
61             MIN_ACCEPT_SEC seconds
62         if (len(frames) < minAcceptSec * target_fps):
63             video_file_name = ""
64             print(f'Event took less than {minAcceptSec}
65                 seconds')
66         else:
67             writer = cv2.VideoWriter(video_path, cv2.
68                 VideoWriter_fourcc(*"avc1"), target_fps, (
69                     target_width, target_height))
70             for i in range(0, len(frames)):
71                 writer.write(frames[i])
72             writer.release()
73
74         # Write frame to image
75         image_file_name = f'{camera_id}_{model_id}_{time}
76             + ".png"
77         image_path = 'images/' + image_file_name
78         cv2.imwrite(image_path, frames[int(len(frames) / 2)
79             ])

```

```

63
64     # Upload & publish detection
65     self.upload_and_publish_detection(camera_id,
66                                     model_id, image_file_name, video_file_name)
67
68     # Remove the processed object's frames from memory
69     del tracked_objects[model_id][track_id]
70     del missing_count[model_id][track_id]
71
72     def inference(self, url, camera_id, algorithms, cap):
73         for algorithm in algorithms:
74             model_id = algorithm['id']
75             # Load model for the algorithm
76             self.models[model_id] = YOLO(
77                 "models/" + algorithm["configs"]["model"])
78
79             # Prepare annotators
80             self.box_annotators[model_id] = sv.BoxAnnotator(
81                 thickness=2,
82                 text_thickness=2,
83                 text_scale=1
84             )
85
86             # Initialize dictionaries to store frames and
87             missing counts
88             self.tracked_objects[model_id] = {}
89             self.missing_count[model_id] = {}
90
91         prev = 0
92         target_width, target_height, target_fps, skip_frames
93         = self.target_width, self.target_height, self.
94         target_fps, self.skip_frames
95
96         while True:
97             time_elapsed = time.time() - prev
98             _, frame = cap.read()
99
100             if (time_elapsed > 1 / target_fps):
101                 prev = time.time()
102
103                 if cv2.waitKey(30) == 27:
104                     break
105
106                 if (frame is None):
107                     print('Cannot get frame')
108                     time.sleep(10)
109                     continue
110
111                 for algorithm in algorithms:
112                     process_frame = frame.copy()
113                     model_id = algorithm['id']

```

```

110     model = self.models[model_id]
111     box_annotator = self.box_annotators[
112         model_id]
113
114     result = model.track(process_frame,
115                          agnostic_nms=True, persist=True, show
116                          =False, tracker='bytetrack.yaml')[0]
117     detections = sv.Detections.from_yolov8(
118         result)
119
120     # Filter detections with correct
121     class_id & acceptable threshold
122     class_ids = algorithm["configs"]["
123         class_id"]
124     threshold = algorithm["configs"]["
125         threshold"]
126
127     # Create a mask for the filtering
128     criteria
129     mask = np.isin(detections.class_id,
130                   class_ids) & (detections.confidence >
131                                threshold)
132
133     # Apply the mask to both detections and
134     track_ids
135     detections = detections[mask]
136
137     track_ids = []
138     if result.bboxes.id is not None:
139         track_ids = result.bboxes.id.tolist()
140     track_ids = [track_id for track_id, keep
141                 in zip(track_ids, mask) if keep]
142
143     current_track_ids = set(track_ids)
144
145     # Update tracked_objects dictionary with
146     current detections
147     for track_id in current_track_ids:
148         if track_id not in self.
149             tracked_objects[model_id]:
150             self.tracked_objects[model_id][
151                 track_id] = []
152             self.missing_count[model_id][
153                 track_id] = 0
154             self.tracked_objects[model_id][
155                 track_id].append(process_frame)
156
157     # Update missing counts for disappeared
158     objects
159     for track_id in list(self.
160                           tracked_objects[model_id].keys()):

```

```

142         if track_id not in current_track_ids
143             :
144                 self.missing_count[model_id][
145                     track_id] += 1
146                 # If missing for more than
147                     SKIP_FRAMES frames -> Send
148                     detection
149                 if self.missing_count[model_id][
150                     track_id] > skip_frames:
151                     self.process_object_frames(
152                         camera_id, model_id,
153                         track_id, str(int(prev)),
154                         self.tracked_objects[
155                             model_id][track_id], self.
156                             .tracked_objects, self.
157                             missing_count)
158             else:
159                 # Reset missing count if the
160                 object reappears
161                 self.missing_count[model_id][
162                     track_id] = 0
163
164         if not self.tracked_objects[model_id]:
165             # print("All tracked ids are saved,
166             reseting model...")
167             model.predictor.trackers[0].reset_id
168             ()

```

## C.2 Performance Testing

Listing C.3: Messages Benchmark Script

