ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Σχολή Τεχνολογίας

Τμήμα Ψηφιακών Συστημάτων

Π.Μ.Σ. Σύγχρονα Συστήματα Επικοινωνιών και Διαδίκτυο των Πραγμάτων

Σχεδίαση και Ανάπτυξη Μη Εποπτευόμενου Συστήματος IoT για Αυτοματοποιημένη Παρακολούθηση Κυκλοφορίας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Φαλτάκας (ΑΜ: Μ023123022)

Επιβλέπων: ***Γεώργιος Καρέτσος, Καθηγητής***

ΛΑΡΙΣΑ, Ιούνιος 2025

UNIVERSITY OF THESSALY

School of Technology

Digital Systems Department

MSc in Modern Communication Systems and Internet of Things

Design and Deployment of a Headless IoT System for Automated Traffic Monitoring

MASTER THESIS

Georgios Faltakas (RN: M023123022)

Supervisor: ***Georgios Karetsos, Professor***

LARISSA, June 2025

Υπεύθυνη Δήλωση περί Ακαδημαϊκής Δεοντολογίας και Πνευματικών Δικαιωμάτων

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα πτυχιακή εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον, και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή.

Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο Δηλών

(Υπογραφή)

Γεώργιος Φαλτάκας

23/6/2025

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων	**Γεώργιος Καρέτσος** Καθηγητής Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Φώτιος Γκιουλέκας** Επιστημονικός Συνεργάτης Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Δημήτριος Κοσμάνος** Επιστημονικός Συνεργάτης Τμήμα Ψηφιακών Συστημάτων, Πανεπιστήμιο Θεσσαλίας

Ημερομηνία έγκρισης: 30-6-2025

Περίληψη

Οι εξελίξεις στο Διαδίκτυο των Πραγμάτων (IoT), το edge computing, και την υπολογιστική όραση (computer vision) μετασχηματίζουν τον τρόπο με τον οποίο συλλέγονται, επεξεργάζονται και αναλύονται δεδομένα σε εφαρμογές έξυπνης κινητικότητας. Ωστόσο, πολλές υφιστάμενες λύσεις δεν έχουν ξεπεράσει το στάδιο της θεωρητικής προσέγγισης και δεν αξιοποιούν αποτελεσματικά αυτές τις τεχνολογίες, περιορίζοντας τη δυνατότητα εφαρμογής τους σε πραγματικά περιβάλλοντα. Στο πλαίσιο αυτό, η παρούσα εργασία εξετάζει τον σχεδιασμό, την ανάπτυξη και την υλοποίηση ενός πλήρως αυτοματοποιημένου, μη εποπτευόμενου IoT συστήματος για παρακολούθηση της κυκλοφορίας, το οποίο ενσωματώνει τεχνικές υπολογιστικής όραση σε ένα ευρύτερο, λειτουργικό application framework. Για την επίτευξη αυτού του στόχου, υιοθετήθηκε μια αποκεντρωμένη, edge-to-cloud, event-driven αρχιτεκτονική. Το σύστημα καταγράφει βίντεο μέσω ενός edge device και τα ανεβάζει σε υπηρεσία αποθήκευσης στο cloud. Ένα webhook ανιχνεύει κάθε νέο αρχείο και ενεργοποιεί το central processing pipeline σε έναν υπολογιστή γενικής χρήσης χωρίς επιτάχυνση GPU. Χρησιμοποιώντας ένα προεκπαιδευμένο μοντέλο YOLO, η μονάδα επεξεργασίας εκτελεί το computer vision pipeline για ανίχνευση, ταξινόμηση, εκτίμηση ταχύτητας και ανάλυση συμφόρησης οχημάτων. Τα εξαγόμενα δομημένα δεδομένα αποθηκεύονται σε μία βάση δεδομένων χρονοσειρών και γίνονται `publish` μέσω MQTT, ώστε να είναι διαθέσιμα τόσο για ανάλυση σε σχεδόν πραγματικό χρόνο, όσο και για ιστορική επεξεργασία. Το τελικό αποτέλεσμα είναι ένα πλήρως λειτουργικό και αυτόνομο σύστημα, το οποίο επιδεικνύει σταθερή απόκριση και αποδοτική χρήση πόρων, αποδεικνύοντας την καταλληλότητά του για παρακολούθηση της κυκλοφορίας σε έξυπνες πόλεις σε σχεδόν πραγματικό χρόνο, δίνοντας παράλληλα έμφαση στην επεκτασιμότητα, την προσαρμοστικότητα και το χαμηλό κόστος υλοποίησης.

Abstract

Advancements in IoT, edge computing, and computer vision are transforming the way data is collected, processed, and analyzed in smart mobility applications. However, many existing solutions remain theoretical and fail to effectively leverage these technologies, limiting their applicability in real-world deployments. To this end, this thesis explores the design, development, and deployment of a fully automated, headless IoT traffic monitoring system that integrates computer vision into a broader, functional application framework. To achieve this goal, a decoupled edge-to-cloud, event-driven, architecture was adopted. The system captures video segments through an edge device and uploads them to a cloud-based storage service. An event-driven pipeline is triggered via webhook notifications upon file upload, initiating centralized processing on a general-purpose PC without GPU acceleration. Using a pre-trained YOLO model, the processing unit executes a computer vision pipeline that performs vehicle detection, classification, tracking, speed estimation, and congestion analysis. The extracted metrics are structured, stored in a time-series database, and published via MQTT for both near real-time and historical analysis. The result is a fully functional, autonomous system that demonstrates consistent end-to-end latency and efficient resource utilization, validating the architecture's suitability for near-real time traffic monitoring in smart city environments while promoting scalability, adaptability, and cost-effectiveness.

Ευχαριστίες

Η περάτωση της παρούσης διπλωματικής εργασίας σηματοδοτεί την ολοκλήρωση των σπουδών μου στο Π.Μ.Σ. «Σύγχρονα Συστήματα Επικοινωνιών και Διαδίκτυο των Πραγμάτων» του Τμήματος Ψηφιακών Συστημάτων του Πανεπιστημίου Θεσσαλίας. Θα ήθελα να ευχαριστήσω ιδιαίτερα τον επιβλέποντα καθηγητή μου, τον κ. Γεώργιο Καρέτσο, με τον οποίο είχα εξαιρετική συνεργασία καθ' όλη τη διάρκεια εκπόνησης της διπλωματικής εργασίας μου. Η πρόσβαση που μου παρείχε στο εργαστήριο Ασύρματων & Κινητών Τηλεπικοινωνιακών Συστημάτων, καθώς και ο απαραίτητος τεχνολογικός εξοπλισμός που διέθεσε, συνέβαλαν αποφασιστικά στην επιτυχή ολοκλήρωση της εργασίας. Θα ήθελα επίσης να ευχαριστήσω τους συναδέλφους μου στο εργαστήριο Ασύρματων & Κινητών Τηλεπικοινωνιακών Συστημάτων για την εποικοδομητική συνεργασία. Τέλος, πάνω από όλα θα ήθελα να ευχαριστήσω την οικογένεια μου για τη διαρκή στήριξη της καθ' όλη τη διάρκεια της εκπαιδευτικής πορείας μου.

Γεώργιος Φαλτάκας

23/6/2025

Contents

1 IoT and Computer Vision for Smart Traffic Monitoring

The growing complexity of urban environments, combined with advancements in digital technologies, has led to the emergence of smart traffic systems that rely on real-time data acquisition and analysis. Central to this evolution is the combination of Internet of Things (IoT), edge computing, and computer vision, which enables the development of smart infrastructure and responsive smart city services. This chapter provides a technical overview of how these technologies work and how they are applied in traffic monitoring systems.

1.1 Internet of Things and Edge Computing

The internet of Things (IoT) refers to the global network of physical devices, ranging from household appliances to industrial machinery, that are embedded with sensors, actuators, software, and communication technologies, allowing them to collect, transmit, and receive data over the internet without human intervention. This interconnectivity creates intelligent environments capable of reacting to real-world conditions in real-time, supporting applications in healthcare, agriculture, manufacturing, and especially in smart cities, where IoT facilitates traffic control, environmental monitoring, and infrastructure management.

In such smart environments, IoT devices generate vast volumes of data that need to be processed efficiently. Traditional centralized computing models, which rely on transmitting all data to cloud servers, often suffer from bandwidth constraints and high latency. These limitations become more apparent as the number of connected devices increases rapidly [1].

To tackle these challenges, edge computing has emerged as a significant architectural shift. Edge computing is a distributed computing framework that brings enterprise applications closer to data sources such as IoT devices or local edge servers. Instead of offloading all raw data to centralized servers for processing, edge computing allows data to

be processed locally, directly at or near its point of origin, using IoT sensors, smart gateways, or embedded computing devices. This decentralized approach reduces latency, alleviates network bandwidth demands, and improves system responsiveness, especially in time-sensitive or connectivity-constrained environments [2].

Edge devices, which tend to be typically small-scale and energy-efficient computing platforms, are used to execute localized operations such as data acquisition, control tasks, or lightweight automation. In real-world deployments, these devices manage peripheral components as well, such as sensors or cameras, and handle tasks like capturing media, pre-processing content, and securely transmitting files to cloud-based services. In such architectures, edge devices often serve as intermediaries between the physical environment and cloud infrastructure, facilitating timely and context-aware interaction with remote systems. This edge-to-cloud paradigm provides a balanced framework. Specifically, edge computing provides immediate, task-specific execution and autonomy, while cloud platforms enable centralized coordination, long-term storage, and higher-order analysis.

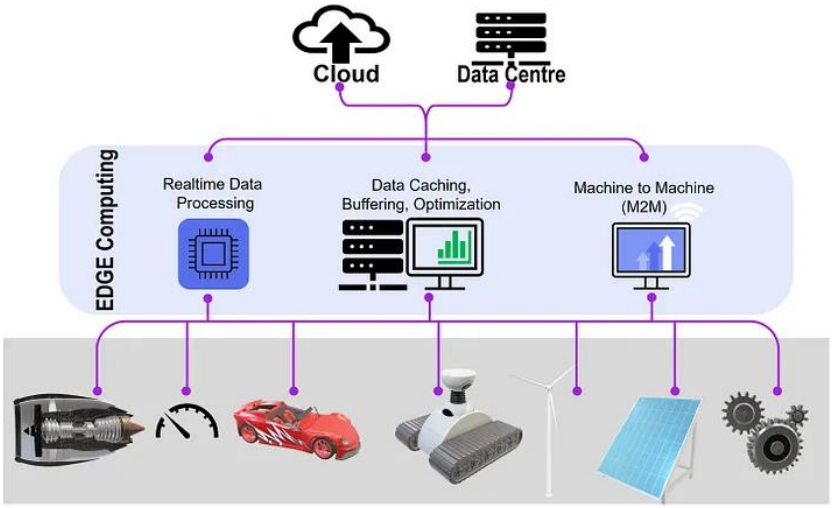


Figure 1: Edge Computing hierarchy [1]

Together, IoT and edge computing form a synergistic foundation for building distributed, intelligent systems capable of operating in real-world environments. Whether applied to smart infrastructure, industrial systems, or context-aware monitoring, this integration supports timely execution of localized tasks, while maintaining connectivity with cloud services for storage and accessibility. This layered approach enables systems to operate autonomously at the edge, while remaining scalable and manageable at a central level [3].

1.2 Smart Traffic Management

The emergence of smart traffic management systems as a key IoT application highlights the growing need for more efficient, adaptive, and data-driven urban transportation systems. By embedding IoT-enabled sensors, cameras, and compact computing units across road networks, cities and operators can monitor and analyze key mobility indicators regarding real-time traffic management.

One of the most complex challenges urban environments around the world face is traffic congestion, which results in increased travel times, fuel consumption, emissions, and economic losses. Traditional traffic monitoring approaches, such as manual counting or inductive loop detectors, often fail to provide the granular, real-time data needed for proactive response. In contrast, modern smart mobility systems leverage edge and cloud technologies to continuously capture and analyze traffic-related metrics that are directly relevant to mobility flow and congestion dynamics [4].

Key metrics include vehicle counts, vehicle type classification (e.g., differentiating between motorcycles, cars, buses, and trucks), and vehicle speed estimation. These data points can be further used to compute traffic intensity (vehicles per unit time) and assess congestion levels. The need for automated, real-time traffic analytics is a given, as cities aim to support rising populations and mobility demands. Systems capable of extracting these metrics autonomously reduce dependency on expensive infrastructure and human intervention, making them scalable and cost-effective. Furthermore, such data is essential for developing predictive models of traffic behavior, supporting smarter infrastructure planning and adaptive traffic control strategies [5].

Ultimately, combining IoT with real-time traffic analytics leads to more responsive and resilient urban mobility ecosystems, where decisions rely on constant, localized observations rather than rigid models or delayed reports.

1.3 Computer Vision in Smart Traffic Management

In the evolving landscape of urban transportation, computer vision has emerged as a pivotal technology within smart traffic management systems. By enabling machines to interpret and process visual data from traffic environments, computer vision facilitates real-time monitoring and decision-making, enhancing the efficiency and safety of road networks.

1.3.1 What is Computer Vision?

Computer Vision (CV) is a rapidly evolving specialized subfield of Artificial Intelligence (AI) focused on enabling computers and systems to extract meaningful insights from digital images, videos, and various visual inputs. Its primary goal is to mimic human visual perception, enabling machines to "see," interpret, and make judgments based on visual data. This process involves analyzing visual data using complex algorithms and deep learning (DL) models to identify objects, comprehend scenes, and extract high-level insights. In contrast to simple image processing, which mainly focuses on enhancing or manipulating image data (e.g., resizing, filtering, or adjusting brightness), computer vision is concerned with extracting semantic meaning and contextual information from visual scenes.

Computer Vision is fundamental to numerous contemporary AI and Machine Learning (ML) systems, equipping machines with the ability to perceive and understand the physical world through visual perception. The introduction of Convolutional Neural Networks (CNNs), which draw inspiration from the human visual cortex, has revolutionized the field of CV. These networks allow models to automatically discover layered features from vast amounts of visual information, resulting in significant enhancements in precision across various computer vision tasks. This advancement facilitates the development of advanced applications that were once beyond reach, establishing CV as a foundation of the ongoing AI development.

