FYP Final Report

Developing an AI System for Smart Traffic: From Vehicle Detection to Data Analytics Dashboard

by

HUANG, I-wei and LI, Yu-hsi

GCH7

Advise by

Prof. Gary Shueng-Han CHAN

Submitted in partial fulfillment of the requirements for
COMP 4981 & CPEG 4901

in the

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

2024-2025

Date of submission: April 18, 2025

Abstract

Smart traffic analytics systems have become increasingly vital in densely populated urban areas that consistently face challenges related to traffic congestion and accidents. While sensor-based solutions are commonly employed to gather traffic information, they often incur high installation and maintenance costs due to the specialized nature of the devices and deployment, which also limits scalability. In contrast, vision-based solutions utilizing deep learning models to analyze video frames from cameras present a cost-effective alternative with all-in-one functionality.

This project aims to develop a computer vision-based smart traffic analytics system and a comprehensive dashboard interface that provides actionable traffic insights for K11 MUSEA administrators. We fine-tuned a YOLO11 model to adapt to Hong Kong's unique environment and integrated it with the ByteTrack algorithm to create a robust vehicle tracking system. Our design includes modular task components for extracting key traffic metrics such as average speed, vehicle count, traffic flow rate, congestion levels, and illegal parking detection. The results will be visualized in a dashboard that incorporates a robust data pipeline and storage solution.

Our system has been rigorously tested and validated, achieving real-time processing capabilities. It is designed to be dynamic and highly adaptable, allowing deployment in various locations based on camera settings and system configurations. This solution offers a scalable, efficient, and adaptable framework that significantly enhances traffic management capabilities, ultimately contributing to smarter urban mobility in Hong Kong.

Table of Content

Abstract **1**

1 Introduction **4**
1.1 Overview . . . 4
1.2 Objective . . . 4
1.3 Literature Survey . . . 5
1.3.1 SMATS Traffic Solutions . . . 5
1.3.2 Q-Free Traffic Monitoring Systems . . . 6

2 Methodology **7**
2.1 Design . . . 7
2.1.1 System Architecture . . . 7
2.1.2 Edge Traffic Analyzer . . . 7
2.1.3 Dashboard Development . . . 11
2.1.4 System Integration . . . 13
2.2 Implementation . . . 13
2.2.1 Edge Traffic Analyzer . . . 13
2.2.2 Dashboard Development . . . 25
2.3 Testing . . . 29
2.3.1 YOLO11 Model Fine-Tuning Validation . . . 29
2.3.2 Edge Traffic Analyzer Testing . . . 30
2.3.3 API Endpoint Testing . . . 31
2.3.4 Dashboard and Database Testing . . . 32
2.4 Evaluation . . . 34
2.4.1 Edge Traffic Analyzer Evaluation . . . 34
2.4.2 Dashboard Evaluation . . . 36

3 Discussion **38**
3.1 YOLO11 Model Fine-Tuning Performance . . . 38
3.2 Dynamic Expendable System with Various Tasks . . . 38
3.3 Illegal Parking Detection Real-world Scenario . . . 39
3.4 Database Implementation . . . 39
3.5 Lower System Workload for Edge Traffic Analyzer . . . 40
3.6 Hardware Limitation during Development Phase . . . 40

4 Conclusion **42**

5 References **43**

6 Appendix A: Meeting Minutes **45**
6.1 Minutes of the 1st Project Meeting . . . 45
6.2 Minutes of the 2nd Project Meeting . . . 46
6.3 Minutes of the 3rd Project Meeting . . . 47
6.4 Minutes of the 4th Project Meeting . . . 49
6.5 Minutes of the 5th Project Meeting . . . 51
6.6 Minutes of the 6th Project Meeting . . . 52

6.7 Minutes of the 7th Project Meeting 53
6.8 Minutes of the 8th Project Meeting 54
6.9 Minutes of the 9th Project Meeting 56
6.10 Minutes of the 10th Project Meeting 57
6.11 Minutes of the 11th Project Meeting 59
6.12 Minutes of the 12th Project Meeting 61
6.13 Minutes of the 13th Project Meeting 62

7 Appendix B:Project Planning 64
7.1 Distribution of Works 64
7.2 GANTT Chart 64

8 Appendix C:Software and Hardware Requirements 65
8.1 Software 65
8.2 Hardware 65

9 Appendix D: Model & Algorithm Structure 66
9.1 YOLO11 Model Structure 66
9.2 ByteTrack 66

1 Introduction

1.1 Overview

Hong Kong, one of the most densely populated cities globally, with an astonishing 38.40 million vehicle kilometers per day [1], consistently faces severe traffic congestion challenges, significantly affecting urban mobility and quality of life. One of the prominent shopping and cultural complexes, K11 MUSEA, is located in the bustling Tsim Sha Tsui district, which often experiences intense traffic congestion on Salisbury Road and its surrounding area near its entrances. This congestion is further exacerbated by incidents of illegal parking, adversely affecting overall traffic flow. Given this situation, K11 MUSEA has identified a critical need for actionable insights to effectively manage congestion and proactively redirect traffic towards alternative parking entrances or routes.

In response to these challenges, the goal of this project, in collaboration with K11 MUSEA, is to develop a smart traffic data analytics system that easily monitors the surrounding area's traffic situation and illegal traffic actions that lead to congestion. This system leverages advanced computer vision technology to accurately extract key traffic metrics in real-time, including vehicle count, average vehicle speed, traffic flow rate, congestion levels, and instances of illegal parking. By capturing and analyzing these critical metrics, the system equips K11 MUSEA administrators with actionable insights, enabling informed decisions to proactively manage and mitigate traffic congestion. The system will be deployed on edge AI cameras, along with a comprehensive traffic data dashboard for the K11 MUSEA authority. Moreover, the system architecture is designed with flexibility in mind, allowing easy configuration and adaptability for deployment at various other locations experiencing similar traffic management issues and incorporating various related traffic analytics tasks.

This initiative aligns with Hong Kong's Smart City Blueprint 2.0 [2], which prioritizes the development of Intelligent Transport Systems (ITS) and advanced traffic monitoring solutions. By integrating deep learning-based computer vision technology with a comprehensive dashboard, the system enables real-time extraction of traffic insights, providing a data-driven foundation for future improvements in urban mobility and smart traffic management.

1.2 Objective

The primary goal of this project is to develop both the system for real-time traffic data analytics deployed on edge devices and a dashboard that serves as the data visualization hub for all the collected traffic data. To achieve these goals, we fulfill the following objectives:

1. **Edge Traffic Analyzer Development**

    1.1. **Developing Vehicle Detection and Tracking System**: To design and implement an AI-powered system capable of detecting and tracking vehicles in live camera video feeds. This involves fine-tuning a YOLO11 model [3] for vehicle detection and integrating the ByteTrack algorithm [4] to maintain consistent tracking of vehicles across frames.

    1.2. **Managing Zone-Specific Tasks**: To design a modular task management system that handles multiple traffic-related tasks within user-defined zones. This ensures scalable and flexible processing of various traffic analytics operations.

1.2.1. **Traffic Metrics Extraction**: To develop algorithms that process tracked vehicle data for real-time traffic analysis. This includes computing average speed, vehicle count, vehicle types, traffic flow rate, and congestion levels, ensuring accurate and dynamic assessment of traffic conditions.

1.2.2. **Illegal Parking Detection**: To implement a system that identifies and alerts for illegal parking instances. This involves designing algorithms to recognize parking violations and integrating alerts into the dashboard for immediate action.

1.3. **Real-Time Data Transmission to Dashboard API Server**: To facilitate seamless and efficient communication between the Edge Traffic Analyzer and the dashboard system. This requires implementing API-based data transmission, ensuring that extracted traffic metrics and illegal parking alerts are accurately sent to the Dashboard API Server via POST requests in real-time.

2. **Dashboard Development**

2.1. **Receiving and Processing Traffic Data**: To establish a robust data pipeline that efficiently receives real-time traffic data from the Edge Traffic Analyzer. This involves managing incoming API requests, validating received data, and ensuring its accurate integration into the dashboard.

2.2. **Developing Real-Time Data Storage and Retrieval System**: To create a backend system that enables low-latency traffic data storage and retrieval, ensuring that real-time updates are consistently available. This includes deploying a Redis in-memory database [5] to handle high-speed data storage and retrieval efficiently.

2.3. **User-Friendly Dashboard for Traffic Insights**: To design and develop an intuitive and robust dashboard that effectively visualizes different task information based on user selection. This involves utilizing the Streamlit Python Framework [6] to build a responsive and accessible web-based interface, providing traffic administrators with real-time insights for decision-making and further analysis.

2.3.1. **Traffic Metrics Visualization**: Display key traffic metrics such as vehicle count, traffic flow rate, average speed, and congestion levels using statistical graphs.

2.3.2. **Illegal Parking Detection Alerts**: Present snapshots of detected illegal parking incidents for quick reference.

1.3 Literature Survey

There have been numerous developments in smart traffic management systems aimed at extracting traffic information with high accuracy and efficiency. Below, we discuss two implementations: SMATS Traffic Solutions and Q-Free's traffic monitoring systems.

1.3.1 SMATS Traffic Solutions

SMATS Traffic Solutions is a company specializing in real-time traffic data collection using radar-based sensors [7]. Their systems provide highly accurate traffic flow analysis, detecting up to 256 vehicles simultaneously and classifying them into seven distinct vehicle types. The sensors are designed to be non-intrusive and require minimal infrastructure modification, making them suitable for both temporary and permanent installations.

SMATS employs IoT-enabled traffic monitoring devices that integrate Bluetooth and Wi-Fi sniffing technology to track vehicle movements and estimate travel time and congestion levels. This system allows traffic agencies to analyze historical and real-time traffic data for pattern recognition and congestion mitigation. Additionally, SMATS integrate with cloud-based analytics platforms, providing visual insights for optimizing traffic management strategies.

While SMATS provides highly precise traffic data, its reliance on specialized radar sensors and wireless sniffing devices means that its implementation requires dedicated equipment and professional calibration. This can lead to higher deployment costs and limitations in scalability compared to solutions that leverage existing infrastructure.

1.3.2 Q-Free Traffic Monitoring Systems

Q-Free offers a multi-technology approach to traffic monitoring, utilizing a combination of sensor-based and vision-based systems [8]. Their traffic data collection solutions integrate:

- In-road sensors: Piezo-polymer, -quartz, and strain gauge sensors embedded in roadways.
- Inductive loops: Permanently installed to detect vehicle presence and classification.
- Automatic Number Plate Recognition (ANPR): Capturing vehicle identification data.
- CCTV cameras: Providing real-time visual monitoring of traffic conditions.
- Height-width-length laser scanners: Enhancing vehicle classification accuracy.
- Diversion signs and Weigh-in-Motion (WIM) systems: Assisting in traffic flow adjustments and freight monitoring.

This comprehensive system allows precise monitoring of vehicle movement, classification, and speed, enabling authorities to optimize traffic signals, enforce traffic regulations, and conduct congestion analysis. Unlike purely radar-based systems, Q-Free's solution provides multiple layers of verification, ensuring robust data collection even in complex traffic environments. However, the cost and infrastructure requirements for deploying Q-Free's system are significant, as it relies on fixed sensors, cameras, and in-road installations. This makes the solution highly effective but less adaptable for areas where deploying dedicated infrastructure is impractical.

While SMATS and Q-Free provide highly accurate and reliable traffic monitoring solutions, they both require specialized hardware and significant infrastructure investment. In contrast, our project explores a deep learning-based computer vision solution that operates using ordinary traffic camera video feeds. By leveraging AI-powered vehicle detection and tracking algorithms, our approach aims to build a scalable, cost-effective alternative that can extract key traffic insights, such as vehicle count, average speed, and congestion level, without the need for specialized sensors. This method allows for flexible deployment, making it a viable solution for smart city initiatives that aim to enhance traffic management using existing camera infrastructure.

2 Methodology

2.1 Design

The design phase of the project included the following aspects:

2.1.1 System Architecture

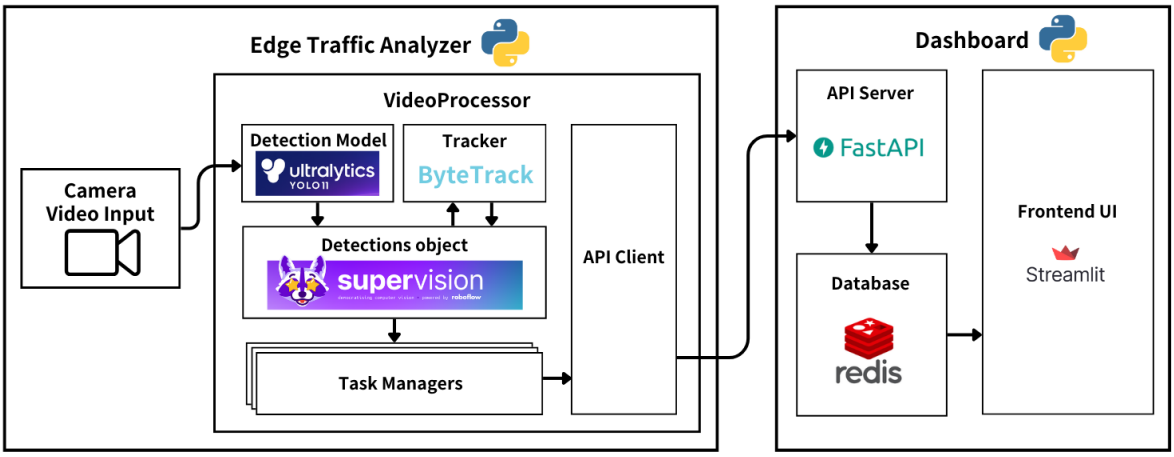


Figure 1: System Architecture

Our traffic data analytics system includes two main modules:

1 An Edge Traffic Analyzer for vehicle detection and tracking to extract traffic data locally.

2 A Dashboard for the visualization and real-time display of traffic information.

2.1.2 Edge Traffic Analyzer

The Edge Traffic Analyzer component is designed to process and analyze traffic conditions locally on an edge device. This system is comprised of several functionalities:

2.1.2.1 Vehicle Detection

We selected the YOLO11 model as our object detection framework for its proven speed and accuracy in real-time applications [3]. YOLO11 is originally pretrained on the COCO dataset, which contains a wide variety of objects but is too generic to capture the specific characteristics of vehicles in the Hong Kong environment. To enhance performance, we fine-tuned the YOLO11 model using a custom vehicle detection dataset tailored to the local traffic conditions of Hong Kong. Detailed information on the fine-tuning process and dataset composition is provided in the Implementation section. This fine-tuning process, involving more specialized data and focused on diverse vehicle types under the Hong Kong environment, ensures improved accuracy in detecting vehicles such as cars, taxis, buses, and trucks.