```

1 import pika
2 import json
3 import datetime
4 import os
5
6 # importing necessary functions from dotenv library
7 from dotenv import load_dotenv
8
9 # loading variables from .env file
10 load_dotenv()
11
12 RMQ_HOST = os.getenv("RMQ_HOST")
13 RMQ_PORT = int(os.getenv("RMQ_PORT"))
14 RMQ_QUEUE = os.getenv("RMQ_QUEUE")
15
16 def publish_benchmark(queue_name='benchmark'):

```

```

17     try:
18         # Establish a connection to RabbitMQ
19         connection = pika.BlockingConnection(pika.
20             ConnectionParameters(RMQ_HOST, port=RMQ_PORT))
21         channel = connection.channel()
22
23         # Declare the queue
24         channel.queue_declare(queue=queue_name, durable=True)
25
26         for package_index in range(1, 5000):
27             # Create the detection data
28             detection_data = {
29                 "timestamp": datetime.datetime.now().isoformat
30                 (),
31                 "data": package_index,
32             }
33
34             # Publish the message
35             channel.basic_publish(exchange='', routing_key=
36                 queue_name, body=json.dumps(detection_data))
37
38             print(f" [x] Sent {detection_data}")
39
40             # Close the connection
41             connection.close()
42
43             return detection_data
44         except:
45             print('Something wrong happened when transmitting
46                 message')
47
48 # Main loop
49 if __name__ == "__main__":
50
51     publish_benchmark()

```

Listing C.4: Messages Benchmark Logic

```

1 @Public()
2 @RabbitSubscribe({
3     queue: 'benchmark',
4     errorHandler: (channel, msg, error) => {
5         console.error('Error processing message:', error);
6         channel.nack(msg);
7     },
8 })
9 async handleBenchmarkMessage(msg: any) {
10     try {
11         // console.log('Received message:', msg);
12         const { timestamp, data } = msg;
13         const deltaTime = new Date().getTime() - new Date(
14             timestamp).getTime();

```