Computer vision involves a broad set of tasks, each designed to extract specific forms of insight from visual data. Some of the most significant tasks include:

- *Image Classification*: Assigning a label to an entire image (e.g., determining if an image contains a car or a truck). This task often relies on large-scale labeled datasets such as COCO (Common Objects in Context) and ImageNet.
- *Object Detection*: Identifying and localizing multiple objects within an image by drawing bounding boxes around them and assigning class labels. Algorithms like YOLO (You Only Look Once) are widely adopted for real-time object detection due to their speed and accuracy.
- *Image Segmentation*: Classifying each pixel in an image to delineate different objects or regions. This includes:
  - *Semantic Segmentation*: Labeling pixels by category (e.g., road, car, pedestrian).

    - *Instance Segmentation*: Distinguishing between individual objects of the same category (e.g., two separate cars).
- *Object Tracking*: Monitoring the movement of one or more objects across video frames, combining object detection with temporal analysis.
- *Optical Flow*: Estimating motion between consecutive video frames, either for object tracking or understanding the movement of the camera itself.
- *Pose Estimation*: Detecting key points or joints of objects or people to infer spatial orientation. This is frequently used in gesture recognition and biomechanics. [6]

These tasks are often interconnected and can be combined to enable applications such as autonomous navigation, traffic monitoring, robotic perception, and augmented reality. As computational power continues to grow and datasets become more comprehensive, the capabilities of computer vision systems are expected to expand further, driving innovation across many sectors. Computer vision technologies are increasingly being integrated into a wide range of real-world systems, driving innovation across diverse sectors.

In the field of autonomous vehicles, computer vision plays a pivotal role in enabling cars to understand their surroundings. Specifically, it facilitates pedestrian and vehicle detection, lane following, traffic sign recognition, and obstacle avoidance. Industry leaders have built entire perception stacks, which are heavily reliant on advanced CV models.

Security and surveillance systems have also evolved through CV-powered automation, with applications ranging from motion detection and intrusion alerts to facial recognition, crowd analysis, and behavioral monitoring.

In healthcare, computer vision supports clinicians in diagnosing medical conditions by analyzing X-rays, MRIs, and CT scans for anomalies such as tumors or fractures. It is also used in robotic-assisted surgeries, patient monitoring, and even predictive diagnostics, contributing to faster and more accurate medical decisions.

In the retail sector, computer vision is employed for tasks such as inventory tracking and customer behavior analysis. Similarly, manufacturing environments use CV for quality control, defect detection, and real-time process control, improving product consistency and reducing manual errors. In industrial settings, computer vision is referred to as machine vision, emphasizing its role in automated inspection and process automation.

Agriculture also benefits from vision-based monitoring systems for crop health monitoring, weed identification, disease detection, and automated harvesting, all of which support the shift toward precision agriculture [6].

As these examples show, the versatility and robustness of computer vision systems have made them indispensable across critical infrastructure, consumer services, and industrial automation. Among these domains, smart traffic management has emerged as a promising area, where CV enables automated traffic monitoring, vehicle classification, and dynamic response to road conditions. The following section discusses the application of computer vision in urban mobility and traffic systems.

1.3.2 Computer Vision for Traffic Management and Analysis

As urban population increases and transportation networks become increasingly sophisticated, older methods of traffic management like inductive loop detectors and manual counting are proving insufficient for real-time adaptive control. In this context, computer vision has developed as a revolutionary technology for smart traffic management, offering enhanced capabilities for monitoring, analysis, and optimization of traffic systems. In general, compared to conventional sensors and methods, computer vision has some clear advantages:

- *Non-Intrusive Deployment*: Cameras can be installed without significant alterations to existing infrastructure, reducing installation time and costs.
- *Rich Data Capture*: Visual data provides comprehensive information, including vehicle types and behaviors, which traditional sensors cannot discern.
- *Integration with AI and Machine Learning*: CV systems can leverage AI techniques such as deep learning to continuously improve their accuracy and adaptability in diverse environments. This integration allows the extraction of complex metrics like vehicle speed estimation and flow analysis, which require the recognition of patterns across frames and the application of temporal models.
- *Scalability and Flexibility*: Software-based systems can be updated and scaled with relative ease, accommodating evolving traffic patterns and technological advancements.
- *Automation*: CV enables automated traffic monitoring, minimizing the need for human intervention and enhancing efficiency.

Computer vision enables a range of applications that contribute to more efficient traffic monitoring:

- *Vehicle Detection and Classification*: CV models are trained to detect and categorize vehicles by type (e.g., motorcycles, cars, buses, trucks), facilitating more detailed traffic analysis and management strategies.
- *Speed Estimation*: CV systems can estimate vehicle speeds by tracking their movement across video frames, aiding in traffic conditions monitoring, speed regulation and enforcement efforts.
- *Traffic Flow Analysis and Congestion Detection*: By monitoring vehicle movements, CV systems can assess traffic density and flow patterns, as well as identify congestion hotspots, enabling dynamic adjustments to traffic signals or routing strategies and reducing congestion
- *Incident Detection*: Real-time analysis allows immediate identification of accidents or stalled vehicles, prompting quicker emergency responses and minimizing secondary incidents [7].

While computer vision offers significant benefits for traffic monitoring and control, it also faces several technical and ethical challenges that can impact its effectiveness. The main challenges are the following:

- *Lighting Conditions*: Changes in lighting, such as low light, glare, or heavy shadows, can interfere with the accuracy of object detection and tracking.
- *Weather Conditions*: Adverse weather conditions, including rain, fog, snow, or dust, can distort image quality and negatively impact the performance of computer vision algorithms.
- *Camera Calibration and Setup*: Proper calibration and setup of cameras is essential to ensure accurate positioning and tracking of objects in the field of view, as misalignment can lead to detection errors.
- *Occlusions*: Objects may be partially or fully occluded by other objects in the scene, making it difficult for the system to detect and track them accurately.
- *Accuracy*: CV systems are prone to errors, such as false positives or missed detections, especially in complex scenes.
- *Computational Resources*: Processing high-resolution video streams in real time requires considerable computing power.
- *Cost*: Setting up and maintaining a computer vision-based traffic monitoring infrastructure can be expensive, particularly for wide-area deployments.

- *Privacy Concerns*: Widespread use of cameras in public areas raises concerns about individual privacy and the potential misuse of surveillance footage. Consequently, obtaining special authorization from local authorities may be necessary before installing such systems [8].

1.4 Object Detection with YOLO Architecture

As discussed in the previous chapter, object detection is a fundamental task in computer vision, aiming to identify and track objects within images or video frames. Traditional object detection methods often involve multiple stages, which can be computationally intensive and slow. In contrast, YOLO (You Only Look Once), a popular object detection and image segmentation model, revolutionizes this process by performing object detection in a single pass through a neural network, enabling real-time detection with high accuracy. This efficiency makes YOLO particularly suitable for applications requiring immediate responses, such as traffic monitoring.

1.4.1 Object Detection Algorithms Categorization

Object detection algorithms are primarily categorized into two main types based on their processing approach: single-shot and two-shot detectors.

Single-shot Object Detection

Single-shot detectors like YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector) process the entire input image in a single pass through the neural network, making them computationally efficient. This approach enables real-time detection in resource-constrained environments as well, making it suitable for applications where speed is crucial. However, single-shot detectors may struggle with detecting small objects and can be less accurate compared to two-shot detectors.

Two-shot Object Detection

Two-shot detectors, like R-CNN (Region-based Convolutional Neural Networks) and its variants, use two passes of the input image to make predictions. In the first pass, a set of proposals or potential object locations are generated, while in the second pass, these proposals are refined, and the final predictions are made This approach generally achieves higher accuracy, especially for small object detection, but at the cost of increased computational complexity and slower processing speeds.

Overall, the choice between single-shot and two-shot object detection methods depends on the specific requirements of the application, balancing the trade-off between speed and accuracy [9].

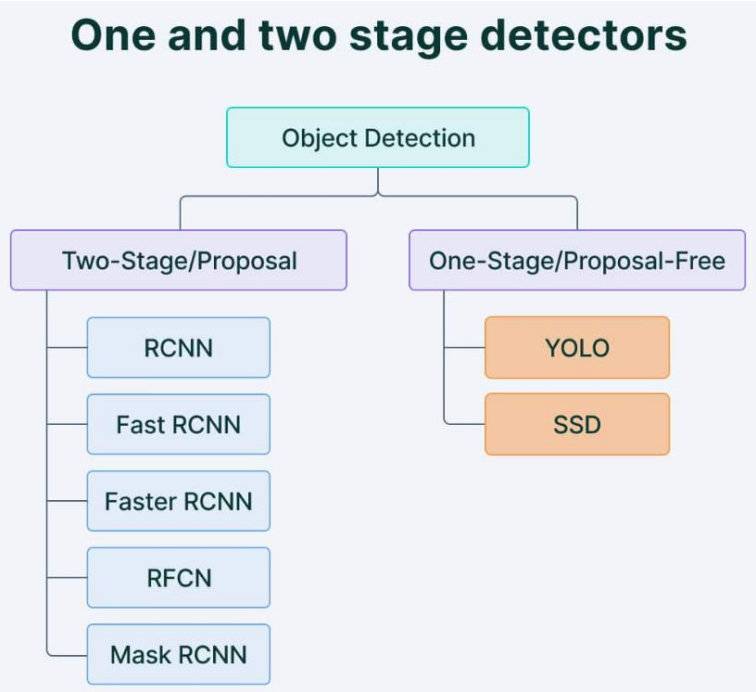


Figure 2: One-tage vs Two-Stage object detectors [9]

1.4.2 Object Detection Models Performance Evaluation Metrics

Evaluating the performance of object detection models is crucial for understanding their accuracy and effectiveness in identifying and localizing objects within images. Several key metrics are commonly used in this evaluation process, each providing unique insights into different aspects of model performance.

Precision

Measures the percentage of true positive predictions out of all the instances the model identified as positive, indicating how well the model avoids false positives [10].

Recall

In contrast to precision, recall represents the percentage of true positives found out of all actual positive instances, reflecting the model's effectiveness in detecting all relevant instances of a class [10].

F1 Score

The F1 Score is the harmonic mean of precision and recall, providing a single metric that balances both. It is particularly useful when the dataset has an uneven class distribution

or when both false positives and false negatives are critical to consider. A higher F1 Score indicates a better balance between precision and recall [10].

Intersection over Union (IoU)

Intersection over Union (IoU) is a standard metric used to evaluate the accuracy of object detection models by quantifying the overlap between the predicted bounding boxes and the ground truth bounding boxes. It serves as a fundamental indicator of how precisely a model can localize objects within an image.

To compute IoU, two areas are considered:

1. *Intersection*: This is the overlapping region between the predicted bounding box and the actual (ground truth) bounding box. It represents the area where the model's prediction correctly overlaps with the actual object.
2. *Union*: This is the total area covered by both the predicted and ground truth bounding boxes, combining both their unique and overlapping regions.

The IoU score is calculated using the formula depicted in Figure 3.

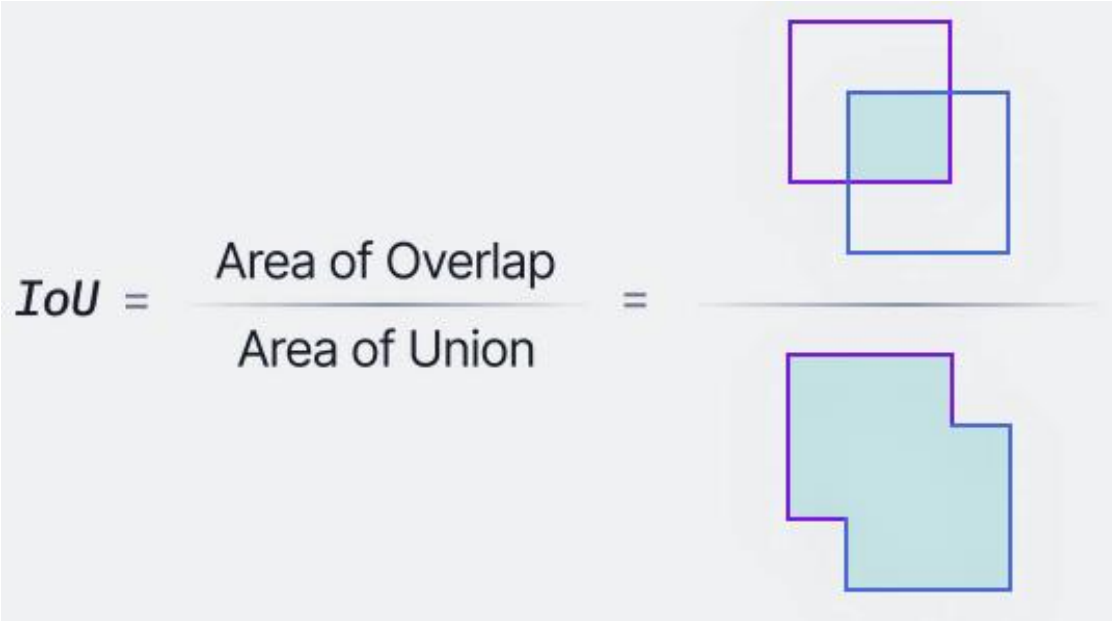


Figure 3: IoU calculation formula [9]

This results in a ratio that ranges from 0 to 1:

- An IoU of 0 means there is no overlap between the predicted and ground truth boxes.
- An IoU of 1 indicates a perfect overlap, meaning the model has predicted the object's position exactly right.

In practice, object detection systems use a threshold (commonly 0.5) to determine whether a prediction is considered a True Positive (TP):

- If IoU ≥ threshold, the detection is counted as correct (TP).
- If IoU < threshold, it's considered incorrect (False Positive or False Negative depending on the context).

Higher IoU values typically reflect better localization precision. However, setting a threshold too high (e.g., 0.9) might lead to very strict evaluations, where even minor misalignments penalize the model.

Essentially, IoU balances two critical goals in object detection, which are placing the box around the correct object and doing so with accurate boundaries. This makes it a valuable metric for training, evaluating, and comparing object detection models like YOLO [11].

Average Precision (AP)

Average Precision (AP) is a metric used to evaluate the performance of object detection models by summarizing the precision-recall curve into a single value. It quantifies the model's ability to balance precision (the accuracy of positive predictions) and recall (the ability to find all positive instances) across different confidence thresholds.