After fine-tuning, the model processes each input video frame through its convolutional network and outputs bounding boxes that delineate the location of each detected vehicle. These bounding boxes represent the vehicles' positions within the frame. A visual example of the bounding boxes is shown in Figure 2.

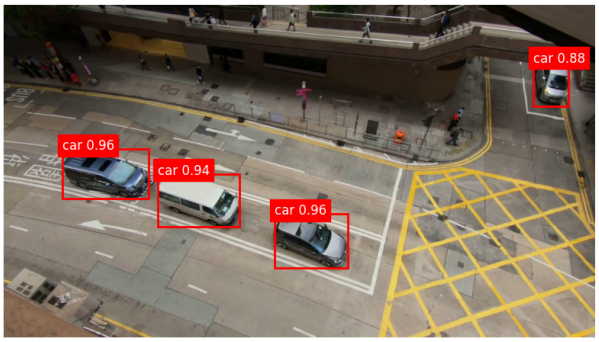


Figure 2: Example of detected bounding boxes for vehicles in a traffic scene.

2.1.2.2 Vehicle Tracking

The ByteTrack algorithm [4] was used for vehicle tracking, following detected vehicles across multiple frames while preserving their unique identity, ensuring that the system does not redundantly detect the same vehicles.

ByteTrack operates by first predicting the possible new locations of previously detected vehicles using a Kalman filter [9], which estimates the future position of each tracked object based on its current state. These predictions are then matched with the detected bounding boxes from the latest frame provided by the vehicle detection model.

ByteTrack processes both high-confidence and low-confidence detections, where the confidence value indicates the probability that the detected bounding boxes from the latest frame match the same identity as the predictions from previous frames. It assigns detected objects to predicted positions based on Intersection-over-Union (IoU) scores and appearance features. This approach helps retain vehicles even if their detection confidence temporarily drops, allowing for more consistent tracking across frames.

2.1.2.3 System Configuration

To generalize our Edge Traffic Analyzer for deployment at different locations, the system supports a variety of configurable parameters defined in a configuration file. These include the camera location name, GPS coordinates (longitude and latitude), and a list of zones to be monitored. Each zone entry contains a zone name, its coordinates, width, length, speed limit, capacity, and an associated list of tasks to be performed in that zone.

A zone represents a user-defined area within the camera frame where traffic analysis is to be conducted. The coordinates specify a polygon that outlines the region of interest using image pixel values. Each zone's width and length describe the approximate real-world dimensions of the zone and are essential for accurate perspective transformation during speed calculation. These values should ideally be measured on-site or estimated using tools such as the Google Maps distance measurement tool, ensuring they reflect the true physical scale of the observed area. The capacity indicates the estimated maximum number of vehicles that can fit in the zone, which, together with the speed limit, is used to assess congestion levels.

Each zone also includes a list of task configurations. These tasks define what traffic analysis operations should be performed in the zone. For example, the traffic metrics extraction task enables extraction of vehicle counts, average speed, flow rate, and congestion levels, while the illegal parking detection task monitors for illegal parking behaviors based on idle time, speed threshold, and time constraints. Each task can be customized independently within the configuration, allowing for highly flexible and modular analysis across different zones.

A typical setup for traffic metrics extraction involves defining one zone for each direction of a street at a given location, while the setup for illegal parking detections involves defining one zone for each area of a street with parking restrictions. For instance, in Figure 3, two zones are configured for Salisbury Road near K11 MUSEA, corresponding to each direction of traffic flow.

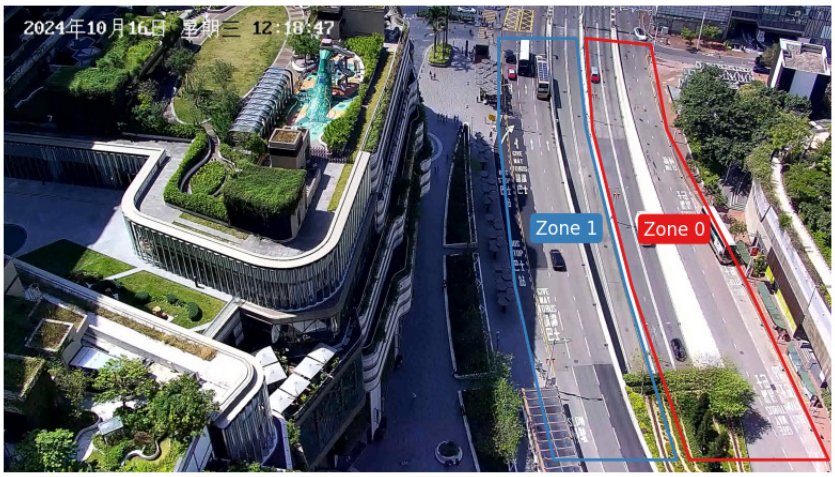


Figure 3: Example of configured traffic zones at Salisbury Road.

However, the zone definition is highly flexible and can be adapted to the specific needs of the deployment. More granular setups can define individual zones for each lane, while broader configurations can encompass an entire street segment without distinguishing directions. Similarly, task assignments can vary per zone, allowing certain zones to focus on traffic flow analysis while others focus on enforcement-related tasks such as illegal parking detection.

2.1.2.4 Traffic Metrics Extraction

- **Average Vehicle Speed:** Average vehicle speed offers insight into how freely vehicles are moving within a zone. Monitoring average speeds over time enables the detection of changes in traffic conditions, which could be useful in various traffic optimization efforts.

  The average vehicle speed for each zone is calculated by averaging the individual speeds of all unique vehicles in the zone. Individual vehicle speed is calculated by measuring the displacement of the vehicle's coordinates across consecutive frames and dividing the distance by the time between frames.

  However, this approach using raw image coordinates to calculate the speed of a three-dimensional space causes perspective distortions, where pixel distances do not accurately represent real-world distances depending on the camera angle. To address perspective distortion, a perspective transformation was applied to convert pixel-based coordinates into real-world measurements, ensuring accurate speed calculations. More details will be explained in the Implementation section.

- **Vehicle Counts:** Vehicle count is valuable for understanding instantaneous traffic density, which can indicate the current congestion level in different areas. Accurate vehicle counts help illustrate how traffic conditions fluctuate throughout the day and establish real-time awareness of high-traffic zones.

Vehicle counts are calculated by counting the unique vehicle IDs in each zone at the instant of the current frame. ByteTrack's ability to assign and maintain unique IDs for vehicles ensures accurate counting without duplication.

- **Traffic Flow Rate:** Traffic flow rate provides a broader perspective on traffic movement trends over time, helping to distinguish between consistently congested areas and temporary slowdowns. Understanding flow rates allows for a better grasp of how traffic propagates and whether certain routes experience consistent bottlenecks.

  The traffic flow rates are calculated by accumulating the number of unique vehicles that drove inside each zone within a set time interval, typically one hour. The flow rates are calculated separately for each zone.

- **Congestion Level:** Congestion levels provide a high-level overview of traffic conditions in different zones. By tracking congestion dynamically, city planners and traffic management systems can gain data-driven insights into how congestion evolves throughout the day.

  Congestion level is determined using a two-stage threshold-based method with two indexes: Vehicle Count Index (VCI) and Moving Average Speed Index (MASI). The determination rule is illustrated in Figure 4.

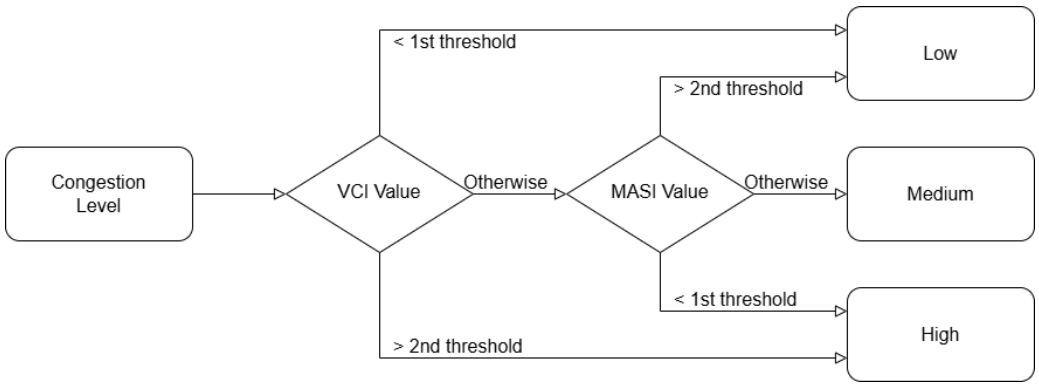


Figure 4: Illustration of congestion level determination.

  - **Vehicle Count Index (VCI):** VCI is defined as the rolling average vehicle count divided by the maximum vehicle capacity of the zone.
  - **Moving Average Speed Index (MASI):** MASI is defined as the moving average of the average speed of the zone divided by the speed limit.

- **Vehicle Type Distribution:** The distribution of vehicle types provides insight into the composition of traffic in each zone. By analyzing the proportions of different vehicle classes, such as buses, minibuses, taxis, trucks, private cars, and motorcycles, traffic administrators can better understand usage patterns on the street. For instance, a high proportion of heavy vehicles may indicate potential for slower traffic flow or increased wear on road infrastructure, while a dominance of public transport vehicles like buses and minibuses may reflect commuter density. Monitoring this distribution over time also helps identify shifts in traffic behavior, such as increased commercial activity or changing transportation preferences.

2.1.2.5 Illegal Parking Detection

This task identifies vehicles that are stationary for prolonged periods and potentially obstructing traffic flow. The objective is to monitor vehicle activity within each zone and detect violations based on configurable criteria, such as minimum idle time, low vehicle speed, time-of-day restrictions, and types of vehicle restrictions. When such criteria are met, the system flags the behavior as a potential illegal parking incident.

2.1.3 Dashboard Development

To provide traffic department administrators with easy access to real-time traffic data and insights through a user-friendly interface, the dashboard transforms complex information into actionable insights for further analysis. The dashboard consists of Backend and Frontend components.

2.1.3.1 Dashboard Configuration

To support flexible expansion, location-based and task-based filtering, we designed a JSON-based configuration file that maps each location to its available tasks. For example, `K11` is linked to tasks like `Traffic Info`, while `KwunTong` corresponds to `Illegal Parking`. This configuration is dynamically loaded into the Streamlit frontend landing page using a cached function. During runtime, the system initializes two variables, `selected_location` and `selected_task`, in Streamlit's session state based on user selections from dropdown menus. The dashboard filters available tasks per location and dynamically loads the corresponding visualization module. This modular and data-driven approach allows for easy updates and scalability without altering the core application logic.

2.1.3.2 Backend Database Storage

We utilize Redis (Remote Dictionary Server) [5] to store traffic information extracted from Edge Traffic Analyzers. Redis is a distributed, in-memory key-value database widely recognized for its exceptional speed and low-latency read and write operations, making it an ideal choice for our real-time application.

Upon receiving traffic data from API request, it is ingested into the Redis database and structured as key-value pairs. Each key serves as a unique identifier for a specific traffic metrics entry, constructed from a combination of zone ID, location, timestamp, etc. The associated value contains the relevant traffic metrics, such as vehicle count, average speed, congestion level, and snapshot images.

While Redis supports various data structures for flexible and efficient data modeling, we specifically use JSON through the Redis Modules API to effectively store our data, while for sending and storing image snapshots, we utilize the raw byte format. This setup enables rapid and efficient data retrieval by the dashboard frontend component for real-time display and enhance system integration.

2.1.3.3 Frontend Visualization

To present complex traffic data in an intuitive and accessible format, we adopt Streamlit [6], an open-source Python framework designed for building interactive web applications. Its simplicity and flexibility enable rapid development and visualization, making it an ideal tool for showcasing real-time analytics. By integrating Pandas and Plotly with Streamlit, the dashboard

supports various types of statistical and geospatial visualizations. This combination provides users with a comprehensive overview of the traffic environment and promotes effective, data-driven decision-making.