```
14     console.log(new Date(), new Date(timestamp));
15     console.log('Receive package No.${data} with deta time: ${
      deltaTime} ms');
16
17     this.count_packages++;
18     console.log('Received ${this.count_packages} packages');
19 } catch (error) {
20     throw error;
21 }
22 }
```

# Appendix D

## Technologies Used

### D.1 Hardware

- **Raspberry Pi 4 for Edge Computing:** The Raspberry Pi 4 is a powerful, flexible, and affordable edge computing device. Its quad-core processor, ample RAM, and fast networking make it ideal for local data processing and IoT applications, including sensor preprocessing and cloud communication.
- **IP cameras:** While some advanced IP cameras offer edge processing capabilities like motion detection and basic analytics, many primarily function as data acquisition devices. They capture video and audio, encode it into a digital stream, and transmit it over a network for processing and storage elsewhere

### D.2 Software

- **Node.js 20:** Node.js (version 20) can be deployed on standard servers to handle backend processes, including real-time data processing, event-driven logic, and API interactions. Node.js applications can be triggered by various events, such as sensor data uploads, API requests, or messages from a message broker like Kafka. Node.js's lightweight and high-performance nature allows for efficient handling of I/O operations, making it suitable for processing significant amounts of data.

- **Python for Edge Devices:** Python can be used on edge devices for local data processing and analysis. Python offers a rich ecosystem of libraries like NumPy, SciPy, and OpenCV, suitable for tasks such as sensor data analysis, image processing, and basic machine learning inference. By running these tasks locally on edge devices, Python minimizes latency and bandwidth requirements, enabling faster responses and offline operation.

## D.3 Services Communication

- **RabbitMQ:** RabbitMQ is a reliable message broker that enables different parts of an IoT system to communicate asynchronously. Using the AMQP protocol, it ensures messages are delivered reliably, even with network issues. By acting as a middleman, RabbitMQ separates message senders (like devices) from receivers (like backend services), making the system more flexible and scalable. In IoT, it can collect data, route it for processing, and manage commands. Its high throughput and flexible message handling make it valuable for building scalable and reliable IoT applications.
- **HTTP-based Protocols:** HTTP is a communication protocol suitable for various aspects of IoT systems. It is particularly well-suited for interacting with RESTful APIs, enabling structured communication between devices, backend services, and client applications. Its widespread adoption and support for HTTPS provide a secure channel for data exchange. HTTP is also commonly used for serving web-based interfaces for device management and data visualization. Furthermore, its ability to handle larger data payloads makes it suitable for transmitting media files, firmware updates, and bulk data uploads.

## D.4 Object Storage

- **MinIO:** MinIO is a fast, open-source object storage server that works like Amazon S3. It is flexible and cheaper than cloud storage, and can be used on regular



hardware or virtual machines. Designed for scalability and high availability, it is great for storing large amounts of unstructured data from IoT devices, like sensor readings, images, and videos. Because it's compatible with S3, existing tools and applications can easily use it, simplifying migration and avoiding vendor lock-in. Its high performance allows for fast data storage and retrieval, crucial for real-time IoT applications.

## **D.5 Databases**

- **MySQL:** MySQL is a popular, reliable, open-source database well-suited for storing organized data in many IoT systems. MySQL offers strong data integrity, standard query language (SQL), and can be scaled (by upgrading server resources or distributing data). It's a cost-effective choice for storing sensor data, logs, and other structured IoT data when extreme speed isn't crucial. Its widespread use and support make it a readily available and well-understood option.

# Appendix E

## Project Timeline

The project timeline details the development phases and their estimated completion dates. The planned technologies and methodologies emphasize efficient edge computing architecture, and customized algorithms support systems using Node.js, RabbitMQ, and edge computing.

### E.1 Develop Outline (05/10/2024 - 20/10/2024)

Phase one focuses on finalizing project requirements by collaborating with stakeholders to define key system features like real-time data processing, edge computing architecture, and media files storage. The following technologies are identified for use:

- **IoT Devices and IP cameras:** Dahua body camera and other edge devices like the Raspberry Pi 4.
- **RabbitMQ:** Determine which message queue will be used for services communication.
- **MinIO:** Choosing MinIO as an object storage for all unstructured files, like images, videos, AI models, etc.
- **Edge Computing Framework:** Decide to use Raspberry Pi 4 for data processing at edge side and transmit signals with central service through RabbitMQ.

## E.2 Design (21/10/2024 - 31/10/2024)

The design phase focuses on creating the system architecture and elaborating technical details for both the edge computing layer and the customized algorithm deployment, specifically addressing:

- **System Architecture:** Define the overall system architecture to provide sufficient services for the entire workflow in IoT monitoring tasks.
- **Database Schemas:** Define tables in MySQL to handle most scenarios in a surveillance job environment.
- **Message Queue:** Choose routing strategies for service communication between IoT devices and the central service.
- **Object Storage:** Design a bucket list to identify which types of files will be stored in which bucket.
- **Security:** Implement an authentication and authorization workflow to ensure system security.

## E.3 Development (01/11/2024 - 03/12/2024)

The development phase focuses on the actual implementation of the system. Development tasks include:

- **Edge device setup:** Prepare scripts to install necessary packages and the detection logic scripts.
- **Train customized AI models:** Train YOLO v8 models with customized datasets suitable for surveillance purposes, such as detecting the absence of safety assets or entry into restricted areas.
- **Message queue intergration:** Set up RabbitMQ for communication between IoT devices and the central service, defining queues for individual communication purposes.

- **API Gateway:** Develop an API gateway on the central service for necessary scenarios, such as adding AI models, configuring algorithms, and deploying models to IoT devices.
- **File Storage:** Set up MinIO for storing media files such as detection images and videos, snapshots, etc.
- **Testing and Debugging:** Continuously test Raspberry Pi 4 integration, the RabbitMQ messaging system, data flow to MySQL, media files stored in MinIO, and API Gateway interactions.

## E.4 Testing, Maintenance (04/12/2024 - 20/12/2024)

The testing and maintenance phase is for ensuring the system work as expected in different scenarios. This phase includes:

- **Unit Testing and Integration Testing:** Perform unit tests on individual central service controllers to ensure all API logic is correct according to their design. Test the full workflow, from uploading an AI model to deploying and stopping the detection process on IoT devices.
- **Data Integrity Testing for RabbitMQ:** Test RabbitMQ's capability to retain data even with a large volume of messages sent from IoT services.
- **Fixing Bugs:** Check if the system having any issues any fix them.

## E.5 Reporting (21/12/2024 - 31/12/2024)

The reporting phase focuses on finalizing the research and writing the report. Key tasks include:

- **Preparing final report:** Writing the research report to include all work progress.
- **Preparing demo:** Preparing a demo for the instructor to visualize the final product.

E.6 Gantt Chart

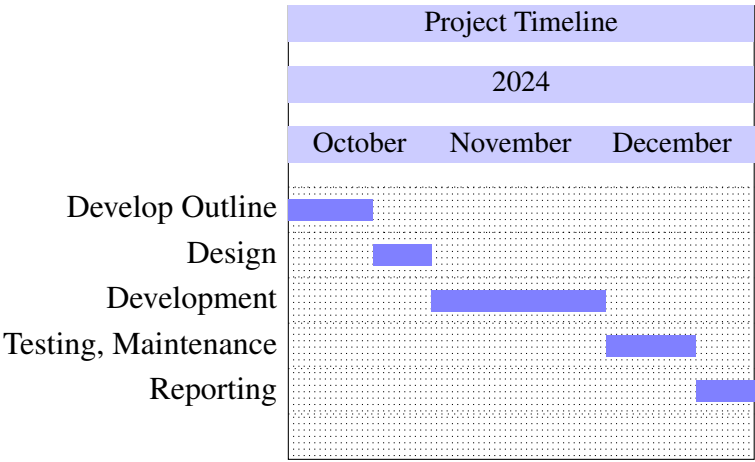


Figure E.1: Project Timeline Gantt Chart