To compute AP, the following steps are followed:

- *Confidence Sorting*: For each object class, the predicted bounding boxes are sorted in descending order based on their confidence scores.
- *Precision-Recall Curve Generation*: Precision and recall values are computed at different confidence thresholds and plotted to form a curve. As the threshold is lowered, recall increases (capturing more true objects), but precision may decline due to the inclusion of more false positives.
- *Area Under Curve Calculation*: The AP for each class is calculated by measuring the area under the precision-recall curve. This area serves as a summary of the model's detection accuracy for that class over all recall levels.

AP values range from 0 to 1, with higher values indicating better model performance in terms of both precision and recall [12].

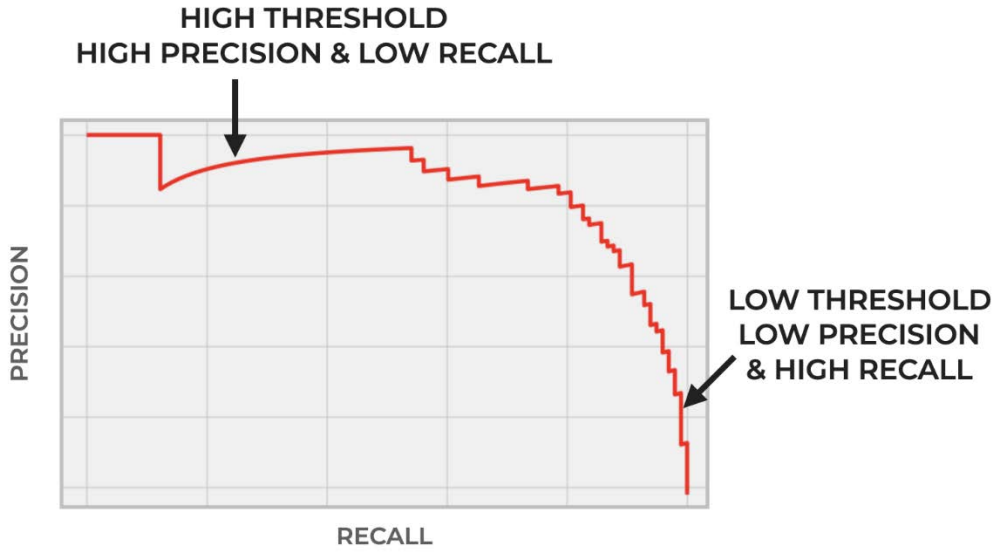


Figure 4: Precision-Recall Curve [13]

Mean Average Precision

Mean Average Precision (mAP) extends the concept of AP to evaluate the performance of object detection models across multiple classes. It provides a single metric that reflects the model's overall ability to detect and localize various objects.

To compute mAP, AP is first calculated individually for every class in the dataset using the procedure that was outlined above. Then, the mAP is obtained by taking the arithmetic mean of all class-specific AP values.

In some cases, like in COCO dataset evaluation, mAP is calculated not only across classes, but also across a range of Intersection over Union (IoU) thresholds, typically from 0.5 to 0.95 in 0.05 increments. This version of the metric, often written as mAP@.5:.95, offers a more comprehensive assessment by measuring how the model performs under varying degrees of localization strictness [12].

1.4.3 What is YOLO?

You Only Look Once (YOLO) is an open-source, real-time, single-shot object detection algorithm that revolutionized the field by introducing a unified, end-to-end approach. Unlike traditional methods that involve multiple stages, such as region proposal, feature extraction, and classification, YOLO treats object detection as a single regression problem, directly predicting bounding boxes and class probabilities from full images in one evaluation, using a single end-to-end convolutional neural network.

Since its introduction in 2015 by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in their famous research paper You Only Look Once: Unified, Real-Time Object Detection [14], YOLO has undergone several iterations, with each one enhancing object detection capabilities, computational efficiency, and adaptability to diverse computer vision tasks. This ongoing evolution underscores the rapid advancements in object detection technologies, with each version introducing novel features and expanding the range of supported tasks. Table 1 showcases the development and advancement of YOLO models over time [15].

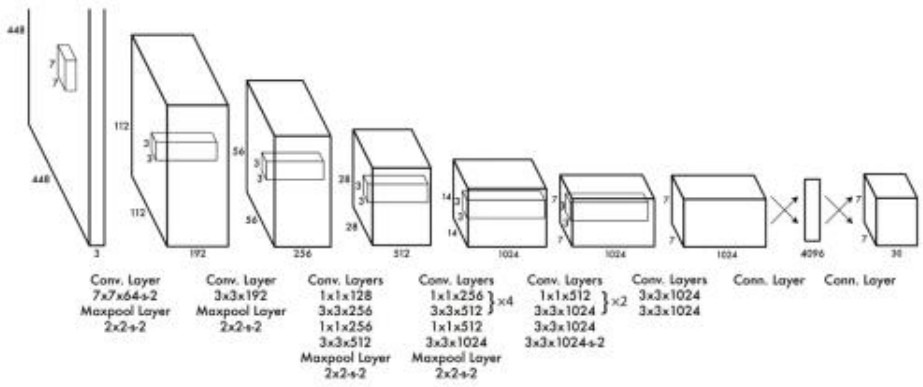
Table 1: Evolution of YOLO models [15]

Release	Year	Tasks	Contributions	Framework
YOLO	2015	Object Detection, Basic Classification	Single-stage object detector	Darknet
YOLOv2	2016	Object Detection, Improved Classification	Multi-scale training, dimension clustering	Darknet
YOLOv3	2018	Object Detection, Multi-scale Detection	SPP block, Darknet-53 backbone	Darknet
YOLOv4	2020	Object Detection, Basic Object Tracking	Mish activation, CSPDarknet-53 backbone	Darknet
YOLOv5	2020	Object Detection, Basic Instance Segmentation (via custom modifications)	Anchor-free detection, SWISH activation, PANet	PyTorch
YOLOv6	2022	Object Detection, Instance Segmentation	Self-attention, anchor-free OD	PyTorch
YOLOv7	2022	Object Detection, Object Tracking, Instance Segmentation	Transformers, E-ELAN reparameterisation	PyTorch
YOLOv8	2023	Object Detection, Instance Segmentation, Panoptic Segmentation, Keypoint Estimation	GANs, anchor-free detection	PyTorch
YOLOv9	2024	Object Detection, Instance Segmentation	PGI and GELAN	PyTorch
YOLOv10	2024	Object Detection	Consistent dual assignments for NMS-free training	PyTorch

The latest YOLO version, YOLO11, builds upon the advancements of the previous models showcased above. Before analyzing the specific architecture enhancements of YOLO11, it is essential to first understand the original architecture of YOLO, which forms the basis of how YOLO object detection really works.

1.4.4 YOLO Architecture

The architecture of the original YOLO model draws inspiration from GoogleNet and consists of a unified convolutional neural network that performs both object localization and classification in a single forward pass. This end-to-end approach enables real-time object detection at high speed and accuracy. The architecture of the CNN model that forms the backbone of YOLO, as presented in the original paper [14], is depicted in Figure 5.



The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

Figure 5: YOLO CNN model architecture [14]

At a high level, the YOLO model begins by resizing the input image to 448×448 pixels, which is then passed through a series of 24 convolutional layers and 4 max-pooling layers. These layers extract spatial features and patterns across the image. Following the convolutional blocks, two fully connected layers are used to perform the final prediction of object classes and bounding boxes. The majority of the network uses the ReLU activation function, while the final output layer applies a linear activation to support the regression of bounding box coordinates and confidence scores. [9]

To enhance training stability and generalization, batch normalization is employed throughout the network, and dropout is used in the fully connected layers to reduce overfitting. Additionally, the initial 20 convolutional layers are often pre-trained on ImageNet to leverage transfer learning. These pre-trained layers are then fine-tuned for object detection by replacing the classification head with additional detection layers [16].

YOLO object detection algorithm is based on the following concepts:

1. Residual Blocks

YOLO divides the input image into an S × S grid, and each cell in this grid is tasked with detecting objects whose centers fall within it. Each cell predicts a fixed number of B bounding boxes, as well as class probabilities and confidence scores indicating both the presence of an object and the accuracy of the predicted box. This localized prediction mechanism allows YOLO to process images efficiently without the need for region proposals.

2. Bounding Box Regression

Each predicted bounding box is represented by a vector containing:

$$Y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, \dots, c_n]$$

- p_c: confidence score (i.e., probability that an object is present in the cell)
- (b_x, b_y): normalized coordinates of the bounding box center relative to the grid cell
- (b_h, b_w): width and height of the bounding box relative to the image dimensions
- $(c_1, c_2, \dots, c_n)$: probabilities for each of the object classes

3. Intersection over Union (IoU)

During both training and inference, YOLO evaluates how closely predicted boxes match the ground truth using Intersection over Union (IoU). This metric helps filter out inaccurate predictions. Only bounding boxes with an IoU greater than a predefined threshold (e.g., 0.5) are retained.

4. Non-Maximum Suppression (NMS)

Multiple bounding boxes with high confidence scores may often overlap the same object. Non-Maximum Suppression (NMS) is applied as a post-processing step to eliminate redundant predictions. NMS retains only the bounding box with the highest confidence score among overlapping boxes and suppresses the others, ensuring that each object is detected only once [9, 16].

1.4.5 YOLO11 Architectural Enhancements

The release of YOLO11 (or YOLOv11) by Ultralytics, the same team behind YOLOv8, marked another significant leap forward in the evolution of real-time object detection models. YOLO11 models can execute the whole range of computer vision tasks, such as object detection, image segmentation and classification, oriented bounding box (OBB) detection, and keypoint detection [17].

YOLO11 is built on top of the YOLOv8 codebase, preserving its modular design but introducing targeted architectural enhancements aimed at improving both speed and accuracy. These enhancements make YOLO11 more efficient, with lighter model variants that can run on edge devices or resource-constrained environments without compromising performance [18].

Ultralytics released five YOLO11 models, which vary in complexity, model size, and computational resources required. Some of their performance metrics are presented in Table 2.

- *YOLO11n (Nano)*: optimized for minimal resources and fast inference.
- *YOLO11s (Small)*: slightly more accurate than Nano while remaining lightweight.
- *YOLO11m (Medium)*: a general-purpose version suitable for most standard tasks.
- *YOLO11l (Large)*: offers greater accuracy for use cases where higher precision is required.
- *YOLO11x (Extra Large)*: the most powerful model, maximizing accuracy and detection performance.

Table 2: YOLO11 models metrics [17]

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

The YOLO11x model demonstrates the top-tier performance of this generation. According to Ultralytics benchmarks, it achieves an mAP of 54.7% on the MS COCO dataset, while the smallest model, YOLO11n, reaches 39.5% mAP [17].

At its core, the YOLO architecture consists of three fundamental components, the backbone, the neck, and the head. Each of these components has been refined with new blocks and attention mechanisms. Particularly, the architectural enhancements of YOLO11 revolve around the C3K2 block, the SPFF module, and the C2PSA block, all designed to improve its spatial information processing capability while ensuring high-speed inference [15].

Backbone

The backbone serves as the primary feature extractor, converting input images into a hierarchy of feature maps at multiple scales through successive stages of convolutional operations. YOLO11 preserves the traditional structure of stacked convolutional layers for progressive downsampling and feature extraction, but replaces the previously used C2f block with a new C3k2 block.

The C3k2 is a custom implementation of the Cross Stage Partial (CSP) bottleneck, characterized by the use of two smaller convolutions (kernel size 2) instead of one large convolution. This change enhances processing speed while maintaining performance.

Another enhancement in the backbone is the introduction of a new Cross Stage Partial with Spatial Attention (C2PSA) block after the Spatial Pyramid Pooling - Fast (SPPF) block, which is retained from previous versions. The C2PSA block enhances spatial attention in the feature maps, allowing the model to focus more effectively on important spatial regions within an image. This in turn, benefits the detection accuracy for objects of varying sizes or occluded ones [15].

Neck

The neck component combines features from various resolutions, allowing the model to better capture multi-scale information. This is typically achieved through operations such as upsampling and concatenation, which integrate feature representations extracted at different stages of the backbone.

Similarly to the backbone, the C2f block in the neck is replaced by the more efficient C3k2 block. By incorporating the C3k2 block after the upsampling and concatenation steps, speed and overall computational performance are enhanced.

Another architectural enhancement of YOLO11 is its emphasis on spatial attention, which is introduced through the C2PSA module. This attention mechanism allows the model to focus on key spatial regions in an image, improving its ability to detect objects more accurately, especially when dealing with small or partially occluded objects. The addition of the C2PSA module is a distinguishing feature of YOLOv11, as this is lacking in YOLOv8 [15].

Head

The head of YOLOv11 handles the final stage of the object detection pipeline, as it converts the processed feature maps received from the neck into output predictions. This

component outputs bounding boxes and class labels that detect and classify the objects within the input image.

As in the backbone and the neck, the head also utilizes C3k2 blocks to process and refine feature maps with high efficiency. These blocks are distributed across different pathways in the head, enabling the model to handle multi-scale features at various depths. The "behavior" of each C3k2 block is controlled by the c3k parameter, which determines its internal structure:

- When c3k = False, the C3k2 module operates similarly to the C2f block, following a standard bottleneck architecture.
- When c3k = True, the bottleneck structure is replaced with a C3 module, allowing for deeper and more complex feature extraction.

In addition to C3k2, YOLOv11 also introduces the C3k block, which brings further flexibility by supporting customizable kernel sizes. This flexibility enhances the model's ability to extract more detailed features from images, leading to better object detection accuracy.

After the C3k2 blocks of the head, several CBS (Convolution-BatchNorm-SiLU) layers are included. These layers further enhance the feature maps by extracting relevant features needed for accurate object detection, stabilizing and normalizing the data flow within the network using batch normalization and applying SiLU (Sigmoid Linear Unit) activation to introduce non-linearity, thereby improving the model's ability to learn complex patterns. These CBS blocks act as essential building blocks in the model's detection pipeline, ensuring that well-refined feature maps are forwarded to the subsequent layers, which are responsible for predicting bounding boxes and class labels.

At the end of each detection branch, a series of Conv2D layers is used to reduce the features to the precise number of outputs required for bounding box coordinates and class predictions. These outputs are then passed to the final Detect layer, which aggregates all prediction results and provides:

- Bounding box coordinates used to localize objects within the image.
- Objectness scores that reflect the model's confidence in the presence of an object.
- Class scores that determine the class of each detected object [15].