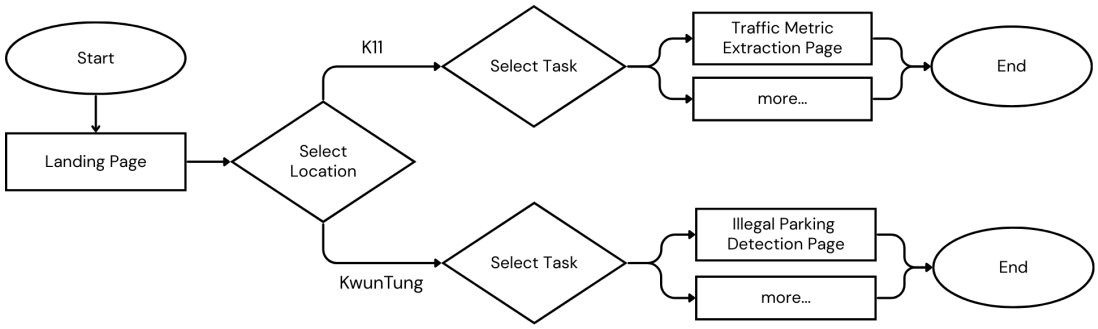


Figure 5: User Flow of Dashboard Interaction

Landing Page The system features a landing page that allows users to select a location and corresponding task, such as Traffic Metrics Extraction or Illegal Parking Detection. Based on the user's selection, the app dynamically navigates to the appropriate task page and filters the data according to the location selected.

Traffic Metrics Extraction Page On the Traffic Metrics Extraction page, users can explore key metrics including average speed, vehicle counts, vehicle type distributions, and congestion level indicators. These metrics are presented through interactive line graphs, histograms, and summary statistics, enabling both real-time monitoring and historical trend analysis.

Illegal Parking Detection Page On the Illegal Parking Detection page, the interface displays the selected location and real-time image snapshots of newly detected violations. This view is continuously refreshed to ensure up-to-date monitoring.

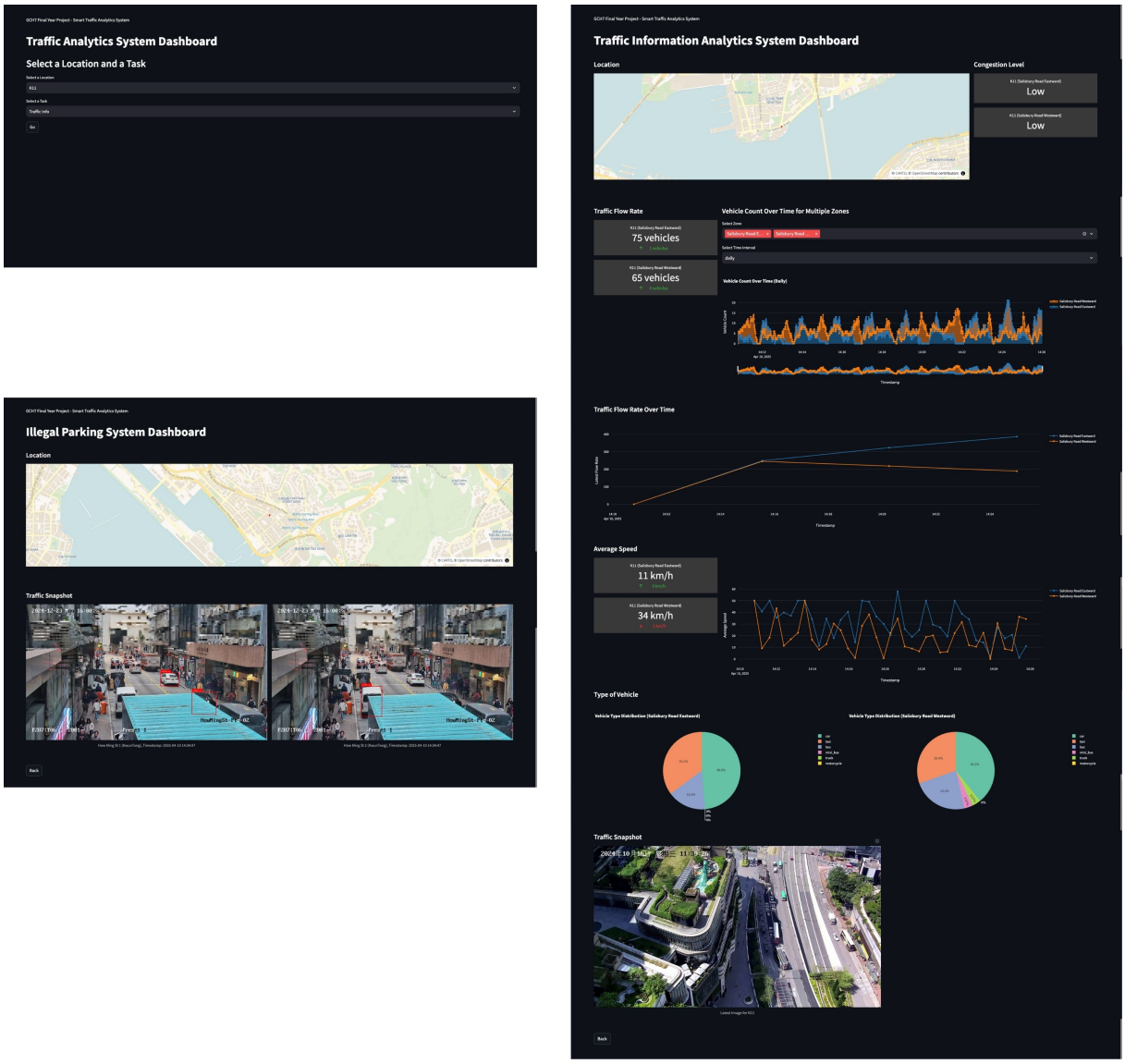


Figure 6: Dashboard Interface for Real-Time Visualization

2.1.4 System Integration

To allow the Edge Traffic Analyzers to transmit extracted traffic information to the Dashboard, the data is sent in the form of API POST requests to the API endpoint of the dashboard hosted by FastAPI [10] with Uvicorn [11]. Since different metrics have varying update frequencies, we designed separate endpoints for each type of traffic data transmission. More details will be explained in the Implementation section.

2.2 Implementation

This implementation section explains in more detail each part of the system and the underlying mechanisms.

2.2.1 Edge Traffic Analyzer

Figure 7 illustrates the data pipeline in the Edge Traffic Analyzer.

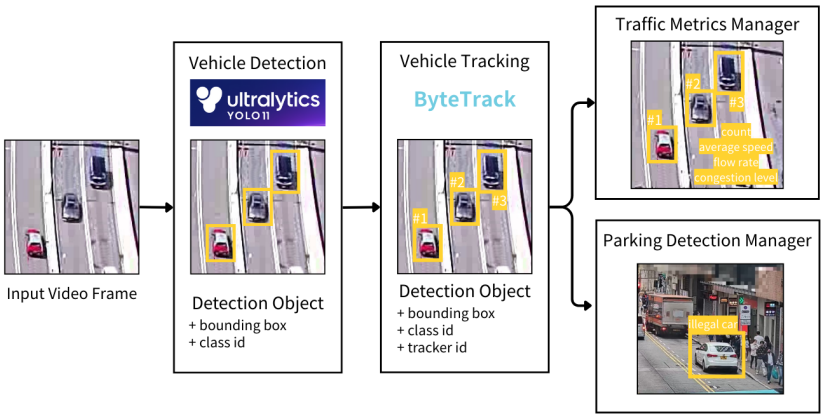


Figure 7: Edge Traffic Analyzer Data Pipeline

2.2.1.1 Real-time Edge Processing

The system is designed to operate directly on edge devices deployed at the target location, enabling real-time processing of live video streams captured by local traffic cameras. By handling computation on-site, the system minimizes latency, reduces the need for large-scale video transmission, and supports immediate extraction of traffic data.

For demonstration purposes, different camera locations are used to showcase distinct tasks. The traffic metrics extraction task is demonstrated using video footage from the K11 site at Salisbury Road, Tsim Sha Tsui. Meanwhile, the illegal parking detection task is showcased using video from How Ming Street in Kwun Tong, a location known for frequent roadside parking activities.

2.2.1.2 YOLO11 Fine-tuning

To fine-tune the pre-trained YOLO11 model for vehicle detection tasks, we constructed a dataset comprising both publicly available and custom-labeled data. The majority of the data was sourced from the Roboflow Universe [12], specifically selecting datasets that feature vehicle images captured in Hong Kong environments. These images include a broad variety of real-world traffic scenes, encompassing different lighting conditions, weather scenarios, and vehicle types common in the region. Figure 8 shows a sample of the dataset selected from Roboflow.



Figure 8: Sample vehicle dataset from Roboflow Universe

To further improve the model's domain adaptation, we augmented the dataset with a small but impactful set of self-labeled images, curated from video frames captured using our actual deployment cameras (e.g., K11 MUSEA), and other video sources with similar camera angles. Labeling was done using Label Studio [13]. Figure 9 illustrates the labeling interface and annotation process.

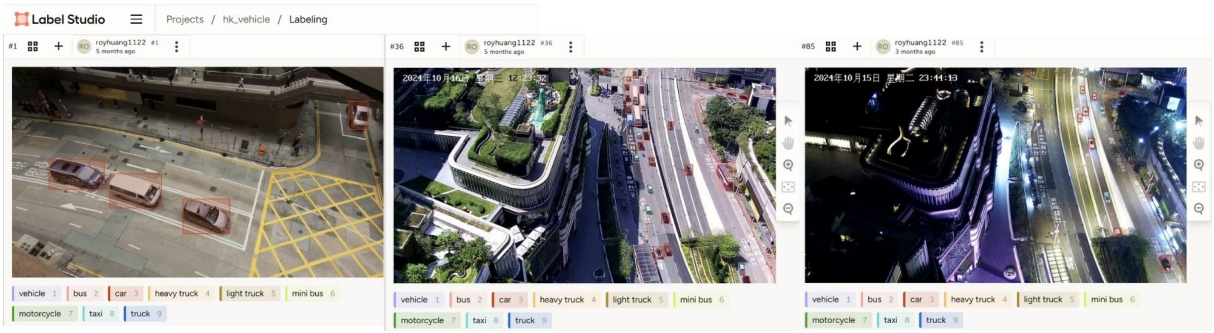


Figure 9: Label Studio interface used for manual annotation

The self-labeled data plays a critical role in enhancing the fine-tuning process. Since these images are drawn directly from the system's operating environment, they better represent deployment specific challenges such as camera angles, resolution, background clutter, and region-specific vehicle appearances (e.g., minibuses, taxis). As a result, they provide highly relevant training examples that help the model adapt to visual patterns it will encounter in real-time inference.

Table 1 summarizes the dataset, with columns representing the training and validation splits, along with their totals and percentages. The rows indicate the class labels, and an additional row at the bottom details the number of self-labeled samples.

Class	Train	Valid	Total	Percentage (%)
Bus	300	82	382	5.58
Car	2849	679	3528	51.50
Mini Bus	429	117	546	7.97
Motorcycle	299	113	412	6.01
Taxi	742	172	914	13.34
Truck	877	191	1068	15.59
Total	5496	1354	6850	100.00
Percentage (%)	80.23	19.77	100.00	
Self-labeled	160	41	201	

Table 1: Dataset summary for YOLO11 fine-tuning

This robust and diverse dataset forms the foundation of our fine-tuning pipeline. By combining publicly sourced and system-specific samples, we ensure that the resulting YOLO11 model is both generalizable and highly adapted to the unique traffic scenarios of Hong Kong, leading to improved detection accuracy and greater reliability in real-world deployment.

2.2.1.3 Vehicle Detection and Tracking

Our system utilizes the Ultralytics [3] Python library to run the fine-tuned YOLO11 model inference on the traffic video frames, and the Supervision [14] Python library's ByteTrack mod-

ule to perform object tracking. Supervision was used to standardize the output format of both YOLO11 and ByteTrack into the `Detections` class [15], ensuring consistency in how detected objects are represented and providing a uniform API for integration with Supervision trackers, annotators, and tools.

The `Detections` class consists of the following attributes:

- **Bounding box coordinates:** An array of shape $(n, 4)$ containing the bounding box coordinates in the format $[x1, y1, x2, y2]$.
- **Class ID:** An array of shape $(n,)$ containing the class IDs of the detections.
- **Tracker ID:** An array of shape $(n,)$ containing the tracker IDs of the detections.

During each video frame, the YOLO11 model runs inference on the frame and generates a `Detections` object, which contains the bounding box coordinates and class IDs of detected vehicles, while the tracker IDs are not initialized. The `Detections` object is then fed into the ByteTracker module to calculate and update their corresponding tracker IDs.

2.2.1.4 Task Managers

To support modular and scalable processing of traffic data, the system adopts a task-based architecture where each analysis task is handled by a dedicated task manager. These task managers are responsible for managing their own internal data, processing logic, and API communication routines. All task managers inherit from a common `BaseTaskManager` interface, ensuring consistency in how tasks are initialized, updated, and executed across zones.

Each zone can be configured with one or more task managers, such as traffic metrics extraction or illegal parking detection, through the configuration file. For every new video frame, detections that fall within a zone and their data (e.g., speed data) are passed to the corresponding list of task managers, where each manager processes the detections independently according to its own task-specific logic.

This design separates concerns and enables clean, modular development. It also allows for easy extensibility, as new traffic-related tasks can be added by simply implementing a new task manager that follows the shared interface.

The details of individual task managers are described in sections 2.2.1.6 and 2.2.1.7.

2.2.1.5 Speed Calculation

Figure 10 illustrates the data pipeline for the speed calculation and the following average speed calculation.

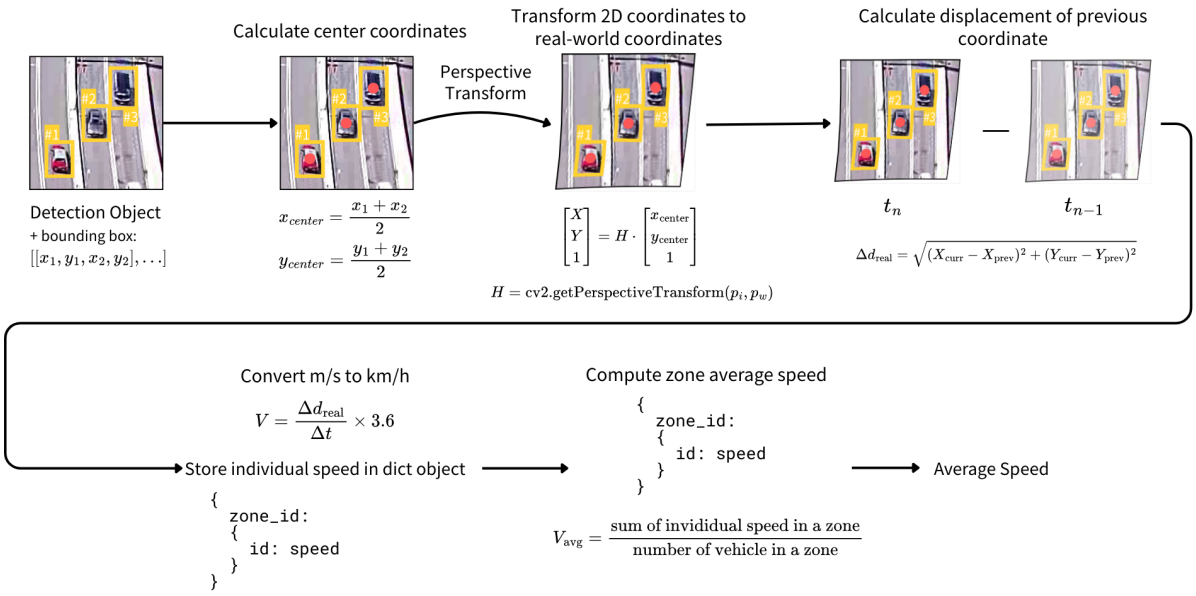


Figure 10: Speed Calculation Data Pipeline

Speed calculation is handled as a shared utility outside of individual task managers, allowing consistent and centralized computation of vehicle speeds across different tasks. This design supports reuse of speed information by multiple task managers. The calculated speed values are stored in a shared data dictionary, indexed by zone and tracker ID, and passed to task managers during each frame update.

The estimation of vehicle speed is performed by tracking the movement of detected vehicles across consecutive frames. Each detected vehicle is represented by the bounding box coordinates $[x_1, y_1, x_2, y_2]$ within the `Detections` object. To determine vehicle speed, the system first computes the center coordinates of the bounding box as:

$$x_{\text{center}} = \frac{x_1 + x_2}{2}, \quad y_{\text{center}} = \frac{y_1 + y_2}{2}$$

The system applies a homography transformation using OpenCV's `getPerspectiveTransform()` [16] to map pixel coordinates from the image plane to real-world coordinates. The transformation requires four image points and their corresponding real-world points, defining a mapping between the two coordinate systems. The system automatically extracts the four corner points from the polygon defining each zone (provided in the configuration file) to serve as the four image points, while the corresponding real-world points are defined as the rectangular corners determined by the specified width and length of the zone (approximated by on-site measuring or estimattion using tools such as the Google Maps distance measurement tool). Figure 11 illustrates the transformation between pixel coordinates and real-world coordinates.

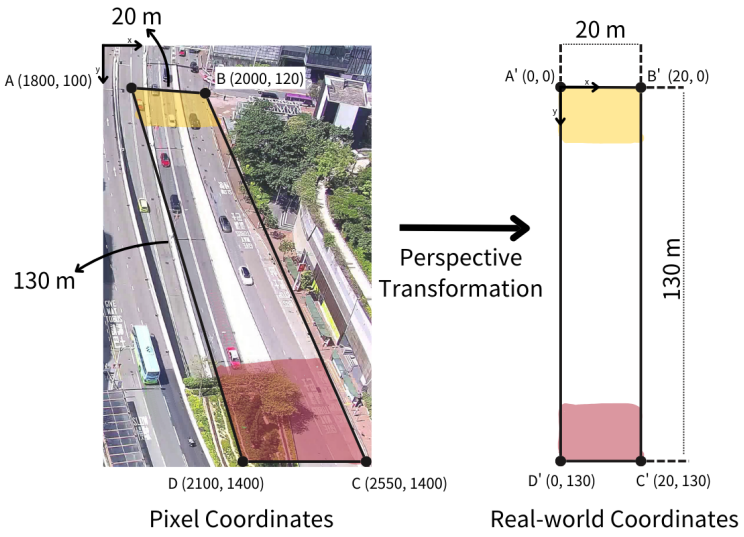


Figure 11: Transformation between pixel and real-world coordinates.

Using the transformation matrix H, obtained via:

$$H = \text{cv2.getPerspectiveTransform}(p_i, p_w)$$

where p_i represents four points in the image and p_w represents their real-world equivalents, any detected vehicle's center position can be transformed into real-world coordinates:

$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \cdot \begin{bmatrix} x_{\text{center}} \\ y_{\text{center}} \\ 1 \end{bmatrix}$$

where (X, Y) represents the vehicle's position in real-world space (measured in meters).

For every detected vehicle, the system retrieves the previous frame's center coordinates stored in the `Detections Manager`.

After applying this transformation to both the previous frame's and current frame's positions, the real-world displacement Δd_{real} is calculated using the Euclidean distance formula:

$$\Delta d_{\text{real}} = \sqrt{(X_{\text{curr}} - X_{\text{prev}})^2 + (Y_{\text{curr}} - Y_{\text{prev}})^2}$$

Once the real-world displacement is determined, the vehicle's speed is calculated using the formula:

$$V = \frac{\Delta d_{\text{real}}}{\Delta t} \times 3.6$$

where Δt represents the time elapsed between frames, and the factor 3.6 converts meters per second into kilometers per hour. The individual speeds are then stored in the shared data structure to be passed to the various task managers.

2.2.1.6 Traffic Metrics Calculation

The extraction of traffic information is managed by the `TrafficMetricsManager`, which processes zone-specific detections for each frame. These detections are obtained after vehicle detection and tracking, and include associated tracker IDs and vehicle class information. Using

this data, the manager computes key traffic metrics, including average vehicle speed, vehicle count, traffic flow rate, and congestion level.

- **Average Vehicle Speed Calculation**

  Vehicle speeds are retrieved from the shared speed data structure, which stores precomputed values indexed by zone and tracker ID. Each frame, the speeds of all vehicles currently within the zone are averaged to produce the zone's average speed:

$$V_{\text{avg}} = \frac{\sum_{i=1}^{N} V_i}{N}$$

  where N is the total number of vehicles detected in the zone during that frame.

  The system maintains a rolling window of speed values for each zone to compute a Moving Average Speed (MAS), which serves as a foundation for congestion analysis.

- **Vehicle Count Calculation**

  Figure 12 illustrates the data pipeline for the vehicle count calculation.

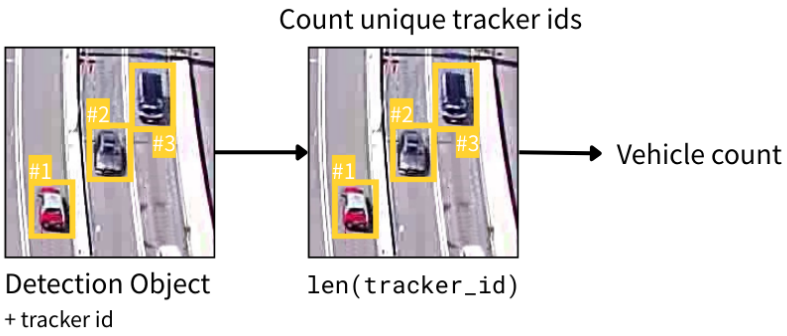


Figure 12: Vehicle Count Data Pipeline

The vehicle count is determined by tracking the unique vehicle tracker IDs assigned by ByteTrack within each defined traffic zone. To efficiently determine whether a vehicle is inside a zone, the system utilizes Supervision's `trigger()` function [17], which evaluates whether a detected object is within a predefined polygonal area. This enables efficient filtering of detections, ensuring that only vehicles inside the designated traffic monitoring zones are considered.

For each frame, the system extracts the tracker IDs of all detected vehicles and checks whether they fall within the respective traffic zones using the `trigger()` function. The number of vehicles in a given zone is then computed as the total count of unique tracker IDs present in that zone at that moment.

The system maintains a rolling window of vehicle counts, storing recent values and computing an average over time. This rolling count serves as a foundation for congestion analysis.

Figure 13 illustrates an example of vehicle count within real detection zones.

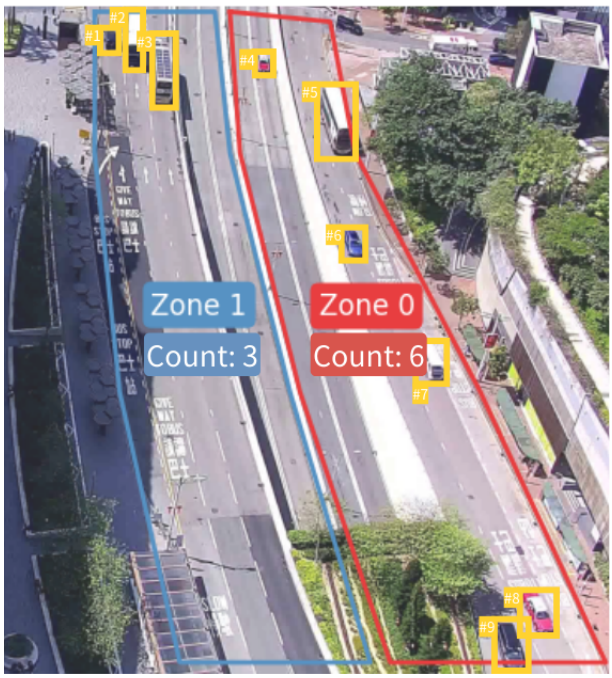


Figure 13: Vehicle Count Example

- **Traffic Flow Rate Calculation**

  Figure 14 illustrates the data pipeline for the traffic flow rate calculation.

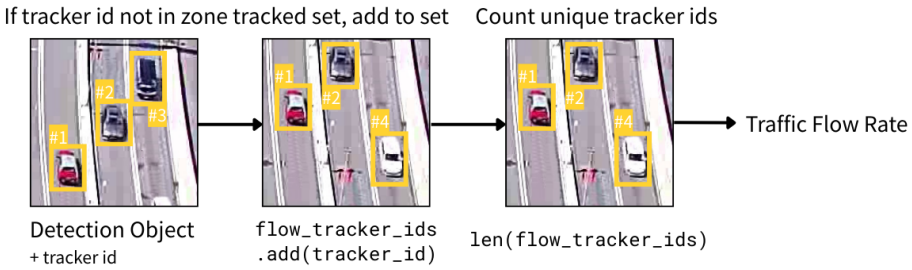


Figure 14: Traffic Flow Rate Data Pipeline

The traffic flow rate measures the number of vehicles that pass through a zone within a fixed time interval, typically one hour. Unlike the instantaneous vehicle count, which reflects the number of vehicles present in a zone at a specific moment, the flow rate captures the number of new unique vehicles entering the zone over time.

To implement this, the system maintains a set of active tracker IDs for each zone, keeping track of which vehicles have already been counted during the current time interval. When a new vehicle is detected in a zone, its tracker ID is checked against the existing set. If the tracker ID has not been recorded for the current interval, it is added to the set, and the flow rate for that zone is incremented accordingly.

At the end of each time interval, the total number of unique vehicles counted in the zone is stored as the latest traffic flow rate for that period, and the set of active tracker IDs resets, allowing the system to capture fresh vehicle movements in subsequent intervals. Figure 15 illustrates an example of traffic flow rate within real detection zones.

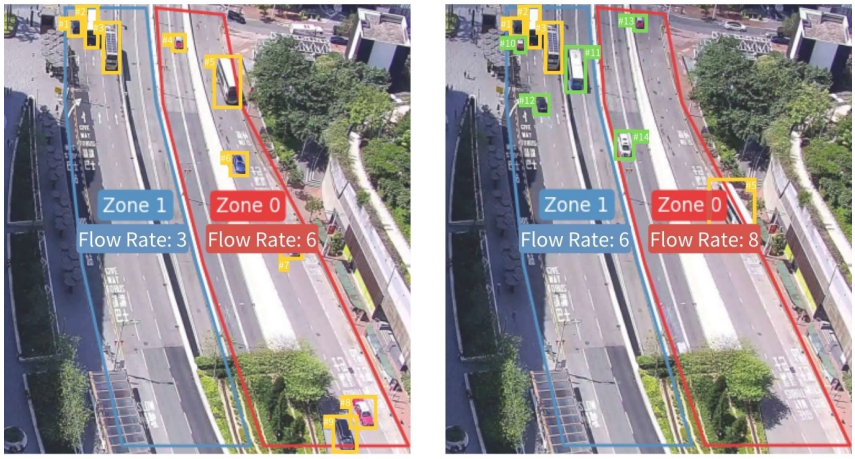


Figure 15: Traffic Flow Rate Example

- **Congestion Level Calculation**

  Figure 16 illustrates the data pipeline for the congestion level calculation.