Figure 6 illustrates the architecture diagram of YOLO11.

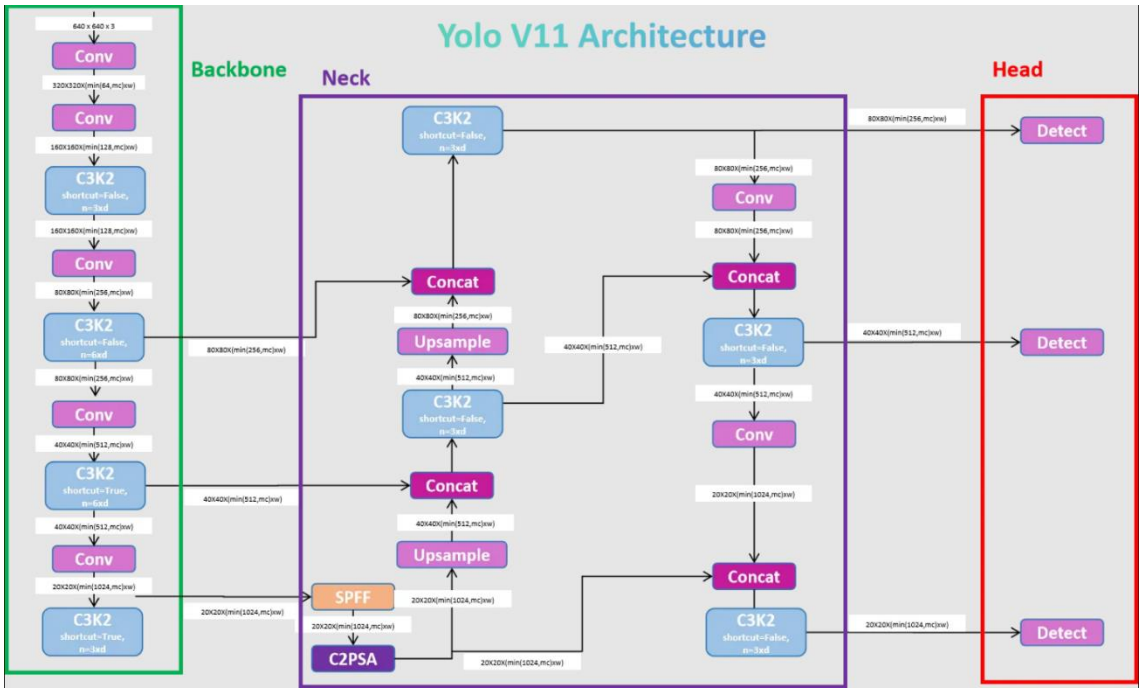


Figure 6: YOLO11 Architecture [19]

1.4.6 COCO Dataset

COCO (Common Objects in Context) dataset is one of the most prominent and widely used benchmark datasets in computer vision. Designed to support developers and researchers in object detection, instance segmentation, and captioning tasks among others, COCO provides a large collection of annotated images of a variety of diverse objects [20].

COCO contains 330,000 images, out of which approximately 200,000 are annotated for detection, segmentation, and captioning tasks. It comprises of 80 object classes (presented in Figure 7), ranging from common objects such as people and different types of vehicles (cars, trucks etc.), to less frequent or more specific objects. Each annotated image includes bounding boxes for object localization, segmentation masks, and up to 5 natural language captions that describe the scene. COCO dataset classes are further grouped into two categories, which are "things" and "stuff". "Things" are objects that are countable and have clearly defined shapes (e.g., person, car), while "stuff" are amorphous background elements or materials (e.g., sky, road) [21].

person	fire hydrant	elephant	skis	wine glass	broccoli	dining table	toaster
bicycle	stop sign	bear	snowboard	cup	carrot	toilet	sink
car	parking meter	zebra	sports ball	fork	hot dog	tv	refrigerator
motorcycle	bench	giraffe	kite	knife	pizza	laptop	book
airplane	bird	backpack	baseball bat	spoon	donut	mouse	clock
bus	cat	umbrella	baseball glove	bowl	cake	remote	vase
train	dog	handbag	skateboard	banana	chair	keyboard	scissors
truck	horse	tie	surfboard	apple	couch	cell phone	teddy bear
boat	sheep	suitcase	tennis racket	sandwich	potted plant	microwave	hair drier
traffic light	cow	frisbee	bottle	orange	bed	oven	toothbrush

Figure 7: COCO dataset class list [21]

Despite the wide spectrum of object types, COCO dataset is subject to inherent bias due to class imbalance (as shown in Figure 8), which is a common issue where certain object classes are significantly overrepresented compared to others. This imbalance can lead to bias in model training, leading to overfitting on dominant classes and underperformance on less frequent ones. To mitigate these effects, techniques like oversampling, undersampling, and synthetic data generation are often applied during training [21].

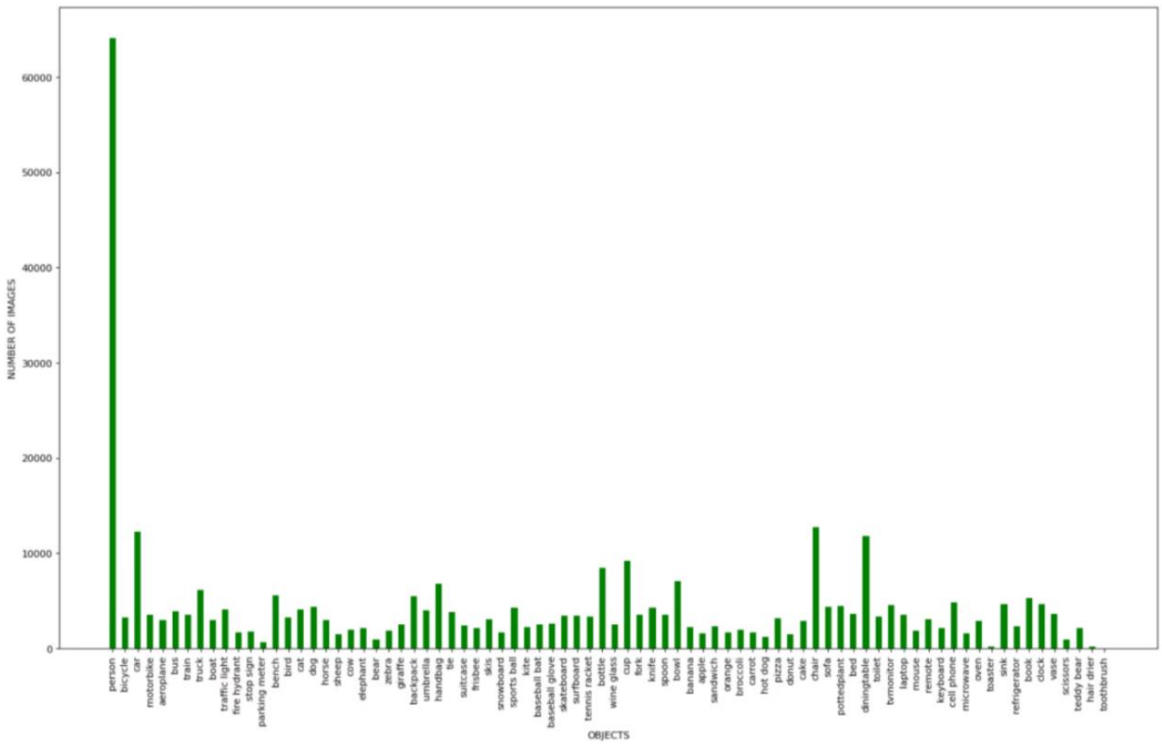


Figure 8: COCO dataset class imbalance [21]

Regarding its structure, the COCO dataset is split into three subsets:

- *Train2017*: consists of ~118,000 images used for training models.
- *Val2017*: contains ~5,000 images for validation during model training.

- *Test2017*: contains ~20,000 images used for benchmarking. The ground truth annotations for this subset are not publicly available and performance is evaluated via submission to the COCO evaluation server [20].

In terms of its benchmarks, COCO provides standardized evaluation metrics that enable precise comparison between models. For object detection, the most widely used metric is mean Average Precision (mAP), while for segmentation tasks, mean Average Recall (mAR) is used. These metrics assess whether a model can detect objects and how precisely it can localize them and distinguish between different classes under varying levels of spatial overlap. For that reason, COCO dataset serves as a preferred benchmark for evaluating the detection accuracy and generalization performance of single-shot detectors like YOLO [20].

1.5 Problem Statement and Application Scope

The widespread adoption of IoT devices, combined with advancements in computer vision have paved the way for an increase in the deployed number of intelligent traffic monitoring systems. However, majority of existing computer vision applications related to vehicular mobility remain limited in scope, as they are often confined to showcasing object detection on pre-recorded video streams. These applications lack the capability for integration into real-world, end-to-end solutions and are rarely deployed in operational environments. Moreover, these approaches often require manual intervention, as they are not designed for continuous, automated operation.

In response to these limitations, this thesis explores the development of a fully automated and headless traffic monitoring system that integrates computer vision into a broader, functional application framework. The system is designed to operate in near real-time, processing raw video footage captured from a remote location via an IoT edge device and transferring it to a physically separate, low-cost processing unit without GPU acceleration. This architectural choice addresses practical deployment scenarios where hardware resources are constrained and cloud-based processing may be impractical, latency-sensitive, or expensive.

In contrast to many existing vehicle object detection demonstrations, the proposed system is intended to operate continuously and autonomously as part of a real-world traffic monitoring pipeline. It detects and classifies vehicles using pre-trained YOLO11

model and extracts traffic metrics, such as vehicle speed and congestion indicators, integrating them into an edge-to-cloud processing workflow that requires no human intervention.

The objective of this work is to bridge the gap between proof-of-concept models and real-world deployable traffic monitoring systems, showcasing how lightweight computer vision architectures, when combined with appropriate cloud services, automated file transfer pipelines, and edge devices, can support scalable, low-maintenance mobility analytics in the field.

2 System Architecture

This chapter presents the high-level design of the traffic monitoring system developed in this thesis. High-level design is the stage of system development where the overall structure is outlined, focusing on how the major components interact and function together, rather than the internal details of their implementation [22]. It offers a conceptual view of the solution and acts as a link between the theoretical problem statement outlined in the previous chapter and the practical deployment.

The proposed system, which has been deployed and tested, is a headless, automated, near real-time traffic monitoring application designed to process raw traffic videos, which are captured by an IoT Edge device and then relayed to a geographically separate, low-cost processing unit, without requiring manual oversight or powerful GPU hardware. By incorporating computer vision, cloud storage, automated data pipelines, and lightweight analytics, this system offers a scalable and cost-effective alternative to traditional traffic monitoring solutions.

This chapter aims to:

- Define the objectives and explain the rationale behind key architectural choices, such as the use of an event-driven, edge-to-cloud architecture and automation pipelines.
- Briefly touch on security-related decisions to provide context for data integrity and access control.
- Provide a high-level overview of the system's operation, based on the architecture diagram, without delving into the detailed logic of each individual component.

2.1 System Design Objectives and Constraints

The design of the proposed traffic monitoring system was guided by a set of clearly defined objectives and real-world deployment constraints. The primary objective was to build and deploy a real, operational application, rather than a plain proof-of-concept. This goal set the foundation for a system that demonstrates technical feasibility, addresses practical deployment challenges, and operates long-term in realistic conditions.

The most fundamental goal was to develop a fully automated, headless system capable of operating continuously without manual intervention. This requirement reflects the need for a monitoring solution that can operate reliably over extended periods without user oversight, making it suitable for use in unattended environments. The automation extends across all stages of the data pipeline, from video capture and file transfer to processing, analytics extraction, and storage.

Another objective was to enable near real-time processing of traffic footage captured from remote locations. Unlike most proof-of-concept computer vision applications that rely on pre-recorded video files, the proposed system is designed to process newly captured video input immediately upon availability.

An additional priority was the selection of low-cost and accessible hardware. A Raspberry Pi 4 Model B embedded device was deployed on the edge, because it balances affordability, compact form factor, and ease of deployment, while offering sufficient performance for capturing and transferring video data. Furthermore, its suitability for remote access makes it ideal for unattended installations. Moreover, the Raspberry Pi supports automatic reboot after power outages, increasing its resilience in environments with unreliable electricity. In contrast, alternatives like the NVIDIA Jetson series embedded devices, though more powerful and capable of GPU-accelerated inference, were ruled out due to their significantly higher cost, larger size, and more complex installation and maintenance requirements, which make them less practical for this use case.

From an operational standpoint, the system needed to accommodate network constraints, particularly limited bandwidth availability and intermittent connectivity at the edge location.

On the processing side, because of the limited or cost-prohibitive access to high-end computer infrastructure in real-world settings, the system was designed to operate on a general-purpose PC without dedicated GPU acceleration. Since the processing unit is not dedicated exclusively to this application, the solution needed to be lightweight and non-disruptive to other concurrent tasks running on the same machine, that is using only CPU resources.

In addition to hardware-related constraints, cost-effectiveness was an important consideration for the selection of software solutions and cloud services. In this context, the

system was developed using tools that do not require paid licenses or recurring subscription fees. This approach ensures that the solution will remain financially viable for long-term deployment without incurring operational expenses.

Finally, privacy and data protection were taken into-account. Since the recorded traffic videos may include sensitive visual data, such as identifiable vehicles or movement patterns, they shall not be made publicly accessible. Therefore, file transfers and storage are handled in a way that maintains confidentiality and limits exposure.

2.2 Core Architectural Approaches

Based on the system's design objectives and constraints, specific architectural approaches were adopted for the development of the proposed system. Those were event-driven automation and Edge-to-Cloud architecture models, providing the foundation for the system's overall architecture. These approaches were selected to ensure a responsive, scalable, and efficient system that meets the defined requirements.

2.2.1 Edge-to-Cloud Architecture

The developed traffic monitoring system adopts a lightweight edge-to-cloud architecture model, which is a distributed computing paradigm that separates data acquisition at the network edge from centralized processing and storage. In general, edge-to-cloud computing refers to systems where edge devices, such as embedded devices, sensors, or IoT endpoints collect data near its source, and selectively transmit it to a remote location or cloud service for further analysis, coordination, or long-term storage [23].