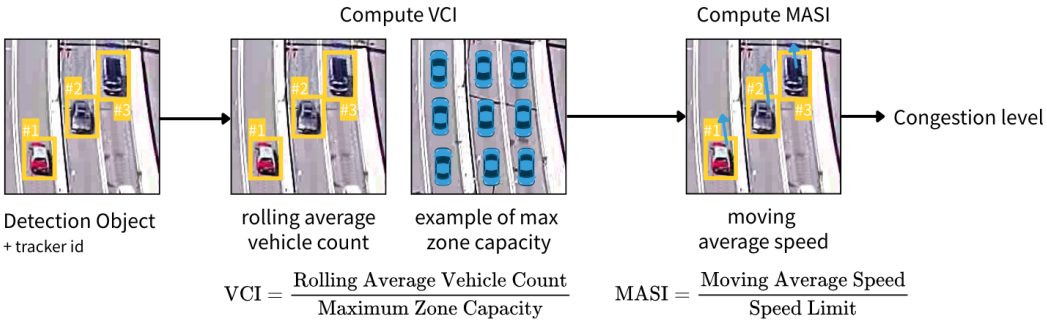


Figure 16: Congestion Level Data Pipeline

Congestion levels are determined using a two-metric threshold-based classification system, incorporating both the Vehicle Count Index (VCI) and the Moving Average Speed Index (MASI).

1. **Vehicle Count Index (VCI):** This metric assesses how full a traffic zone is relative to its maximum capacity. It is defined as:

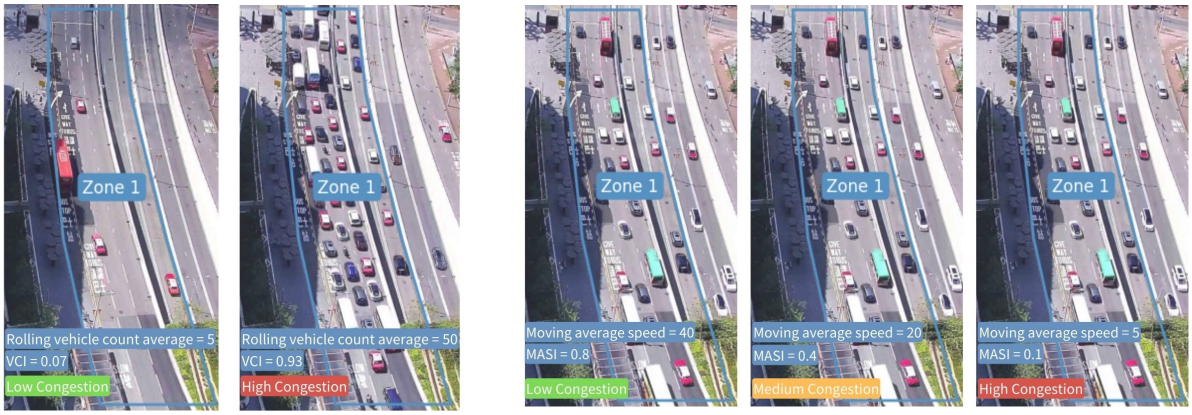
$$\text{VCI} = \frac{\text{Rolling Average Vehicle Count}}{\text{Maximum Zone Capacity}}$$

   The rolling average vehicle count, obtained from the previous count calculations, provides a stable representation of the number of vehicles present over time.

2. **Moving Average Speed Index (MASI):** This metric evaluates how slow vehicles are moving within the zone by comparing the MAS of vehicles against a predefined speed limit. It is computed as:

$$\text{MASI} = \frac{\text{Moving Average Speed}}{\text{Speed Limit}}$$

The congestion levels are then determined based on the definition in the design section 2.1.2.4. Figure 17 illustrates examples of congestion levels within a real detection zone: subfigure 17a shows VCI-based congestion (maximum vehicle capacity = 54, VCI thresholds = (0.6, 0.9)), and subfigure 17b shows MASI-based congestion (speed limit = 50, MASI thresholds = (0.2, 0.6)).



(a) VCI-based congestion (b) MASI-based congestion

Figure 17: Congestion Level Example

- **Vehicle Type Distribution**

  The classification of each detected vehicle is already available in the `Detections` object following the detection stage, with class IDs assigned to represent different vehicle types (e.g., 0 for bus, 1 for car, etc.). For every frame, the system checks whether a detected vehicle has a new tracker ID (i.e., not previously counted during the current interval). If so, it increments a counter in a dictionary that stores the number of vehicles per class ID. This dictionary accumulates the vehicle type counts over a fixed time interval, similar to how traffic flow rate is calculated. Periodically, the counts are packaged and sent to the dashboard via an API request, and at the end of each interval, the dictionary is reset for the next cycle. Figure 18 illustrates an example of vehicle type distribution within a real detection zone.

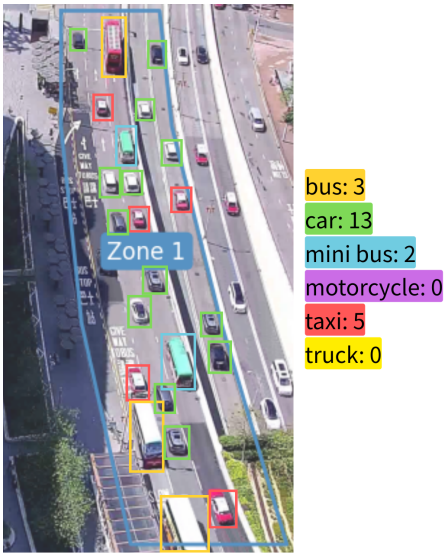


Figure 18: Vehicle Type Distribution Example

2.2.1.7 Illegal Parking Detection

Figure 19 illustrates the data pipeline for the illegal parking detection.

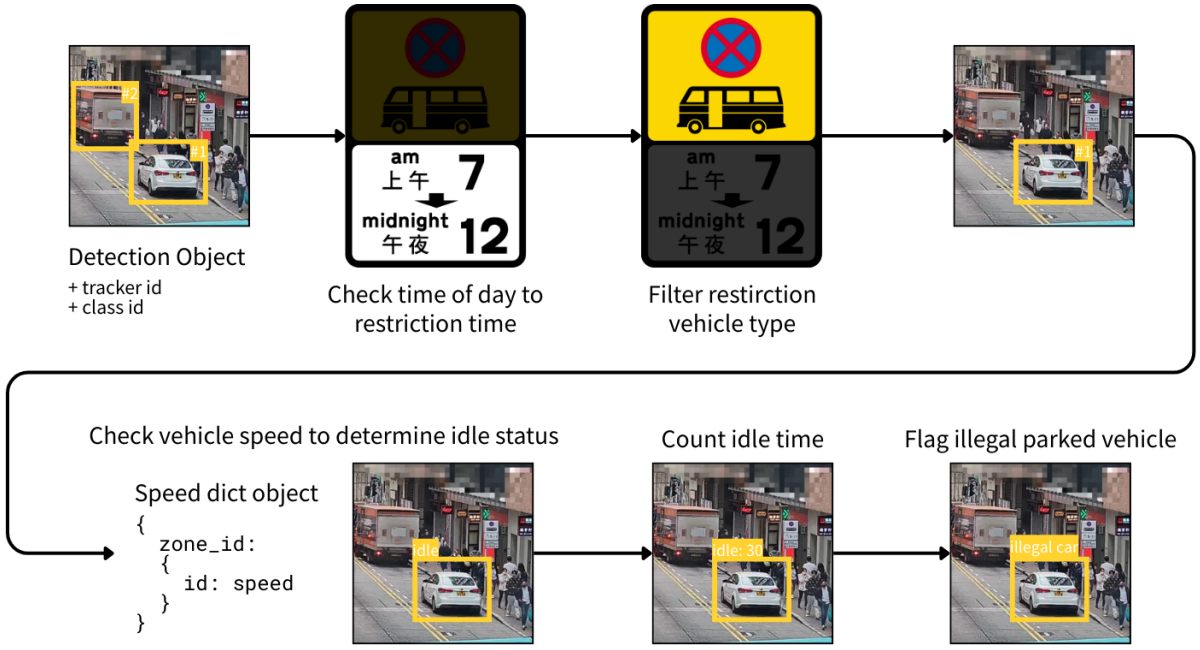


Figure 19: Illegal Parking Detection Data Pipeline

Illegal parking detection is handled by the `ParkingDetectionManager`, which monitors vehicles within each zone to identify cases of prolonged idling or unauthorized stopping. This task is configurable per zone and operates in parallel with other task managers.

For each frame, the ParkingDetectionManager receives zone-specific detections along with shared speed data. It evaluates each vehicle's speed and class to determine whether it meets the criteria for illegal parking. A vehicle is considered idle if its speed remains below a configurable threshold (e.g., 1 km/h) for longer than a predefined idle time (e.g., 30 seconds). The system tracks how long each vehicle remains idle using its tracker ID and frame count.

In alignment with real-world enforcement practices in Hong Kong, where illegal parking regulations often apply only to specific types of vehicles or during designated time periods, the system supports both time-based and vehicle-type-based restrictions. Each zone can be configured to only detect violations within a specified time window (e.g., 07:00 to 19:00), and optionally restrict monitoring to selected vehicle categories (e.g., private cars, taxis, or trucks). These constraints are defined in the configuration file on a per-zone basis, enabling tailored detection logic that reflects practical traffic enforcement policies.

When a vehicle satisfies the criteria for illegal parking, it is flagged, and the system generates an annotated snapshot and sends it to the dashboard API for administrative review. Figure 20 illustrates an example of illegal parking within a real detection zone.

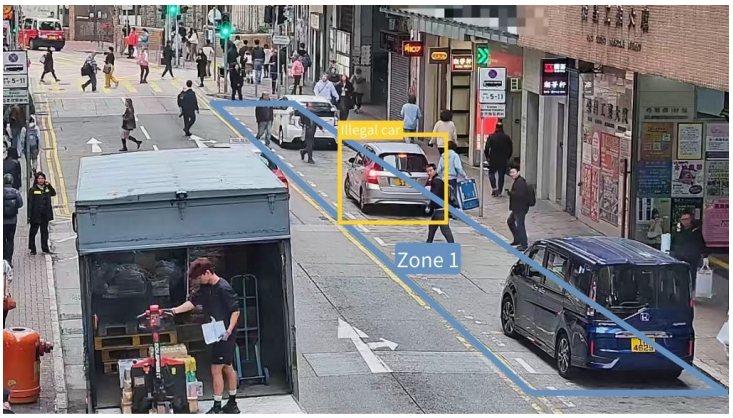


Figure 20: Illegal Parking Example

2.2.1.8 API Request for Sending Traffic Data

The system transmits traffic data to the dashboard server using an asynchronous, task-based API architecture. Each task manager, such as the Traffic Metrics Manager or Parking Detection Manager, is responsible for generating and enqueuing its own API calls based on task-specific metrics or events. These API requests are managed centrally by an `AsyncWorker`, which handles non-blocking HTTP POST requests using the `aiohttp` library.

This architecture enables concurrent, event-driven communication with the server while decoupling traffic analytics logic from networking concerns. Each task manager defines when and what type of data to send. For instance, Traffic Metrics Manager periodically sends data such as vehicle count, flow rate, average speed, and congestion levels, while Parking Detection Manager sends snapshots only when illegal parking activity is detected.

Common data types and their typical transmission intervals include:

1. `send_location`: Sends location metadata once at system initialization.
2. `send_count`: Sends vehicle count data every 1 second.
3. `send_avg_speed`: Sends average speed data every 30 seconds.
4. `send_flow_rate`: Sends traffic flow rate data every 1 second.
5. `send_latest_flow_rate`: Sends the latest flow rate every 3600 seconds (1 hour).
6. `send_congestion_level`: Sends congestion level data every 60 seconds.
7. `send_vehicle_types`: Sends the distribution of vehicle types every 60 seconds.
8. `send_traffic_info_frame`: Sends annotated video snapshots for general traffic insights every 120 seconds.
9. `send_illegal_parking_frame`: Sends snapshots of illegal parking events upon detection.

Each request is queued by its respective task manager and processed asynchronously by the `AsyncWorker`, ensuring that real-time analytics are not blocked by network operations. The API client constructs the payload for each request and handles retries or logging if a failure occurs. This modular, non-blocking design improves system responsiveness and supports varying transmission frequencies for different data types. Figure 21 summarizes the data endpoints and their corresponding asynchronous task functions.

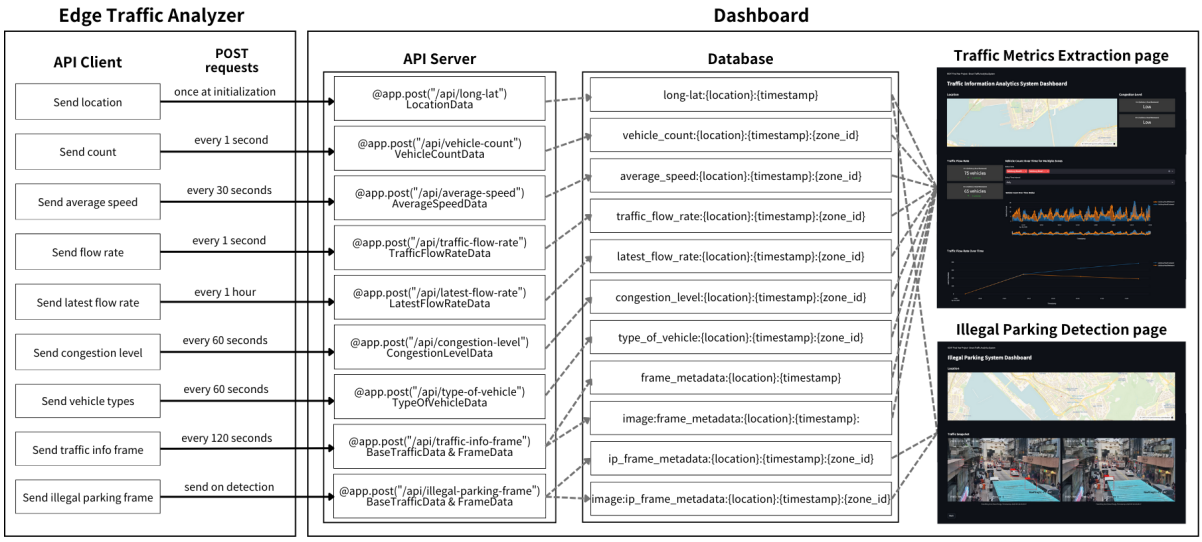


Figure 21: POST functions and corresponding API endpoints in the dashboard API server.