This architectural model is suitable for IoT-based applications that require both real-time responsiveness and broader coordination across multiple distributed endpoints. By distributing computing tasks between edge and cloud, the system can reduce latency, improve scalability, and optimize resource usage. In most edge-to-cloud implementations, edge devices handle operations that require low latency, such as data capture or real-time event triggering, while cloud infrastructure supports more computationally intensive tasks, such as batch analytics, long-term storage, model training, and data processing [24].

In this system, however, the edge node is limited to capturing and uploading raw video data to the cloud, while deferring all computationally intensive tasks to a physically separate, centralized processing unit. This configuration still adheres to edge-to-cloud principles by keeping data collection and data processing tasks separately, both in terms of

physical location and their functional roles within the system. Specifically, data is acquired at the edge, stored in cloud storage, and processed at a remote location. In this scenario, cloud serves as an asynchronous bridge between the distributed edge and centralized processing. A visual representation of the edge-to-cloud architecture in the way it is implemented in this system is provided in Figure 9.

Furthermore, this geographical and functional decoupling contributes to the system's overall robustness and scalability. Scalability is enhanced, as additional edge nodes can be added without demanding major changes to the processing pipeline. Robustness is also improved, as potential points of failure can be isolated. If the edge device loses connectivity or power, it can resume uploading once restored, without affecting the centralized processing logic. Moreover, privacy concerns are addressed more effectively by ensuring that video data remains securely stored in private cloud storage, with controlled access and no public exposure [24].

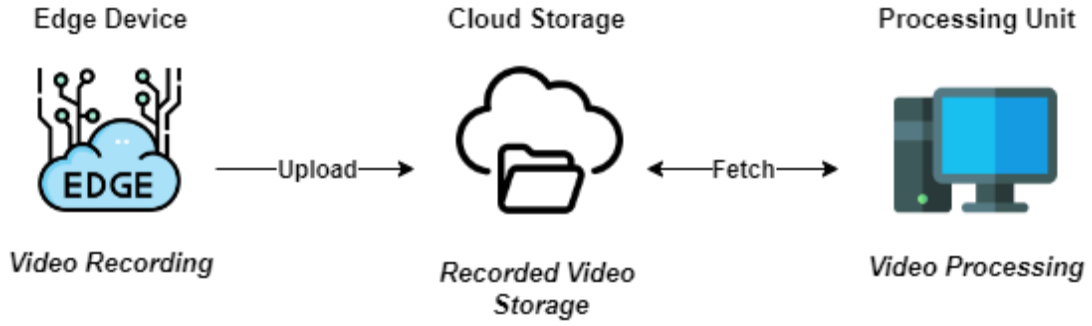


Figure 9: System's Edge-to-Cloud architecture implementation

2.2.2 Event-Driven Architecture

The developed traffic monitoring system adopts the design principles of Event-Driven Architecture (EDA), which is a software design paradigm that facilitates asynchronous, loosely coupled communication between system components through the generation and handling of events, making it an ideal option for dynamic and distributed environments such as IoT-based systems. In EDA, an event represents any significant change in state (e.g. detection of new data input), which then serves as a trigger for subsequent processes [25].

Event-driven architecture was favored over traditional polling methods due to its efficiency in detecting and responding to changes. Polling requires components to periodically check for updates, which introduces latency, increases bandwidth usage, and consumes unnecessary computational resources, especially when changes are infrequent. In

contrast, event-driven systems respond only when a relevant event occurs. In the case of the proposed system, the processing unit doesn't continuously query the cloud storage for newly uploaded video files. Instead, it receives a notification via a webhook when a new file becomes available. This push-based mechanism reduces idle checks, minimizes delay, and contributes to a more scalable and responsive architecture [26].

In the context of this system, the IoT Edge device, responsible for capturing video footage and uploading it to cloud storage, acts as an event producer. The successful upload of a video file triggers a push notification that is received by a webhook listener running on the central processing unit, which serves as an event consumer. Upon receiving this notification, a sequence of automated tasks is initiated —specifically: downloading the video file, running the computer vision processing pipeline, and storing the extracted results in a database. This setup aligns with the publish-subscribe communication model central to event-driven architecture, wherein producers emit events without knowledge of their consumers, and consumers subscribe only to specific event types relevant to their functionality [25].

A key advantage of this architectural approach lies in its asynchronous communication model and decoupled structure, as it allows components to operate independently of one another's execution timelines. Event producers generate notifications without requiring knowledge of the consumers' state or availability, while the event consumers respond to those events on their own schedule, executing the associated processing tasks without needing to query or coordinate directly with the producer. This loose coupling ensures that each component remains modular and self-contained, making the system easier to maintain, test, and scale. It also enhances fault tolerance, as temporary failures in one part of the system, such as a processing node going offline, do not impact the rest, allowing the system to resume normal operation once the issue is resolved. Furthermore, scalability is improved, as additional components can be integrated without disrupting the overall workflow. These characteristics are valuable in distributed IoT systems like the one implemented here, where flexibility, fault tolerance, and scalability are critical [27].

Building on this decoupled design, event-driven architecture also enhances real-time responsiveness and operational flexibility. As the system reacts to events precisely when they occur, it can initiate processing without delay or resource-intensive polling. This near real-time response is especially useful in time-sensitive applications like traffic monitoring, where delays in data processing can reduce the relevance and accuracy of results.

The system can also handle events that happen at irregular intervals, resuming its efficient operation without wasting resources [27].

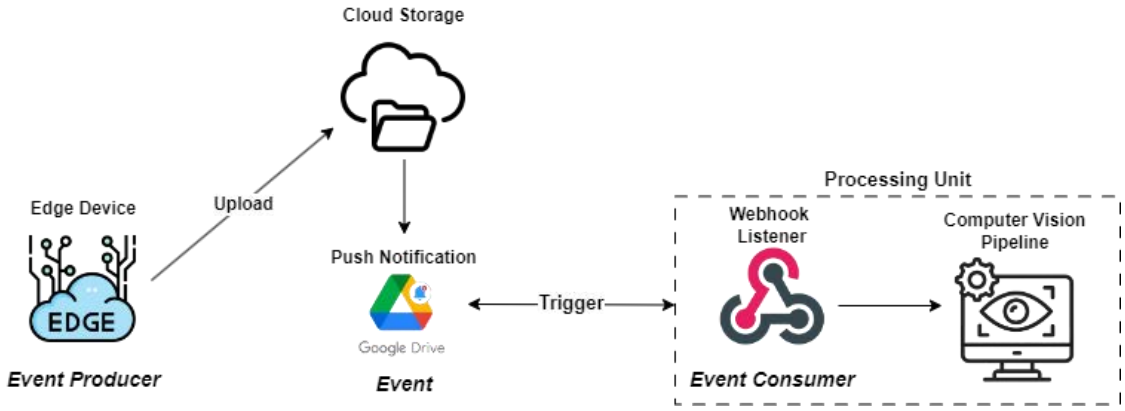


Figure 10: System's Event-Driven architecture implementation

2.3 End-to-End System Architecture

The developed traffic monitoring system is structured as a lightweight, edge-to-cloud, event-driven architecture that operates across spatially and functionally decoupled components. This section provides a high-level overview of the end-to-end architecture, outlining the core components, their responsibilities, and how they interact to form a robust and fully automated traffic analytics pipeline.

An IoT edge device, comprising of a Raspberry Pi 4 Model B equipped with a connected webcam, has been deployed in a remote location, overlooking a road with two lanes per direction. This device autonomously captures traffic video segments at predefined intervals. Each video is saved locally on the edge device with a timestamped filename to ensure traceability. Upon recording completion, the video file is uploaded to a specific folder in Google Drive. The Raspberry Pi also implements local and cloud file management routines, automatically deleting older video files and maintaining only a defined number of latest recordings, to optimize storage usage.

The upload process is managed through the Google Drive API using a Python client that authenticates via OAuth 2.0. This mechanism enables the edge device to gain authorized access to the end user's Google Drive account through a one-time interactive login and token exchange. Once authenticated, access and refresh tokens are securely stored on the device, without requiring repeated user intervention. This approach enables the system to perform uploads securely and autonomously while adhering to the permissions

granted by the user during the OAuth consent process, as defined by the associated access scopes.

The edge device does not perform any form of local inference or video analytics, which reduces its computational burden and energy consumption significantly. This design choice ensures that the device remains lightweight and cost-effective, while still playing a critical role by capturing and uploading raw data that is used for centralized processing.

In the context of this system, Google Drive is utilized not just as a storage backend, but also as a transfer medium between the edge device and the processing unit. Its role is to temporarily host uploaded video recordings and provide integration points via Google APIs to trigger the automated centralized processing pipeline.

The Google Driver folder where video files are uploaded is monitored using Google Drive's "Push Notifications" feature, which allows the system to detect changes, such as the addition of a new video file. This is achieved by registering a webhook endpoint that listens for file creation events. When a new video file is uploaded in the monitored folder, Google Drive sends a notification request to the webhook URL, thereby triggering the centralized processing pipeline. This event-driven approach minimizes unnecessary network and computational load by eliminating the need for constant polling, while enabling the system to respond to file uploads in near real-time.

To enable near real-time response to new video file uploads, the system employs a publicly accessible webhook listener that is hosted on the processing unit. Since the processing unit operates behind a private network firewall and does not have a public IP address, ngrok, a secure tunneling service is used to expose the local webhook to the internet. This setup allows external services, such as Google Drive, to reach the webhook endpoint reliably.

Once a new video file is uploaded to Google Drive and a notification is sent to this public webhook URL, the tunneling service receives the notification and forwards it to the webhook listener running locally on the processing unit. The webhook listener then processes the incoming message and extracts metadata from the notification, such as the file's unique ID and name. This information is used to identify and retrieve the correct file for further processing.

The webhook listener responsible for receiving and responding to file upload notifications also requires secure access to the Google Drive API in order to fetch the uploaded

files. Unlike the edge device, which authenticates through an OAuth 2.0 flow tied to a user account, the webhook listener uses a Google Service Account to interact with Google Drive in a headless, server-to-server manner.

The use of a service account eliminates the need for any manual login or token refreshes during execution, making it suitable for long-running, automated, headless backend services. It also enables fine-grained access control over access scopes and folder permissions, as only the target folder is shared with the service account. This setup ensures that the webhook listener can autonomously retrieve uploaded videos, while maintaining strict access control and separation of roles between components.

The download of the latest uploaded video file is handled via the Google Drive API. Once the video is downloaded into a local directory, the webhook listener automatically triggers the computer vision processing pipeline, executed on the central processing unit, which is general-purpose CPU-only machine without GPU acceleration.

The downloaded video is processed frame-by-frame using a YOLO-based model for vehicle detection, classification, and tracking, as well as for the extraction of additional traffic metrics such as estimated speed and vehicle count per time interval. These metrics are aggregated into a structured JSON object. Once inference is complete, the extracted metrics, encapsulated in the JSON, are published through two parallel channels: to an InfluxDB time-series database for structured storage and historical querying, and to an MQTT broker, secured with TLS encryption, (i.e. MQTTs), for real-time distribution to external subscribers.

The interaction between the processing script and the database is not limited to writing new data. The system also fetches previously stored metric values, particularly vehicle count historic data, to compute dynamic indicators such as traffic flow (vehicles per hour) and to detect the presence of traffic congestion.

Overall, automation is a critical enabler of the system's fully headless operation. On the edge device, scheduled video recordings are orchestrated using CRON jobs, enabling consistent footage capture at predefined intervals without manual intervention. To ensure continuous availability of the webhook tunnel, batch scripts monitor internet connectivity and automatically restart ngrok if found inactive. This is useful in scenarios involving network outages or system reboots.

In addition, retry mechanisms and fallback routines are used across the various system components, addressing common issues such as expired authentication credentials, interrupted file uploads, and database outages. These features ensure that disruptions are handled locally, without causing cascading failures across the system.

The system's decoupled architecture offers it a high degree of fault tolerance. As a result, the system consistently returns to its operational state with minimal human intervention, making the long-term, unattended deployment in real-world environments feasible and sustainable.

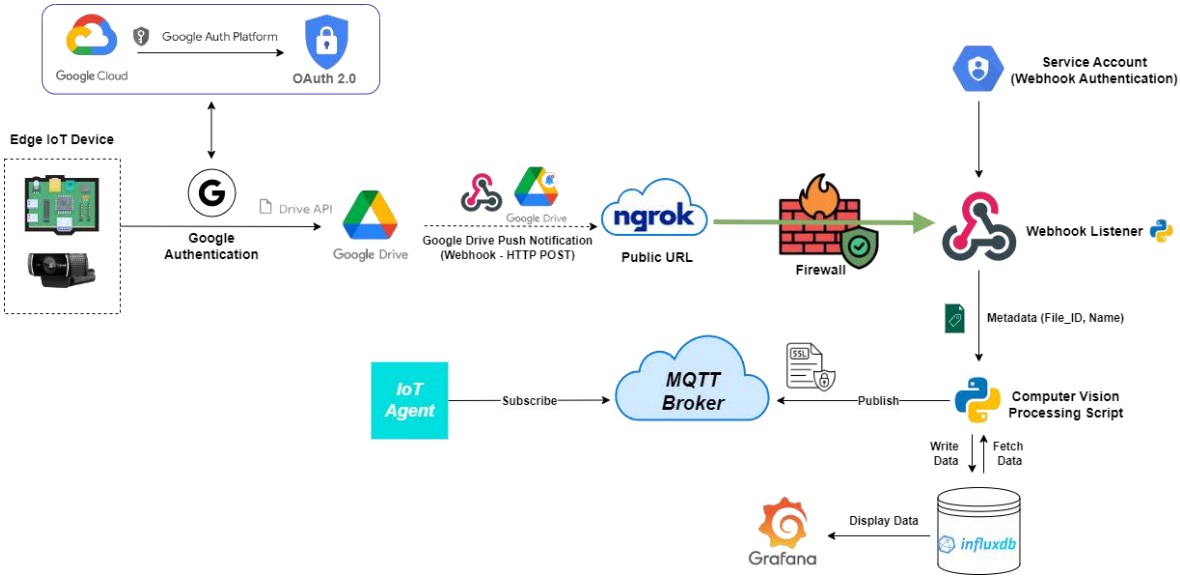


Figure 11: System architecture diagram

Further technical analysis and implementation details of individual system components, such as the edge device functionality, the webhook mechanism, and the computer vision processing pipeline, will be provided in the subsequent chapters.

3 Edge Device Functionality

This chapter provides a technical analysis of the edge device component within the traffic monitoring system. It outlines the architecture and functionality of the edge device, focusing on the hardware, software, and design considerations that enable autonomous operation under real-world conditions. It highlights the technologies used, the automation mechanisms implemented, and the trade-offs made to achieve efficient, secure, cost-effective, and low-maintenance data acquisition and transfer within the system's edge-to-cloud architectural approach.

3.1 Edge Device Overview and Design Considerations

The system's edge component is built around a lightweight, autonomous, and cost-effective setup centered on a Raspberry Pi 4 Model B. Installed at a remote roadside location, the device serves as a dedicated unit for periodic video recording, data pre-processing, and secure cloud-based transfer. It captures raw traffic footage at predefined intervals and uploads it to Google Drive, which acts as cloud storage.

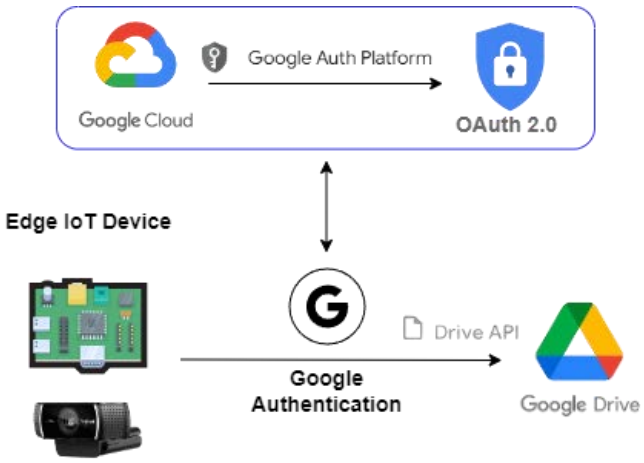


Figure 12: Edge device functionality overview

The device operates unattended in a public building, relying on stable power supply and wireless network connectivity. Its physical deployment in a remote, unattended location significantly influenced the design choices outlined in the following sections, particularly in ensuring autonomous operation and minimal maintenance requirements.

3.1.1 Hardware Platform

The Raspberry Pi 4 Model B was selected as the core computational platform due to its balance of performance, affordability, and energy efficiency. Featuring a quad-core ARM Cortex-A72 (ARM v8) processor and up to 8GB of LPDDR4 RAM, it provides sufficient computational resources to handle video capture tasks and manage file operations without overheating or requiring additional hardware support. Its compact form factor, passive cooling capability, operating temperature range of 0–50ºC, and support for multiple I/O interfaces, such as USB 3.0, USB 2.0, HDMI, and a 40-pin GPIO header, make it suitable for embedded deployments. The board also includes dual-band 802.11ac Wi-Fi, Bluetooth 5.0, and a Gigabit Ethernet port, enabling flexible connectivity depending on deployment conditions. It supports booting from microSD card, while power is supplied via USB-C at 5V/3A, which is adequate to power connected peripherals such as cameras or external drives. These hardware features, along with the device's automatic reboot capability following a power outage, render the Raspberry Pi 4 an ideal choice for field-based, unattended edge computing use cases [28].

A USB Logitech C922 Pro HD webcam serves as the video capture device, configured to capture Full HD (1080p) videos at a specific frame rate [29].

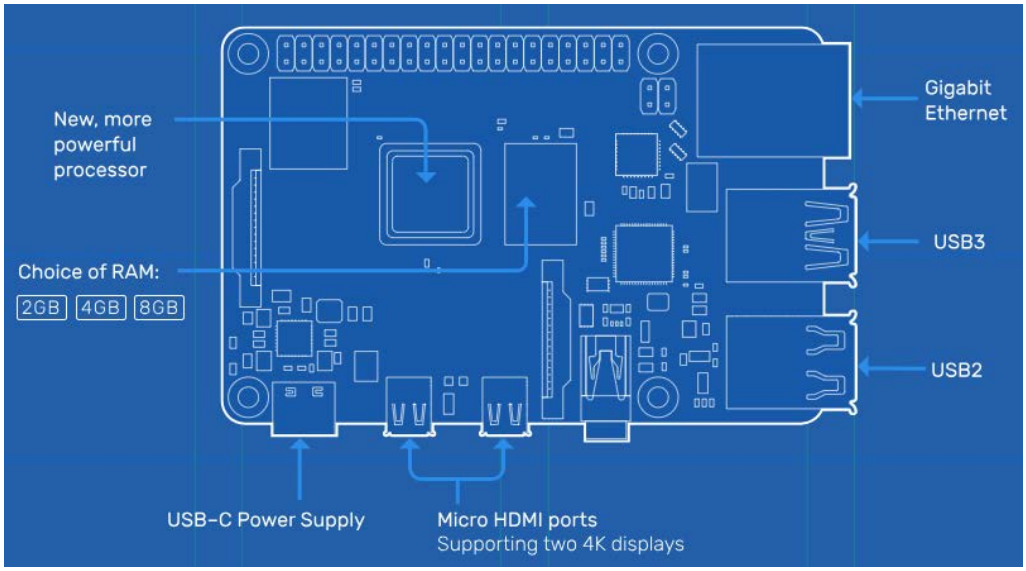


Figure 13: Raspberry Pi 4 Model B tech specs [28]

The camera and Raspberry Pi form an IoT system, where the camera acts as a sensor and the Raspberry Pi as a gateway.

3.1.2 Design Considerations

A key architectural decision was to offload all inference and analytics processing from the edge device to a centralized processing pipeline. This design choice keeps the edge node lightweight in terms of both software and hardware requirements, thereby reducing computational load, lowering energy consumption, and minimizing potential points of failure. Rather than implementing live video streaming and performing real-time analysis, the edge device solely focuses on capturing and uploading discrete video segments to cloud storage at regular predefined intervals. Live stream processing was deliberately avoided, not only due to its higher computational and bandwidth demands, but also because it would complicate the extraction of structured outputs, which is the primary objective of the traffic monitoring system. This approach simplifies both implementation and maintenance, while still fulfilling the system's data collection objective.

Another important consideration was ensuring long-term, unattended operation. One potential challenge to this is the risk of storage exhaustion on both the Raspberry Pi and the Google Drive folder that stores the uploaded videos. To address this, the edge device incorporates automated file management mechanisms. Through two dedicated functions within the `*camera_recording.py*` script, only a defined number of the most recent video recordings are retained, while older files are automatically deleted. These automated cleanup routines help maintain storage availability and ensure uninterrupted operation without requiring manual intervention.

Given its deployment in a remote location, the ability to manage and troubleshoot the device remotely was also essential. To this end, Raspberry Pi Connect, a secure remote access service provided by the Raspberry Pi Foundation, is utilized. This service facilitates a peer-to-peer connection between a Raspberry Pi and another device, such as a laptop or desktop PC, via a web browser. It enables authorized users to connect to their Raspberry Pi device over the internet without requiring static IP addresses, port forwarding, or VPNs, all of which are often impractical or insecure in edge deployments.

Operators can access the device's desktop environment through a secure, browser-based screen sharing interface, allowing full graphical interaction with the system. In addition to screen sharing, Raspberry Pi Connect also enables secure shell (SSH) access, providing a command-line interface for advanced control, debugging, and software maintenance. These capabilities allow for software updates, operational monitoring and

recovery procedures, helping to ensure system uptime without the need for physical access to the device.

Setting up Raspberry Pi Connect involves linking the device to a registered Raspberry Pi account and installing the `rpi-connect` package on the Raspberry Pi. Once installed and authenticated with a Raspberry Pi ID, the device becomes associated with the user's account and begins listening for incoming connections. It is then accessible via the dashboard on the official Raspberry Pi website, as shown in Figure 14. On the client side, no dedicated software installation is required. Remote access is initiated directly through a web browser by logging into the Raspberry Pi Connect service using the same Raspberry Pi ID. The browser securely establishes the connection, enabling access to the Pi's screen or terminal from any location with internet connectivity. Access is governed by both user authentication and device authorization, ensuring that only trusted users can interact with the system remotely. This cloud-managed architecture offers a secure and convenient means of remote administration, making Raspberry Pi Connect an ideal solution for maintaining unattended edge devices deployed in the field [30].

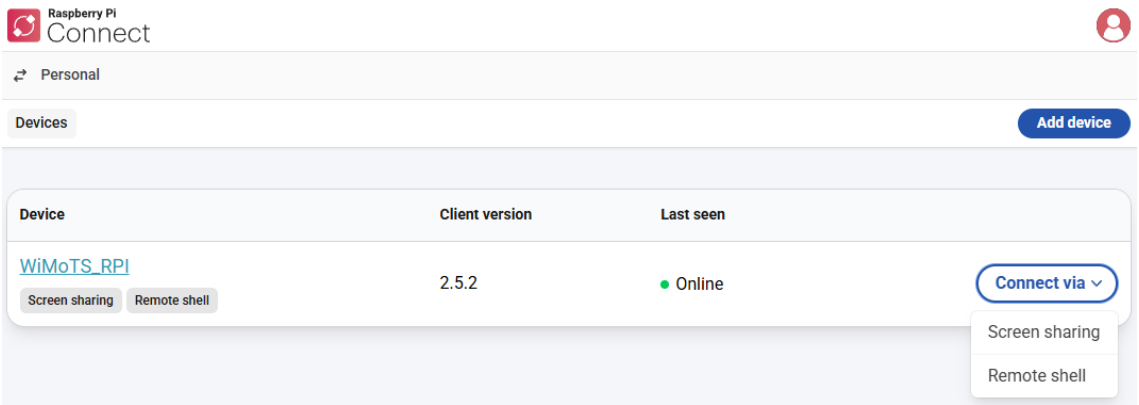


Figure 14: Raspberry Pi Connect interface

3.2 Software Environment Configuration

The edge device operates on Raspberry Pi OS (64-bit), a free Debian-based Linux distribution optimized for the Raspberry Pi's ARM architecture [31]. To ensure dependency isolation, the software stack is managed within a Python virtual environment (venv), which separates project-specific libraries from system-wide packages and minimizes the risk of conflicts or versioning issues during updates or development.

All core operations of the edge device, including video capture, file upload, and automated cleanup, are implemented in a Python script (`*camera_recording.py*`). This

script is executed automatically at regular intervals without manual intervention, enabling the device to function continuously and autonomously in the field.

To achieve this, the system leverages cron, the standard time-based job scheduler available on Unix-like systems. Cron is a daemon - a background process executing non-interactive jobs. It allows predefined tasks, known as cron jobs, to be executed at scheduled times or intervals. Each job is specified using a five-field syntax (minute, hour, day of month, month, day of week), followed by a shell command to execute, and is configured through the user's crontab file, which can be edited with the `crontab -e` command. This mechanism is ideal for running recurring scripts on headless systems where continuous operation is required [32]. Figure 15 illustrates the structure and syntax of a typical crontab entry.

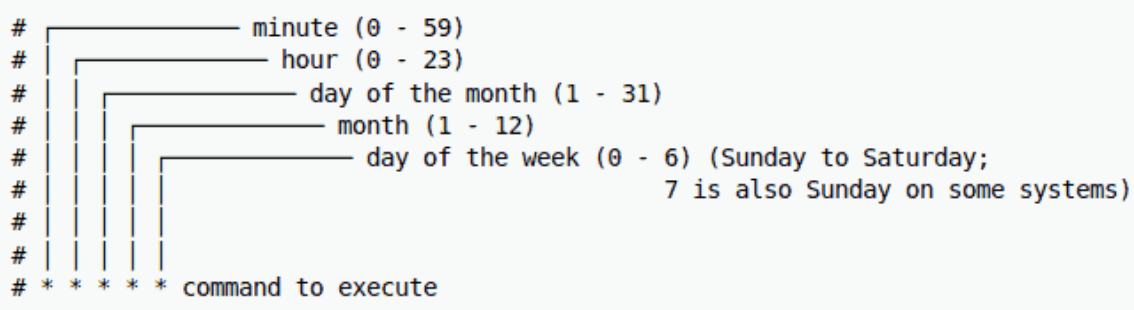


Figure 15: Crontab syntax [32]

In the case of the traffic monitoring system, cron is used to execute the `*camera_recording.py*` script every 15 minutes between 08:00 and 20:55 each day. The */15 directive in the first field denotes a 15-minute interval, while the 9-21 range restricts execution to daytime hours, when traffic activity is most relevant. The remaining asterisks indicate that the task should run every day of the month, every month, and every day of the week. This time window was selected to optimize the relevance of the captured data, while avoiding unnecessary processing during night hours when traffic is minimal. Moreover, capturing four 60-second clips per hour provides a practical and representative sampling rate for estimating hourly traffic flow, which is one of the key metrics extracted by the system's computer vision pipeline.

The configured cron expression is:

**/15 9-21 * * * DISPLAY=:0*

/home/pi48gbwimots/PycharmProjects/Camera/.venv/bin/python

/home/pi48gbwimots/PycharmProjects/Camera/Recording/camera_recording.py

Although the system is designed to operate without a physical monitor, some OpenCV functions, such as `cv2.VideoCapture()`, still require access to the system's graphical session to initialize video input correctly. This requirement stems from the way certain backends handle hardware peripherals, often depending on a valid graphical environment to interface with the camera device. Since cron runs in a minimal shell environment without inheriting user session variables, the DISPLAY variable must be explicitly defined to indicate the active graphical session. By prepending `DISPLAY=:0` to the cron command, the script is linked to the default desktop session (:0), ensuring compatibility with OpenCV and successful initialization of video capture, even in a headless configuration.

In summary, the cron job ensures consistent and reliable execution of the video recording and upload process, supporting fully autonomous operation without requiring manual intervention.

3.3 Video Capture and Local File Management

The primary function of the edge device is to periodically capture short-duration video segments and securely upload them on cloud. Video capture is implemented using OpenCV, a widely used open-source library designed for computer vision and image processing tasks. It provides high-level interfaces for handling operations such as image and video capture, video streaming, and file encoding.

Unlike static images, videos consist of multiple frames played in sequence, requiring careful handling of frame rates and smooth processing. This adds complexity, demanding efficient computation and dynamic frame management.