2.2.2 Dashboard Development

We utilize FastAPI [10] alongside Uvicorn [11] to establish a web server designed to receive calculated traffic information from the Edge Traffic Analyzer. Additionally, we employ Redis as the in-memory database set up in local for data storage and Streamlit [6] as the frontend library for data visualization.

The entire dashboard development process is initiated by receiving traffic information from the Edge Traffic Analyzer to Traffic Information Visualization, encompassing the following phases:

2.2.2.1 Data Structure Design and API Endpoint for Traffic Data Receivement

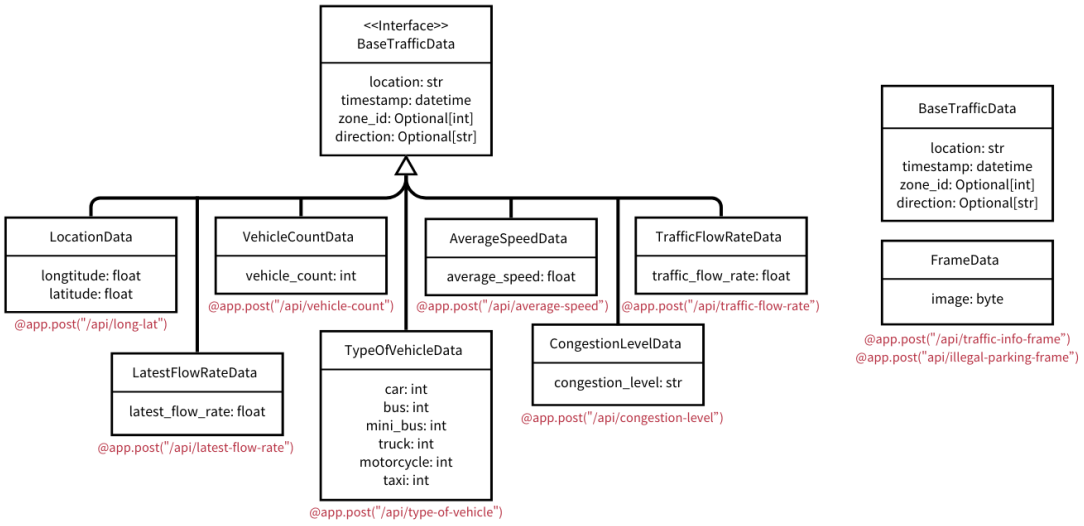


Figure 22: The Class Diagram of received traffic data structure design.

As shown in Figure 22, the foundational data structure is the `BaseTrafficData` class, defined using Pydantic. This class encompasses essential attributes required for the storage and visualization of traffic data:

- **Location**: Represents the name of the geographical location where traffic data is collected (e.g. K11).
- **Timestamp**: Records the precise time of data capture, facilitating time-series analysis.
- **Zone ID (Optional)**: Identifies the specific traffic zone, enabling targeted data aggregation and separation.
- **Direction (Optional)**: Indicates the defined zone direction of traffic flow (e.g. Salisbury Road Eastward).

This base class serves as an interface for seven specialized subclasses: `LocationData`, `VehicleCountData`, `AverageSpeedData`, `TrafficFlowRateData`, `LatestFlowRateData`, `CongestionLevelData`, and `TypeOfVehicleData`. This hierarchical structure facilitates the organization and management of diverse traffic data types, enhancing subsequent storage mechanisms. For example:

- `LocationData` extends the base class to include longitude and latitude, enabling map-based visualization of the location.
- `TypeOfVehicleData` extends the base class by adding attributes for counts of different vehicle types.
- The remaining subclasses (`VehicleCountData`, `AverageSpeedData`, `TrafficFlowRateData`, `LatestFlowRateData`, `CongestionLevelData`) extend the base class to include additional fields specific to their data type.

Each subclass receives JSON payloads from the Edge Traffic Analyzer's POST requests to its own endpoint, as depicted in Figures 21 and 22.

Since snapshot images are transmitted as raw byte data (not JSON) from the Edge Traffic Analyzer, we introduce a dedicated `FrameData` class. It handles the receipt of image bytes while sending associated metadata, including `timestamp` and `location` via `BaseTrafficData` for later retrieval and querying.

2.2.2.2 Redis Database Development

Upon receiving data from the Edge Traffic Analyzer via API calls, the ingestion process into the local Redis database involves several key steps:

1. **Data Modeling to design storage key-value pair**: The incoming data is structured to create a key for Redis storage. The `key` is constructed using a combination of parameters, including the location, timestamp, if applicable, the zone ID, along with the specific traffic information. The associated `value` contains the relevant traffic data formatted as JSON. This structured approach enables efficient querying and retrieval of traffic information based on specific criteria. For example:
    - `LocationData`: The key format is `long-lat:{location}:{timestamp}`, since the location is initialized only once and is location-dependent, making the zone ID unnecessary.
    - `FrameData`:

  - Metadata key for Traffic Extraction Page:
    `frame_metadata:{location}:{timestamp}`
  - Metadata key for Illegal Parking Detection Page:
    `ip_frame_metadata:{location}:{timestamp}:{zone_id}`
  - Image bytes key for Traffic Extraction:
    `image:frame_metadata:{location}:{timestamp}`
  - Image bytes key for Illegal Parking:
    `image:ip_frame_metadata:{location}:{timestamp}:{zone_id}`

  For the Traffic Extraction task, the camera settings capture all zones, resulting in a single snapshot without including the `zone_id` in the key design. In contrast, the Illegal Parking Detection system captures each zone individually, which results in separate zone identifiers in the key design.

- Other subclasses (e.g. VehicleCountData, AverageSpeedData): The key includes location, timestamp, zone ID, and specific info, e.g.
  `average_speed:{location}:{timestamp}:{zone_id}`.

The details of the storage keys for all eight traffic data types are illustrated in Figure 21.

2. **Data Storage**: Finally, the structured data is stored in Redis using SET commands: `redis_client.set(key, value)`, which sets the `key` alongside its corresponding `value` (e.g. the extracted traffic data or snapshot in byte format). Leveraging Redis as an in-memory database allows for rapid access to stored data, enhancing the performance and responsiveness of the real-time traffic analytics system.

2.2.2.3 Information Fetch from Database to Dashboard

The dashboard's frontend component retrieves traffic data from the Redis database using a Redis API `get` call. The data fetching process primarily follows these steps:

- **Key Retrieval and Filtering**: The function queries the Redis database using a wildcard pattern to retrieve all keys associated with the specified `data_type` identifier (e.g. `vehicle_count`, `average_speed`, etc.) and the user-specified `location`. During design, we positioned the `data_type` identifier at the start of the key and the `location` identifier second for faster querying.

- **Data Processing Loop**: For each retrieved key, the function fetches the corresponding value from Redis and parses it from JSON format into a Python dictionary.

- **Data Structuring**: Valid information is appended to a dictionary, sorted by timestamp to facilitate time-series visualization, and stored in a Pandas DataFrame.

Specifically, for traffic snapshots, we first use the above methodology to retrieve the metadata and identify the latest frame by timestamp. Then, we utilize this metadata (i.e. `location`, `timestamp`, and optionally `zone_id`) to construct the key
`image:frame_metadata:{location}:{timestamp}` or
`image:ip_frame_metadata:{location}:{timestamp}:{zone_id}` needed to fetch the corresponding byte image from Redis.

2.2.2.4 Streamlit Traffic Information Visualization

The final stage of implementing our dashboard involves designing a user interface that effectively visualizes the calculated and received traffic data.

We utilize Streamlit [6] Python framework for visualization, benefiting from its ease of use, deployment simplicity, and integration with Python libraries. This allows us to easily use Pandas [18] for DataFrame processing. Additionally, we employ Plotly Express [19] for its diverse graphing capabilities and customizable visual API.

Users first see a landing page with two `selectbox` widgets to choose the `location` and `task`. When a location is selected, the app reads the JSON configuration to filter available tasks for that location. The selections are stored in `st.session_state.selected_location` and `st.session_state.selected_task`, preserving state when navigating to the corresponding task page and filtering data accordingly.

On the Traffic Metrics Visualization Page, the system visualizes eight key traffic metrics as follows:

- **Location Data**: Longitude and latitude coordinates are plotted on a map using Plotly Express's `scatter_map`, displaying an OpenStreetMap view.
- **Congestion Level Data**: This categorical metric is shown with Streamlit's built-in components for easy interpretation of congestion levels.
- **Traffic Flow Rate Data**: Presented as a real-time numeric display using Streamlit.
- **Vehicle Count Data**: Displayed as a line chart via Plotly Express's `line` function to illustrate trends over time. To enhance interactivity, we implemented zone- and time-based filtering: a multi-select widget lets users choose traffic zones dynamically based on available zone mappings, and a selectbox widget lets users select time granularity (yearly, monthly, weekly, daily).
- **Latest Flow Rate Data**: Another line chart (Plotly Express) provides quick insights into hourly traffic conditions.
- **Average Speed Data**: Visualized as both a numeric metric (Streamlit) and a line chart (Plotly Express) to show current and historical speed trends.
- **Type of Vehicle Data**: A pie chart (Plotly Express's `pie`) categorizes and displays the distribution of vehicle types.
- **Frame Data**: Real-time traffic snapshots are shown using Streamlit's image component.

On the Illegal Parking Visualization Page, the system shows two key traffic metrics:

- **Location Data**: As above, plotted on a map with `scatter_map`.
- **Frame Data**: Snapshots of illegal parking in each zone are displayed via Streamlit's image component.

To enhance user experience, each task page refreshes every one second, ensuring the dashboard remains responsive and up-to-date. The Dashboard visualization can refer to Figure 6.

2.3 Testing

To ensure system correctness, reliability, and performance, a comprehensive testing process was conducted. This included component-level unit testing, integration testing across modules, API endpoint validation, and real-time performance evaluation in simulated deployment environments.

2.3.1 YOLO11 Model Fine-Tuning Validation

After fine-tuning the pre-trained YOLO11 model on the dataset in section 2.2.1.2, the model's performance was evaluated on the validation set to assess its accuracy in vehicle detection. Several key performance metrics were considered:

- **Precision (PP):** The proportion of correctly identified vehicles out of all predicted detections.
- **Recall (RR):** The proportion of correctly identified vehicles out of all actual vehicles present in the validation set.
- **Mean Average Precision at 0.5 IoU (mAP@0.5):** Measures the model's ability to correctly detect and localize vehicles when Intersection over Union (IoU) is at least 50%.
- **Mean Average Precision across multiple IoU thresholds (mAP@0.5:0.95):** Provides a stricter measure of detection accuracy by evaluating performance over a range of IoU thresholds.

The validation results are presented in Table 2, which summarizes the performance of the fine-tuned model across different vehicle classes:

Class	Images	Instances	Precision	Recall	mAP@0.5	mAP@0.5:0.95
All	486	1354	53.5	61.2	52.7	35.5
Bus	52	82	57.4	80.5	72.4	44.6
Car	135	679	55.2	63.0	61.9	40.9
Mini Bus	60	117	55.3	54.7	50.2	38.1
Motorcycle	55	113	40.0	47.7	24.3	14.1
Taxi	95	172	57.4	51.7	49.6	32.4
Truck	96	191	55.5	69.6	57.9	42.9

Table 2: Performance of Fine-Tuned YOLO11 Model

The fine-tuned YOLO11 model achieved an overall mAP@0.5 of 52.7%, indicating moderate detection accuracy across all vehicle types. However, the mAP@0.5:0.95 score of 35.5% suggests that the model's localization performance decreases at stricter IoU thresholds, indicating room for improvement in precise object bounding box estimation.

- **Buses** had the highest detection accuracy, achieving 72.4% mAP@0.5 and the best recall score of 80.5%, likely due to their large size and distinct features.
- **Motorcycles** had the lowest precision (40.0%) and recall (47.7%), indicating difficulty in detecting smaller vehicles, possibly due to occlusion or motion blur.

- **Trucks** had a high recall (69.6%) but a lower precision (55.5%), suggesting that while the model successfully identifies most trucks, it may also produce false positives.
- **Cars**, which are the most frequently occurring class, had a balanced precision (55.2%) and recall (63.0%), but their mAP scores indicate that bounding box localization could be further refined.

2.3.2 Edge Traffic Analyzer Testing

The Edge Traffic Analyzer is responsible for processing video input, performing vehicle detection and tracking, and executing zone-specific tasks such as traffic metrics extraction and illegal parking detection. To ensure its correctness and reliability, a series of tests were conducted to validate the system's performance in real-time traffic data analysis.

The following testing efforts focus on verifying the accuracy of traffic information extraction, the consistency of detection-tracking integration, and the proper functioning of task-specific logic. In particular, tests were designed to assess whether the system correctly calculates and updates key traffic metrics, including average speed, vehicle count, vehicle types, traffic flow rate, and congestion level, as well as the ability to flag illegal parking events under various configuration scenarios.