Reading a video file requires processing each frame individually, and then displaying or storing it as needed. To support this, OpenCV provides the `VideoCapture()` function, which is used to access the webcam and retrieve frames sequentially during each recording session. The captured frames are then passed to OpenCV's `VideoWriter()` class, which compiles them into a complete video file. This class allows the user to specify key parameters such as resolution, frame rate, and output format, enabling full control over the encoding process [33].

In the context of this system, the USB webcam is initialized via `cv2.VideoCapture(0)` to access the default video input device. Once initialized, the script attempts to configure the capture resolution to 1920×1080 pixels (Full HD). The frame rate is set to 5 frames per second (FPS), a decision that was influenced by a combination of hardware,

software, and system-level constraints. Although the webcam used in the system supports higher frame rates (e.g., 1080p at 30FPS), in practice, several factors prevent consistent performance at these levels within the Raspberry Pi environment.

Firstly, OpenCV interfaces with camera hardware via system-level backends like V4L2 (Video4Linux2) on Linux. When using consumer-grade USB webcams, these backends do not always reliably enforce user-specified settings. This limitation is often due to constraints imposed by the webcam's firmware or driver, which may only be fully configurable through proprietary software. To maintain portability and avoid vendor-specific dependencies, such software is not utilized in this system. Consequently, actual camera performance may deviate from the intended configuration.

Secondly, real-time video encoding and file writing, especially using compressed formats like MP4 with the mp4v codec, impose significant CPU and I/O load. At 1080p, each frame requires substantial processing before it can be encoded and written to storage. On a Raspberry Pi, which has limited CPU power and relies on a microSD card with modest write speed, this can lead to dropped frames or inconsistent capture rates. Lowering the frame rate to 5 FPS reduces this computational and I/O burden substantially.

Taking these characteristics into consideration, 5 FPS was chosen as a pragmatic compromise that balances acceptable visual fidelity with the limitations of the Raspberry Pi platform. This configuration ensures stable and reliable operation during each scheduled capture interval, without overtaxing system resources or degrading the quality of video data required for the computer vision processing and analysis tasks executed in the latter stages of the pipeline.

Each video recording has a fixed duration of 60 seconds, is encoded using the mp4v codec, and is saved in the MP4 container format. This duration balances storage efficiency with analytical value, offering clips that are both easy to manage and consistent with the system's temporal sampling resolution.

Filenames are dynamically generated based on the current timestamp, using the `DD-MM-YYYY_HH-MM-SS` format, ensuring traceability and uniqueness. Once recording is complete, the capture and writer objects are released to finalize the file and free system resources. This shutdown sequence is essential to prevent video file corruption or memory leaks.

In order to prevent local storage exhaustion, an automated retention policy is enforced on the edge device. At the end of each recording cycle, the script scans the working directory for video files that match the standard naming pattern. Files are sorted in descending order by their last modification time using the `os.path.getmtime()` method [34]. Only a defined number of the most recent files is retained, while older recordings are automatically deleted. This logic ensures that the device remains within its storage capacity constraints without requiring manual intervention.

The use of simple file-based management instead of more complex database tracking aligns with the system's low-overhead design philosophy. By integrating recording and cleanup operations into a single script, the edge device maintains a predictable state after each execution cycle. This streamlined approach ensures autonomous, resource-aware, and continuous operation, which are key characteristics for reliable edge computing in remote deployment scenarios.

3.4 Cloud Integration

Once a video is successfully captured and stored locally, it is uploaded to cloud storage to ensure availability for the other components of the traffic monitoring system. To support this functionality, the system integrates with Google Drive via the Google Drive API, allowing secure and automated file uploads immediately after each recording session. This process relies on robust authentication and credential management mechanisms, implemented through the OAuth 2.0 protocol.

Google Drive was selected as the cloud storage platform due to its cost-effectiveness, accessibility, and ease of integration. With free baseline storage and flexible upgrade options for personal accounts, seamless cross-platform availability, and a well-documented API, it provides a reliable and scalable backend for various deployments.

The following section outlines the cloud integration strategy, the file upload and retention workflow, as well as the authentication mechanism that enables the secure transfer of video data from the Raspberry Pi to Google Drive.

3.4.1 Google Drive API and OAuth 2.0

The Google Drive API is a RESTful web service that allows users to create applications that leverage the Google Drive cloud storage and interact with it through authenticated requests. It offers endpoints for a wide range of file operations, including uploading,

downloading, listing, updating, and deleting files, as well as managing folder structures and file metadata. These features make the API highly versatile for integrating Google Drive into automated workflows and data pipelines [35].

To initiate use of the API, developers must first register their application in the Google Cloud Console by creating a project and enabling the Google Drive API as a service. As part of this setup, the developer is required to configure the OAuth 2.0 consent screen, which involves specifying the application name, authorized scopes, and developer contact information, details that will be presented to users during the authorization process. After this configuration, developers can generate OAuth 2.0 credentials, which can be downloaded as a `*client_secrets.json*` file. This file contains the credentials necessary for the application to request user authorization and obtain access tokens. These tokens allow the application to securely interact with the user's Google Drive account, strictly within the bounds of the permissions granted [36].

The OAuth 2.0 protocol is used to securely authorize access to Google Drive on behalf of a user. As a widely adopted open authorization framework, OAuth 2.0 allows third-party applications to access user resources without directly handling sensitive credentials. OAuth 2.0 uses access and refresh tokens: the former grants temporary access to specific resources, while the latter can be used to renew access without requiring repeated user authentication.

A key concept in OAuth 2.0 is the use of scopes, which define the level of access requested by the application. Scopes can be granular, such as read-only access to a user’s Drive, or broad, such as full access to manage all files. By declaring the scopes during the authorization request, the application adheres to the principle of least privilege, ensuring that it only accesses what is necessary and thereby enhancing both security and transparency.

Once authorization is granted, the application stores the resulting access and refresh tokens locally in a `*credentials.json*` file, allowing it to make authenticated API calls. Since access tokens expire after a limited time, they must be periodically refreshed using the refresh token. This token lifecycle management ensures uninterrupted operation of long-running, unattended systems while maintaining user security.

For security purposes, the credentials and tokens must be stored in a secure location on the device and should never be hardcoded or exposed publicly. These practices help ensure the integrity and confidentiality of the authenticated session [37].

3.4.2 File Upload and Cloud Storage Management

Upon completion of each video recording, the `*camera_recording.py*` script initiates the upload process using the `PyDrive2` library, which is a lightweight Python wrapper for the Google Drive API. `PyDrive2` is well-suited for resource-constrained environments, as it abstracts the complexities of HTTP requests and OAuth 2.0 authorization, enabling secure and efficient interaction with Google Drive [38].

To upload a video, the script first constructs a file object using the `GoogleDrive.CreateFile()` method. This object is initialized with metadata that includes the file's title, derived from the timestamped filename assigned during recording, and the `parents` field, which specifies the destination folder via its unique Google Drive folder ID. This ID is hardcoded into the script to ensure consistent routing of all uploads to a centralized cloud location.

After setting the metadata, the script attaches the recorded video file to the Drive file object using the `SetContentFile()` method and executes the upload with a single call to the `Upload()` method. This call transmits the file to Google Drive using credentials previously obtained via the OAuth 2.0 flow. The upload process is fully automated and integrated into the same execution cycle as the recording [39].

To optimize cloud storage usage, the system implements an automated retention policy for the target Google Drive folder, mirroring the approach used for local file management. After each successful upload, the script queries the target Drive folder to retrieve all files in the folder using a search condition of the form:

q = f"'{folder_id}' in parents and trashed=false"

The resulting list of files is filtered to include only those that follow the naming convention for recorded videos. These files are then sorted in descending order based on their `createdDate` field, which corresponds to the upload timestamp.

The system retains only a defined number of the most recent files in the designated Drive folder. If the number of video files exceeds this threshold, the oldest ones are deleted using the `Delete()` method. This automated cleanup ensures the system remains within Google Drive's free-tier storage limits while maintaining a concise, up-to-date dataset for the computer vision processing stage that follows in the pipeline.

3.4.3 Security Mechanism

In order to interact securely with the Google Drive API, the edge device must be authorized to access the user's cloud storage on their behalf. This is achieved via the OAuth 2.0 protocol. Its implementation is facilitated by the PyDrive2 library, which simplifies token handling, scope management, and secure request authorization.

Before the edge device can access Google Drive, the developer must first register the application through the Google Cloud Console. This process involves three main steps: creating a project, enabling the Google Drive API, and configuring OAuth 2.0 credentials.

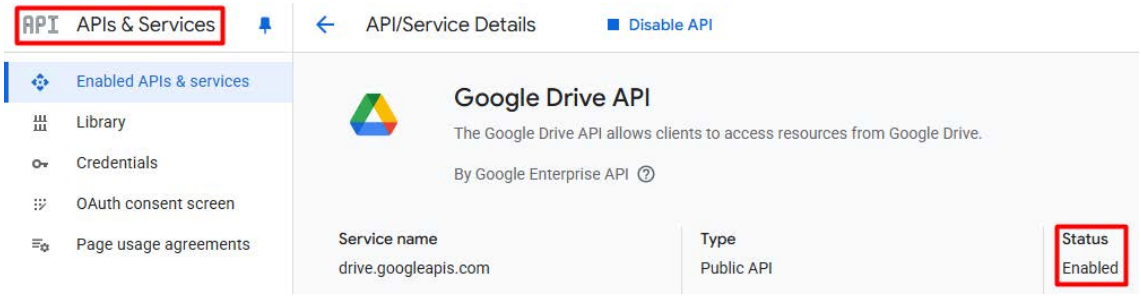


Figure 16: Enabling the Google Drive API

After creating a new project in the Google Cloud Console and enabling the Google Drive API, the OAuth 2.0 consent screen must be configured. This includes setting the application name, user's contact information, and specifying the access scopes. As shown in in Figure 17, the `https://www.googleapis.com/auth/drive.file` scope is granted, which is classified as a non-sensitive scope, and allows access to manage files in the user's Google Drive. Non-sensitive scopes provide the smallest scope of authorization access and only require basic app verification [40].

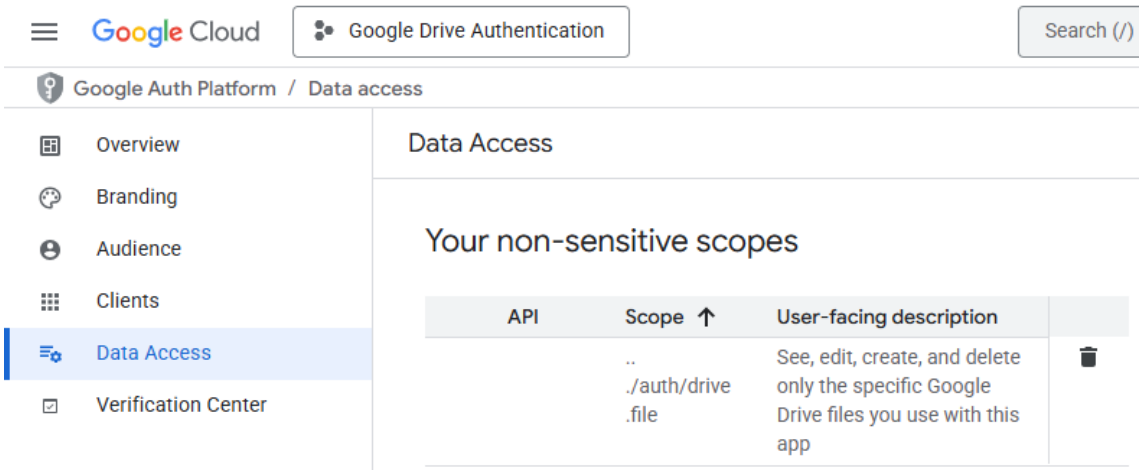


Figure 17: Access scopes configuration

The developer must then generate OAuth 2.0 credentials. The appropriate application type is "Desktop App", as the authorization process involves launching a local web server for interactive login. Upon creation, the client credentials can be downloaded as a `*client_secrets.json*` file. This file contains the client ID and client secret associated with the application, as well as redirect URIs and other metadata required to initiate the OAuth flow.

Figure 18: OAuth 2.0 Credentials generation

When the script is run for the first time, it uses this file to launch a local authorization server. A browser window opens, prompting the user to log in and approve the requested permissions. Once the user consents, Google issues an authorization code, which is exchanged for an access token and a refresh token. These are saved locally in a file named `*credentials.json*`. This process is handled automatically by PyDrive2.

On subsequent executions, the application loads `credentials.json` and checks whether the access token is still valid. If the access token it contains is still valid, it is used to authenticate all outgoing requests to the Google Drive API. If the token has expired, the system refreshes it silently using the refresh token. This functionality is enabled by explicitly requesting offline access during the initial OAuth flow. By specifying `access_type='offline'`, the application ensures that Google will issue a long-lived refresh token.

This configuration is essential for autonomous systems intended to operate continuously in remote or unattended environments. By enabling offline access and implementing a seamless token refresh mechanism, the system maintains persistent access to Google Drive without requiring repeated user authentication. This ensures reliable, long-term operation, even after reboots or prolonged idle periods, which is critical in edge deployments where manual reauthorization is impractical.

If the refresh token is invalidated (e.g., through manual revocation by the user), the system falls back to the original web-based authorization flow. The user is prompted to reauthorize the application, restoring access without affecting system integrity.

The `*credentials.json*` file is stored locally on the edge device in a non-public project directory. All sensitive authentication credentials are managed externally and are not hardcoded in the source code, in accordance with secure coding and credential management best practices [41].

3.4.4 Recorded Video Upload Latency Analysis

To evaluate the impact of different network configurations on video upload performance, the table below presents a comparison of theoretical upload speeds and corresponding average upload durations observed under varying connection types. Measurements were conducted on a series of newly captured video files, each with an approximate size of 18 MB. The results indicate the practical differences between wireless and wired connections in terms of latency and throughput, highlighting the influence of protocol overhead and hardware limitations on real-world performance.