2.3.2.1 Unit Testing

Name	Stmts	Cover
src/api_client.py	83	92%
src/async_worker.py	41	100%
src/task_managers/parking_detection_manager.py	97	97%
src/task_managers/traffic_metrics_manager.py	133	100%
src/utils/config_loader.py	32	100%
src/utils/speed_calculator.py	51	100%
src/utils/utils.py	25	84%
src/utils/zone_setup.py	102	62%
src/video_processor.py	113	94%
Total	677	91%

Table 3: Edge Traffic Analyzer Code Coverage Summary

Unit tests were implemented to validate core functionalities across the traffic analytics pipeline, from video processing and zone configuration to task-specific logic and asynchronous API communication. The test suite achieved an overall coverage of 91%, demonstrating a high level of verification for system behavior. Most modules achieved near or full coverage, including `async_worker.py`, `speed_calculator.py`, `traffic_metrics_manager.py`, and `config_loader.py`, each at 100%, ensuring that foundational logic for speed computation, configuration loading, and task handling is thoroughly tested. The API client, a critical component for real-time dashboard communication, achieved 92% coverage, validating key request paths and response handling. The `parking_detection_manager.py` also reached 97% coverage, verifying its ability to track idle vehicles and detect violations accurately.

Although `zone_setup.py` reported a lower coverage of 60% due to its interactive GUI components for manual zone drawing, this does not affect the core analytics system and can be

safely excluded from automated tests. Overall, the test results confirm robust backend reliability, accuracy in real-time traffic metric extraction, and seamless integration with the API and dashboard.

2.3.3 API Endpoint Testing

The API Testing ensures that the system correctly communicates traffic data between the traffic data analytics system pipeline, database, and dashboard interface. The tests evaluate whether API endpoints are functioning correctly and returning accurate data in real-time.

2.3.3.1 API Functional Test



Figure 23: API documentation

For functional testing, we leveraged the interactive API documentation generated from the FastAPI framework. This documentation provides a user-friendly interface to manually explore and exercise every endpoint within the API 23. Using this interface, we specify parameters, submit request payloads, and observe live responses directly. The primary goals of this functional testing phase were to:

- Verify that each API endpoint operates in accordance with the specified requirements.
- Confirm that the API returns the correct status codes, messages, and data structures for both valid and invalid input.
- Ensure robustness by exploring edge cases manually, such as missing or malformed inputs.

The testing process included:

- Parameter Input: Manually entering test values into the input fields provided by the interactive documentation. This involved both correct payloads and scenarios with deliberately omitted or invalid parameters.
- Request Execution: Clicking the “Execute” button sends real requests to the running API.

- Response Verification: Comparing the returned responses against the expected results defined in our API specification. This included checking HTTP status codes (e.g., 200, 422, 500) and examining the response body for data conformity and correctness.
- Edge Cases: Testing boundary and error conditions (such as invalid data types, missing fields, and empty inputs) to ensure that the API handles these gracefully with appropriate error messages.

Results

All API endpoints were tested successfully using the interactive documentation. The observed results were as follows:

- **Endpoint Behavior:** Every endpoint returned the expected HTTP response. For valid inputs, responses returned status code 200 along with the correct data structure, and for invalid inputs (e.g., missing required fields), endpoints returned a 422 error with detailed validation error messages.
- **Error Handling:** Edge case inputs and improper requests were correctly identified by the API, and appropriate error messages were generated. Server error handling was also validated, ensuring that exceptions were caught and managed.
- **Overall Performance:** The interactive API testing verified that the entire set of endpoints functioned as intended, with response times within acceptable limits and all functional requirements satisfied.

2.3.4 Dashboard and Database Testing

Since Dashboard and Database are closely related, we decided to conduct the test together.

2.3.4.1 Unit Testing

Name	Stmts	Coverage
api.py	124	99%
app.py	34	65%
pages/illegal_parking.py	18	100%
pages/traffic_info.py	50	100%
src/chart_plot.py	140	100%
src/info_fetch.py	74	97%
Total	440	97%

Table 4: Dashboard Code Coverage Summary

For the unit test, we conduct the test for all functions across the codebase. The result for both the dashboard and the backend, including API and Redis database interactions, has produced robust results. Overall, the test suite achieved 97% coverage across the codebase. Notably, the API endpoints in `api.py` are nearly fully covered at 99%, while our pages for illegal parking and traffic information, along with the supporting modules for chart plotting and data fetching, have achieved complete or near-complete coverage. Although `app.py` has a lower coverage of 65%, reflecting the inherent complexity of UI layout and the dynamic nature of the dashboard,

which was tested in a later section, and does not impact the key functionalities. Overall, these results confirm that our unit tests effectively verify the critical backend logic and data processing while also providing reasonable confidence in the dashboard's reliability and integration with the database.

2.3.4.2 Dashboard Functional & UI Testing

For the Functional/UI testing of our dashboard, we conducted a series of manual tests to evaluate both the functionality and usability of the interface built with Streamlit. In this phase, testers navigated the dashboard interactively to simulate real user behavior. The manual tests focused on several key aspects:

- **User Navigation and Interaction:**

  Testers accessed the interactive API documentation and the dashboard to verify that critical user flows functioned correctly. For example, selecting a location and task from dropdown menus, clicking the "Go" button to load the corresponding dashboard page, and using the "Back" button to return to the initial selection screen were all verified. Each action was monitored to ensure that the corresponding updates in the session state occurred as expected and that the dashboard responded promptly.

- **UI Component Verification:**

  The dashboard's visual components (e.g., maps, charts, and metrics) were scrutinized. Testers confirmed that the rendered charts (e.g., traffic flow, vehicle types, and speed trends) accurately represented the underlying data retrieved from Redis. Attention was paid to the consistency of visual styling, the clarity of displayed information, and the overall layout of the dashboard.

- **Data Integrity and Error Handling:**

  By manually inputting both valid and invalid data through the interactive elements, testers verified that the system correctly processed inputs and produced the corresponding outputs. In cases of invalid input, appropriate error messages and UI cues were confirmed, ensuring a graceful handling of edge cases and providing clear feedback to users.

- **Overall User Experience:**

  In addition to functional correctness, qualitative feedback was collected on the dashboard's responsiveness, ease of navigation, and overall aesthetic appeal. The manual evaluation confirmed that the dashboard not only met its functional specifications but also provided a user-friendly interface with intuitive navigation and real-time data refresh.

Results

All manual tests confirmed that the dashboard behaves as intended. User interactions resulted in the correct state transitions and appropriate rendering of visualizations and metrics. Error conditions were handled gracefully, and the overall look and responsiveness of the interface met our design criteria. The manual functional/UI testing hence corroborated the robustness and usability of the dashboard, ensuring that it fulfills both its operational and user experience requirements.

2.3.4.3 Integration and Performance Testing

To validate the overall system performance, we conducted comprehensive integration and performance testing across the complete processing pipeline, from the live raw video input to the dashboard display. This end-to-end evaluation encompasses all crucial components, including the vehicle detection and tracking, traffic metric extraction, illegal parking detection, API client POST requests, database storage in Redis, and the subsequent retrieval of data and real-time render visualizations on the Streamlit dashboard.

The testing scenario simulated a live deployment by continuously processing a high-resolution video feed (2560×1440) consisting of 1500 frames in our development environment (see Appendix 8.2 for hardware specifications). The Edge Traffic Analyzer successfully detected and tracked vehicles, extracted relevant traffic metrics, and identified illegal parking events in real time. The extracted data was transmitted asynchronously via API POST requests, processed by the backend server, and stored in Redis. The dashboard then retrieved this data and rendered dynamic visualizations with minimal latency.

Performance measurements are summarized in Table 5.

Device	Frames Processed	Time Taken (sec)	Average FPS
CPU	1500	199.66	7.51
GPU (CUDA)	1500	48.66	30.82

Table 5: Performance Comparison of CPU and GPU

The overall latency from video capture through data processing, API communication, to dashboard rendering is minimal enough to be considered negligible. This indicates that our development hardware is capable of handling real-time processing demands efficiently without introducing noticeable delays.

To conclude, the integration and performance testing confirmed that the system's components work seamlessly together, delivering a robust, real-time traffic analysis and monitoring solution. The achieved throughput and low latency ensure that the dashboard accurately reflects live conditions, enabling responsive and reliable traffic management in a real-world scenario.

2.4 Evaluation

Upon successful completion of the testing phase, the project underwent a comprehensive evaluation to determine whether each of the stated objectives had been met. This evaluation focuses on the functional performance and integration of all components, mapped back to the project objectives outlined in Section 1.2.

2.4.1 Edge Traffic Analyzer Evaluation

2.4.1.1 Vehicle Detection and Tracking System

The YOLO11 model fine-tuned for vehicle detection achieved an overall mAP@0.5 of 52.7%, indicating moderate detection accuracy in detecting common Hong Kong vehicle types in diverse environments. An analysis of incorrect predictions revealed several common sources of error:

1. **Class Confusion:**

- Some trucks were misclassified as buses, likely due to their similar shape and appearance.
- Mini buses and taxis had overlapping features, contributing to occasional misidentifications.

2. **Low Confidence in Small Vehicles:**

- Motorcycles had significantly lower recall and mAP scores, indicating missed detections or low-confidence predictions, especially in crowded environments.
- Images where the vehicles are farther away from the camera are more likely to be missed.

Despite having room for improvements, the vehicle detection model combined with the ByteTrack tracking algorithm, the system was able to consistently maintain unique identities across frames, enabling accurate calculation of per-vehicle metrics such as speed and presence duration. The evaluation confirms that the system successfully established a robust detection-tracking pipeline suitable for real-time traffic analysis.

2.4.1.2 Zone-Specific Task Management

The modular task manager framework, with each zone supporting configurable task instances, enabled the system to handle multiple analytics tasks concurrently. Unit tests and integration tests validated the behavior of both `TrafficMetricsManager` and `ParkingDetectionManager`, with logic correctly applied per zone and accurate metrics produced.

This design proved highly scalable, allowing zone behavior to be adjusted by updating configuration files without altering core logic. Thus, confirming that the system can support dynamic zone-specific task execution.

2.4.1.3 Traffic Metrics Extraction

Testing confirmed that the TrafficMetricsManager accurately calculates and updates key traffic metrics in real time, including average speed, vehicle count, traffic flow rate, vehicle type distribution, and congestion level. The rolling average mechanism for speed and count enabled meaningful congestion classification using VCI and MAS indexes.

All metrics were periodically sent to the dashboard API at configurable intervals. These results confirm that the system is successful in extracting traffic metrics with high accuracy and responsiveness.

2.4.1.4 Illegal Parking Detection

Testing confirms that the `ParkingDetectionManager` effectively identified illegal parking violations, incorporating realistic constraints such as vehicle types and restricted time windows typical in Hong Kong. Vehicles violating specified criteria were consistently detected, with violations triggering real-time API snapshots correctly.

While effective, the algorithm is inherently vulnerable to occlusions, which can interrupt the continuity of vehicle tracking. When an idle vehicle is temporarily obscured or lost due to occlusion, its timer is reset upon re-identification, resulting in false negatives cases where a violation should have been flagged but was missed.

2.4.1.5 Real-Time Data Transmission

The asynchronous `AsyncWorker` and API client implementation achieved reliable, non-blocking communication with the dashboard, handling high-frequency data without impacting real-time processing. Testing confirmed accurate transmission of vehicle counts, speeds, flow rates, congestion levels, and illegal parking frames.

Interactive API testing showed successful validation of expected behaviors and appropriate error handling. The architecture ensured consistent data delivery without blocking the main processing thread. This confirms successful fulfillment of real-time data transmission.

2.4.2 Dashboard Evaluation

2.4.2.1 Receiving and Processing Data Evaluation

We established a reliable data pipeline to receive real-time traffic data from the edge traffic analyzer. The dashboard's API endpoints effectively received and processed incoming requests and have been rigorously tested using both automated unit tests and manual functional tests through the interactive API documentation. These tests confirmed that incoming API requests are correctly handled by payloads are properly validated with robust error handling and data is reliably processed. Any issues with invalid inputs were promptly identified and reported via appropriate status codes.

2.4.2.2 Real-Time Storage and Retrieval Evaluation

Our backend employs Redis in-memory database, to store traffic data at low latency. Integration testing confirmed that data written into Redis conforms to the expected formats and is accurately retrieved. Low-latency during integration testing showed that the complete processing pipeline (from raw video input to dashboard data retrieval) introduces negligible processing time with enough computer power.

2.4.2.3 Visualization and Dashboard Functionality Evaluation

The dashboard, built with Streamlit, was designed to offer an intuitive and responsive interface for traffic administrators. Functional and UI testing, both automated and through manual user feedback, confirmed that navigation between pages is seamless, with smooth transitions and rapid updates. Key components such as traffic metrics visualizations, maps, and illegal parking detection snapshots alert are rendered correctly and provide clear, actionable insights. However, the relatively lower unit-test coverage (65%) for UI components highlighted the inherent complexity of dynamic user interfaces, but these did not compromise overall functionality. By conducting additional UI functional tests on the user, we could enhance test coverage and UI reliability.

2.4.2.4 System Integration and Performance Evaluation

The integration testing simulated an end-to-end real-world scenario, starting from live raw video inputs processed by the edge traffic analyzer (covering vehicle detection, tracking, and traffic metric extraction) through to API requests and, finally, dashboard visualization. The performance tests demonstrated that the edge traffic analyzer can achieve approximately 30 fps with GPU support, while API transmission depends on sending frequency and can be done with

nearly real-time, with minimal overall latency. This confirms that our processing pipeline is optimized for real-time traffic monitoring and analysis under the current loading.

What Went Well

- **Robust data pipeline:** All components of the data pipeline effectively communicate and process data with high reliability, as evidenced by both unit and integration test results.
- **Real-time performance:** The system achieved the targeted throughput (30 fps) and minimal latency, validating that the chosen hardware and software stack are well-suited for real-time processing.
- **Edge analytics robustness:** The Edge Traffic Analyzer consistently processed raw video streams locally with high accuracy, effectively executing detection, tracking, and zone-specific task management. This robust performance underpins the reliable extraction of traffic metrics and illegal parking events.
- **User interface:** The dashboard provides clear and intuitive visualizations, enabling traffic administrators to quickly understand and react to live traffic conditions.
- **High test coverage:** With overall unit test coverage for the Edge Traffic Analyzer at 91% and the Dashboard at 97%, critical functionalities—including detection and tracking logic, zone-specific task execution, API communication, data processing, and real-time visualization—have been thoroughly validated.

Areas for Improvement

- **Scalability under high load:** Although our performance tests confirm robust processing under current conditions, further testing under increased load (with distributed inputs and higher data volumes) is warranted to ensure scalability for future deployments.

3 Discussion

3.1 YOLO11 Model Fine-Tuning Performance

As table 2 shows, the fine-tuned YOLO11 model demonstrated moderate detection accuracy, performing well for larger vehicles such as buses and trucks, but struggling with smaller objects like motorcycles. Although vehicles are more likely to be undetected when they are far away from the camera and appear small, as they move closer, they become larger and thus easier to detect.

To improve the model's performance, the following enhancements could be considered:

- Hard negative mining to increase robustness against false positives.
- Additional labeled data for underperforming classes, particularly motorcycles and taxis.
- Post-processing techniques such as confidence threshold adjustments to reduce misclassifications.
- Data augmentation strategies to improve performance under varied lighting and occlusion conditions.

Despite some limitations, the model remains suitable for real-time traffic monitoring applications, providing a strong foundation for vehicle detection in urban environments.

3.2 Dynamic Expendable System with Various Tasks

Our system is designed to be dynamically expandable, allowing for the effortless integration of new tasks and camera locations as long as the configuration is appropriately set. This flexibility means that our traffic analytics platform can be easily extended to cover additional regulatory violations or operational scenarios, such as monitoring non-compliance with traffic signals or tracking unauthorized pick-up and drop-off activities in restricted zones.

A key enabler of this dynamic extensibility is the Zone Setup Tool, which facilitates the easy configuration of new zones in any given location. With this tool, users can interactively define zone boundaries by simply clicking on points in a reference image captured from the camera. The tool then generates and updates the corresponding configuration settings automatically. This not only reduces the configuration overhead but also minimizes the potential for errors when setting up new locations.



Figure 24: Steps for Zone setup tool

The ease-of-use provided by the Zone Setup Tool enables administrators to rapidly deploy the system across multiple sites with minimal manual intervention and high configurability. Once a live camera is properly set up and adequate computing resources are available, users can define analysis zones through the tool and configure task-specific settings with ease. New

visualization pages (similar to the existing ones for traffic metric extraction and illegal parking detection) can then be developed and seamlessly integrated into the dashboard for real-time monitoring. As a result, the system not only meets current traffic management demands but also provides a scalable and adaptable framework capable of evolving alongside regulatory changes and diverse urban conditions.

3.3 Illegal Parking Detection Real-world Scenario

The current illegal parking detection system employs a straightforward time-based approach, whereby vehicles are flagged if they remain idle below a specific speed threshold for a designated period. While this method provides a basic mechanism for identifying potential violations, it does not fully encompass the complexity of real-world traffic situations. For instance, in areas where a traffic signal exists, such as a red light, the vehicle may naturally come to a stop. In such cases, the idle time should be interpreted differently, as stopping for a red light is a legitimate traffic behavior rather than a parking violation.

Moreover, the current implementation is inherently vulnerable to occlusions. When a vehicle is temporarily obscured in heavy traffic, the continuity of its tracking can be disrupted, leading to a reset of its idle timer and the risk of false negatives, where a genuine violation may be overlooked.

Future work should aim to enhance the robustness of illegal parking detection by incorporating additional contextual factors such as real-time traffic signal data and employing more advanced tracking algorithms that can withstand short-term occlusions. By taking these factors into account, the system can more accurately distinguish between legitimate stops and illegal parking, ultimately providing a more reliable tool for traffic management in complex urban environments.

3.4 Database Implementation

Traditionally, Redis is used as a cache in front of another ”real” database, such as MySQL or PostgreSQL, to improve application performance. It leverages the speed of memory to alleviate load on the central application database for data that changes infrequently and is requested often, as well as for less mission-critical data that evolves frequently [20].

In our current system architecture, we utilize Redis as our sole database, taking advantage of its in-memory capabilities for fast read and write operations. However, this approach presents challenges regarding scalability. Although Redis regularly dumps the database to disk, there are risks associated with limited storage capacity and significant data transfer between memory and disk. As our processed data grows, we risk overwhelming the database and compromising availability.

To address these issues, we propose two key strategies:

- **Integrating an Additional Disk-Based Database:** This will allow for better caching strategies within Redis by:
  - **Prime Cache:** Establish a prime cache strategy to fill the caches with data so that subsequent Directory Server behavior reflects normal operational performance, rather than ramp-up performance. This ensures that frequently accessed data is readily available [21].
  - **Cache Hit and Cache Miss Handling:** Implement effective mechanisms to manage cache hits and misses, optimizing data retrieval processes.

- **Incorporating Redis Plug-ins and High Availability Setups:** Redis offers features that enable it to function as a full-fledged primary database when coupled with plug-ins and high availability configurations. This approach eliminates single points of failure, allowing systems to recover gracefully and quickly. Reliable data crossover ensures that no data is lost during transitions from primary to secondary databases, alongside automatic failure detection and recovery.

These strategies will enhance our system's scalability and reliability, enabling us to manage increasing data volumes effectively while ensuring optimal performance for real-time analytics. Future work will focus on optimizing these processes and exploring alternative storage solutions to further improve our data management capabilities, including investigating distributed database architectures and hybrid storage solutions that combine the strengths of both in-memory and disk-based systems.

3.5 Lower System Workload for Edge Traffic Analyzer

In the current implementation of the Edge Traffic Analyzer, the pre-recorded video input is processed on a computer with GPU support to be able to achieve near real-time analytics (~30 fps); however, with only CPU support, the system can only process up to 10 fps.

Several attempts were made to try to make the system workload lower to be able to potentially be run on an edge device, achieving the desired goal of having Edge computing for the Edge Traffic Analyzer:

- **Smaller computer vision model:** The main performance bottleneck of the system is associated with the computing power needed for the object detection model, in which the model size heavily influences the processing speed of the whole system. However, smaller models have the trade-off of lower accuracy in detecting the vehicles, which is crucial for correct traffic data extraction.

- **Stride in video processing:** By increasing the strides in the video processing frames, the frames needed to be processed per second can be dropped significantly. For example, with a stride of 3, a 30 fps video only needs to be processed every third frame, lowering the effective fps to 10. However, with an increased stride in the frames, the temporal gap between frames is also increased, which conflicts with the internal calculation of the ByteTrack tracking algorithm, thus significantly lowering its ability to track vehicles, and breaking the system's most traffic calculation mechanisms.

Future work will be research on additional methods for lowering the system workload to be able to run on edge devices.

3.6 Hardware Limitation during Development Phase

During the development phase, our system was tested using pre-recorded video instead of live camera feeds due to our current edge device's insufficient GPU support. While this approach allowed us to develop, validate, and fine-tune our detection model and tracking algorithms under controlled conditions, it did not fully replicate the performance dynamics of a live environment. This hardware limitation meant that, although we could ensure the correctness and robustness of our processing pipeline, the real-time throughput and latency metrics may differ once a GPU-enabled edge device is deployed.

Additionally, for the illegal parking detection task, the actual cameras have not yet been set up and deployed. As a practical workaround, we utilized pre-recorded video from the Kwun Tong area for demonstration purposes. This allowed us to simulate real-world scenarios and test our algorithms effectively. The use of pre-recorded video not only facilitated comprehensive testing and iterative improvement but also enabled us to showcase our system's capabilities to stakeholders.

If the edge device's computing power is sufficient, we anticipate that actual deployment will benefit from enhanced processing speeds, further reducing latency and achieving the desired real-time performance. We are optimistic that transitioning to live camera feeds will provide more accurate insights and improve our system's overall effectiveness in smart traffic analytics.

4 Conclusion

The development of our AI-powered smart traffic analytics system specifically addresses the complex traffic management challenges faced in Hong Kong's dense urban environment. This project successfully delivered a comprehensive, end-to-end solution that begins with real-time vehicle detection and culminates in an intuitive data analytics dashboard. By fine-tuning the YOLO11 model with locally relevant datasets, we significantly improved vehicle detection accuracy, specifically tailored to Hong Kong's unique traffic conditions. Furthermore, our custom-designed algorithms effectively extract essential traffic metrics from raw camera inputs, providing accurate insights into vehicle counts, average speeds, congestion levels, and illegal parking events.

The modular architecture of the system, which includes configurable settings for diverse urban scenarios, enhances adaptability and ease of deployment. This design ensures that the system can readily scale or adjust functionalities to meet varying requirements of different urban locations. The robust data pipeline developed facilitates seamless transmission and efficient storage of real-time traffic data, ensuring reliability and quick accessibility for users. Additionally, the user-friendly dashboard interface makes complex traffic data readily interpretable, significantly aiding administrators in making informed traffic management decisions.

Despite these achievements, several areas present opportunities for future improvement and research. First, the current illegal parking detection module can be further enhanced to increase its robustness, especially under challenging scenarios such as vehicle occlusion or tracking interruptions. Secondly, integrating a disk-based database would also be beneficial for handling extensive historical data and further improving data retrieval and analytics capabilities. Lastly, system optimization efforts aimed at reducing the computational workload could enable broader deployment scenarios, particularly on lower-powered edge devices, thereby expanding the system's applicability and practicality across a wider range of environments.

5 References

[1] "The Annual Traffic Census 2022," Tech. Rep. [Online]. Available: https://www.td.gov.hk/filemanager/en/content_5200/Annual%20Traffic%20Census%202022.pdf.

[2] "Hong Kong Smart City Blueprint 2.0," Tech. Rep. [Online]. Available: https://www.smartcity.gov.hk/modules/custom/custom_global_js_css/assets/files/HKSmartCityBlueprint(ENG)v2.pdf.

[3] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics YOLO*, original-date: 2022-09-11, Jan. 2023. [Online]. Available: https://github.com/ultralytics/ultralytics.

[4] Y. Zhang, P. Sun, Y. Jiang, *et al.*, "ByteTrack: Multi-Object Tracking by Associating Every Detection Box," 2021. doi: 10.48550/ARXIV.2110.06864. [Online]. Available: https://arxiv.org/abs/2110.06864.

[5] *Redis/redis*, original-date: 2009-03-21T22:32:25Z, Feb. 2025. [Online]. Available: https://github.com/redis/redis.

[6] *Streamlit/streamlit*, original-date: 2019-08-24T00:14:52Z, Feb. 2025. [Online]. Available: https://github.com/streamlit/streamlit.

[7] *Smartmicro Traffic Data Sensor*, en-US. [Online]. Available: https://www.smatstraffic.com/radar-sensors/.

[8] *HI-TRAC® TMU4 | High-speed WIM and traffic data collection*, en-US. [Online]. Available: https://www.q-free.com/product/hi-trac-tmu4-high-speed-weigh-in-motion/.

[9] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," en, *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, Mar. 1960, issn: 0021-9223. doi: 10.1115/1.3662552. [Online]. Available: https://asmedigitalcollection.asme.org/fluidsengineering/article/82/1/35/397706/A-New-Approach-to-Linear-Filtering-and-Prediction.

[10] *GitHub - fastapi/fastapi: FastAPI framework, high performance, easy to learn, fast to code, ready for production*, en. [Online]. Available: https://github.com/fastapi/fastapi.

[11] *Uvicorn*. [Online]. Available: https://www.uvicorn.org/.

[12] *Roboflow Universe: Computer Vision Datasets*, en. [Online]. Available: https://universe.roboflow.com/.

[13] M. Tkachenko, M. Malyuk, A. Holmanyuk, and N. Liubimov, *Label Studio: Data labeling software*, 2020. [Online]. Available: https://github.com/HumanSignal/label-studio.

[14] *Supervision*. [Online]. Available: https://github.com/roboflow/supervision.

[15] *Core - Supervision*. [Online]. Available: https://supervision.roboflow.com/0.24.0/detection/core/#supervision.detection.core.Detections.

[16] *OpenCV: Geometric Image Transformations*. [Online]. Available: https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html.

[17] *Polygon Zone - Supervision*. [Online]. Available: https://supervision.roboflow.com/detection/tools/polygon_zone/.

[18] *Pandas documentation — pandas 2.2.3 documentation*. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/.

[19] *Plotly*. [Online]. Available: https://plotly.com/python/plotly-express/.

[20] M. Yusuf, *Redis Explained*, en. [Online]. Available: https://architecturenotes.co/p/redis.

[21] *Priming Caches (Sun Directory Server Enterprise Edition 7.0 Administration Guide)*. [Online]. Available: https://docs.oracle.com/cd/E19424-01/820-4809/gbsfb/index.html.

[22] B. Le, *An Introduction to BYTETrack: Multi-Object Tracking by Associating Every Detection Box*, en. [Online]. Available: https://www.datature.io/blog/introduction-to-bytetrack-multi-object-tracking-by-associating-every-detection-box.