Table 3: Video upload time under different network conditions

Connection Type	***Theoretical Upload Speed (Mbps)***	***Average Upload Time (sec)***
Wi-Fi 4	10	22.60
Wi-Fi 5	50	7.70
Ethernet	100	3.40

While the theoretical upload time for a video file can be derived from its size and nominal bandwidth, real-world conditions typically introduce significant additional latency. These discrepancies result from protocol overhead (e.g., HTTPS encryption, metadata exchange), wireless channel variability, and limited processing resources on the edge device. For example, benchmark tests show that the Raspberry Pi 4, despite supporting dual-band 802.11ac Wi-Fi, achieves real-world wireless throughput substantially below its theoretical maximum, with upload speeds often constrained by driver efficiency, signal quality, and CPU usage during encryption [42]. Moreover, the Google Drive API incurs extra overhead through session negotiation, chunked upload management, and post-upload finalization. As a result, observed upload durations consistently exceed ideal estimates, especially on lightweight hardware like the Raspberry Pi connected via Wi-Fi [43].

The observed data highlights a diminishing return in performance with higher Wi-Fi speeds on the Raspberry Pi, due to the wireless performance variability and system-level constraints described above. However, ethernet provides significantly more stable and efficient throughput, making it the preferred option for latency-sensitive or high-throughput applications.

4 Webhook Mechanism

This chapter provides a technical analysis of the webhook-based event detection and triggering mechanism used in the traffic monitoring system. It focuses on the implementation of Google Drive's push notification system, the secure exposure of a webhook listener in a private network environment, and the server-to-server interaction enabled via a service account for automated and unattended operation. The design introduces improved system modularity, responsiveness, and enables long-term unattended execution under real-world deployment conditions.

4.1 Webhook Architecture and Design Considerations

To enable real-time responsiveness and eliminate the inefficiencies of continuous polling, the system adopts a webhook-based mechanism that allows Google Drive to actively notify the processing unit when new video files are uploaded. This webhook mechanism serves as the primary interface between Google Drive and the centralized computer vision pipeline, enabling near real-time detection and processing of incoming footage. By replacing traditional polling with an event-driven model, the system minimizes latency and reduces unnecessary resource consumption.

The webhook listener runs on a private processing unit without a public IP address. To make it externally accessible for receiving Google's webhook callbacks, it is exposed to the internet via a secure tunnel using ngrok. Upon receiving a change notification from Google Drive, the webhook extracts relevant metadata (e.g., file ID), downloads the video, and initiates the video processing workflow.

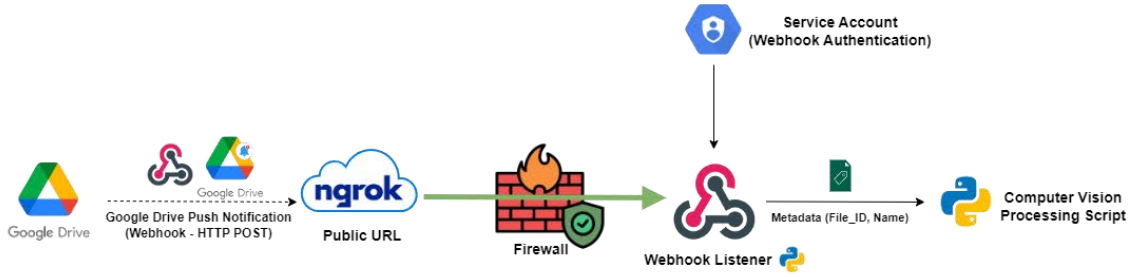


Figure 19: Webhook mechanism architecture

4.1.1 Exposing the Webhook via ngrok

A webhook is a lightweight, event-driven communication mechanism that allows external services to notify an application when a specific event occurs. Instead of continuously polling a resource for changes, a webhook allows the server to send an HTTP POST request to a designated URL whenever an event is triggered. This makes webhooks highly efficient for integrating real-time workflows across distributed systems [44].

In the context of the traffic monitoring system, a webhook listener runs locally on the processing unit and awaits HTTP POST notifications from Google Drive via the Drive API's push notification feature [45]. However, since the processing unit is located behind a firewall and does not have a public IP address, it cannot directly receive incoming requests from Google's notification servers.

To bridge this connectivity gap, the system uses ngrok, a secure tunneling and reverse proxy service that allows external services to access local network resources behind NAT or firewalls. Ngrok establishes an outbound connection from the local machine to its cloud infrastructure and creates a publicly accessible HTTPS URL, effectively exposing the webhook listener to the internet without requiring a public IP or domain name [46].

Once the tunnel is established, the system retrieves a static subdomain and uses it as the callback endpoint during webhook subscription with the Google Drive API. This ensures that change notifications for new file uploads are reliably forwarded to the local listener.

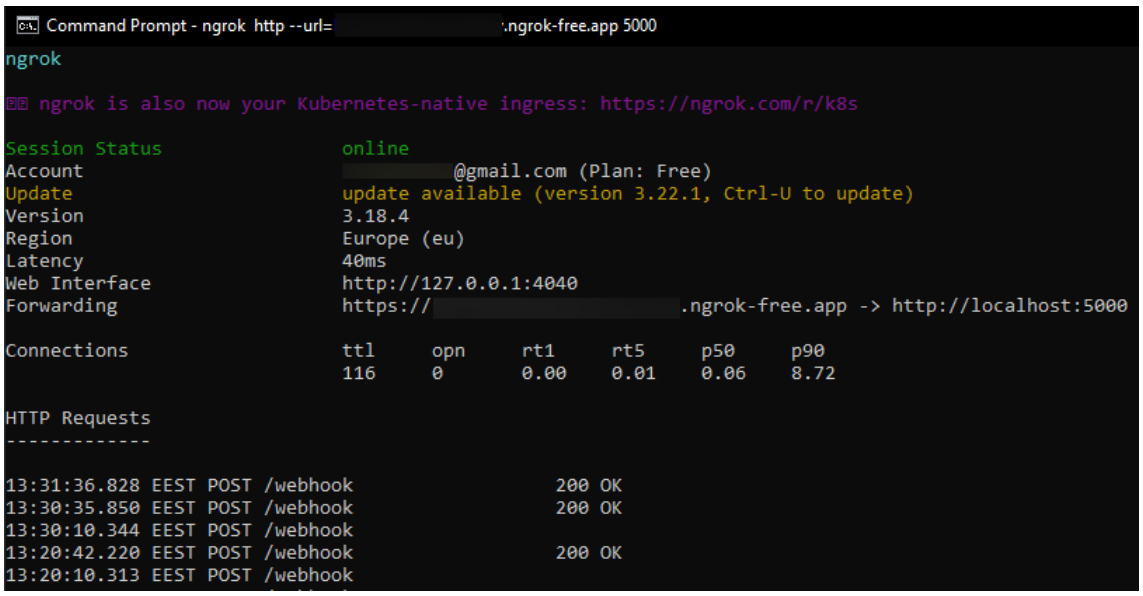


Figure 20: ngrok tunnel dashboard

Ngrok's tunnel startup is fully automated as part of the system's initialization routine, ensuring continuous availability of the listener after reboots or crashes without manual intervention. This is achieved by placing a custom batch script (start_ngrok.bat) in the Windows startup folder, which launches ngrok with the appropriate subdomain and port binding upon system boot.

In addition to startup automation, a secondary watchdog script (restart_ngrok.bat) runs continuously in the background and checks for both internet availability and ngrok process status. If connectivity is lost or ngrok is unexpectedly terminated, the script automatically reinitializes the tunnel.

Overall, this approach offers a reliable method for exposing a local webhook listener in constrained network environments, without requiring static IPs, domain registration, or firewall reconfiguration. By combining ngrok's tunneling capabilities with automated startup and fault recovery scripts, the system achieves persistent public accessibility, aligning with the design goals of autonomy, operational reliability, and ease of deployment.

4.1.2 Push Notifications via Google Drive API

Google Drive API offers a push notification system that enables applications to receive real-time alerts when changes occur in files or folders. This event-driven approach eliminates the need for continuous polling, thereby reducing network overhead and improving efficiency. The configuration is automated by a dedicated script (`*register_webhook.py*`).

Push notifications in Google Drive are facilitated through notification channels. To set up a channel, an application sends a watch request to the Drive API, specifying the resource to monitor and providing a publicly accessible HTTPS endpoint (e.g., webhook URL) to receive notifications. Once established, Google Drive sends an HTTP POST request to the specified webhook URL whenever the monitored folder changes (e.g., when a new video file is uploaded).

To configure such a notification channel, the system issues a `files().watch()` request, which must include the following parameters:

- *id*: A unique identifier for the channel, generated at runtime using a UUID (e.g., `CHANNEL_ID = str(uuid.uuid4())`). This value must be globally unique across all channels and is used to identify the lifecycle of the watch.

- *type*: Set to `web_hook` to indicate that the channel will deliver notifications via HTTP POST requests to a webhook endpoint.
- *address*: The public webhook URL (in this case, the ngrok-exposed URL that points to the locally hosted webhook listener).
- *resourceID*: A server-assigned identifier representing the Drive resource being monitored.

If the watch request successfully creates a notification channel, it returns an HTTP `200 OK` status code. The `200` OK response is required by Google Drive to confirm receipt of the webhook notification. Without it, Google Drive may retry sending the notification, assuming the request failed.

Once the watch is active, Google sends an HTTP POST request to the webhook listener each time a change occurs in the monitored folder. This notification request includes several HTTP headers that convey metadata about the event. These headers are used to determine the nature and source of the change:

- *X-Goog-Resource-State*: Type of change on the watched resource (e.g., add, update, remove). In this system, the update state is received when a new file is added to a folder due to a change in its `children` collection.
- *X-Goog-Resource-ID*: Persistent ID of the watched resource.
- *X-Goog-Channel-ID*: Unique developer-defined ID for the notification channel.
- *X-Goog-Message-Number*: Sequential number for ordering and deduplication of notifications.
- *X-Goog-Channel-Expiration*: Expiration timestamp for the current notification channel, which requires periodic renewal.
- *X-Goog-Changed*: Type of modified collection (e.g., children, permissions). In this system, `children` indicates file additions to a folder.
- *X-Goog-Resource-URI*: Direct Drive API URL of the changed resource.
- *X-Forwarded-For, X-Forwarded-Host, X-Forwarded-Proto*: Proxy headers added by ngrok to relay the original request context.

A limitation of the files().watch() method is the temporary nature of notification channels. A channel's lifetime is limited to a maximum of 24 hours (86,400 seconds) from its creation time. If the expiration field is not explicitly set during registration, the API defaults to a 1-hour lifespan (3,600 seconds). After expiration, the channel is automatically

deactivated, and no further notifications are delivered unless a new watch request is issued [45].

To address this constraint and ensure uninterrupted monitoring, the system employs an automated renewal mechanism based on Windows Task Scheduler. A scheduled task is configured to execute the webhook registration script once every hour, ensuring that a new notification channel is created before the currently active one expires. As part of the renewal process, the system first deactivates the existing channel by invoking the channels().stop() method, and then issues a new files().watch() request using a freshly generated channel ID. This proactive approach eliminates the need to manage expiration timestamps manually and ensures continuous reception of change notifications. It also enhances system robustness by preventing lapses in webhook delivery due to channel expiration or unexpected interruptions.

The scheduled task is configured with the following properties:

- *Trigger*: Triggered one time and then repeated every 1 hour indefinitely.
- *Action*: Start a program, specifically a .bat script that initializes the appropriate Python virtual environment and executes the webhook registration script.
- *Working Directory*: The batch script first sets the application folder as the working directory to ensure that the Python script and any relative file paths it uses are correctly located and executed.

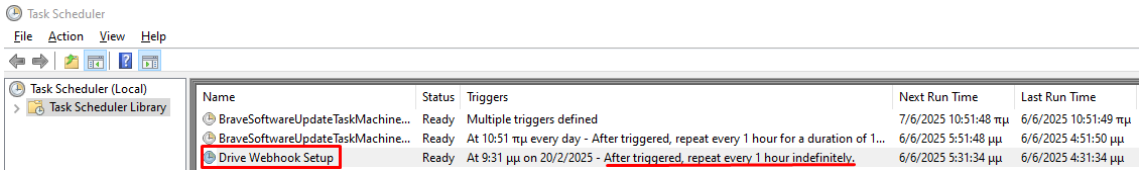


Figure 21: Scheduled task configuration

4.2 Webhook Listener Implementation

The webhook listener (*webhook_listener.py*) is implemented in Python using Flask, a lightweight WSGI web framework well-suited for microservices and API endpoints. It exposes a single HTTP POST endpoint at /webhook, which is responsible for handling push notifications sent by Google Drive when changes occur in the monitored folder.

Each time a new file is uploaded to the Drive folder, Google sends an HTTP POST request to the public webhook URL (exposed via ngrok). This request contains no body payload but includes a set of HTTP headers as described in Section 4.1.2. These headers

are parsed by the Flask route to determine the nature and context of the event. The key headers processed by the listener are:

- *X-Goog-Resource-State*: The type of event (e.g., `update`, which in this case usually indicates a new file added to the folder's children).
- *X-Goog-Resource-URI*: The canonical API endpoint for the changed file, from which the file ID can be extracted.
- *X-Goog-Channel-ID* and *X-Goog-Resource-ID*: Used to validate the source of the notification and ensure it matches the expected Drive resource.

The listener filters out irrelevant events (e.g., resource states that are not `update`) and proceeds only when the notification corresponds to a valid file addition.

Unlike some webhook systems that include a full JSON payload, Google Drive push notifications are minimal by design. However, the X-Goog-Resource-URI header typically includes a full REST endpoint such as:

`https://www.googleapis.com/drive/v3/files/{fileId}?alt=json`

The listener extracts the file ID by parsing this URI and isolates the relevant portion between `/files/` and the query string (`?alt=json`). This ID is then used to request the file's metadata and content from the Drive API. To reduce false positives, the system applies additional checks using filename patterns and MIME types (e.g., video/mp4) to ensure the file is a valid traffic video before processing it further.

Once a valid file ID has been identified, the listener invokes the Drive API's `files().get_media(fileId=...)` method to download the uploaded video into a local directory on the processing unit. The file is saved with its timestamp-based filename to ensure traceability. This step is critical because the Drive push notification does not include file content or metadata directly, but only a reference. Thus, this API call bridges the gap between detection and data acquisition.

The measured download latency of the uploaded video from Google Drive to the central processing unit exhibits minimal variance and consistently low latency, ranging from 100 to 110 ms, as illustrated in Figure 22. This suggests that the cloud storage retrieval process is not a significant bottleneck in the system's overall pipeline. Given the small file size and the use of a stronger network connection on the processing side, video retrieval from the cloud can be considered effectively instantaneous for the purposes of end-to-end system latency analysis.

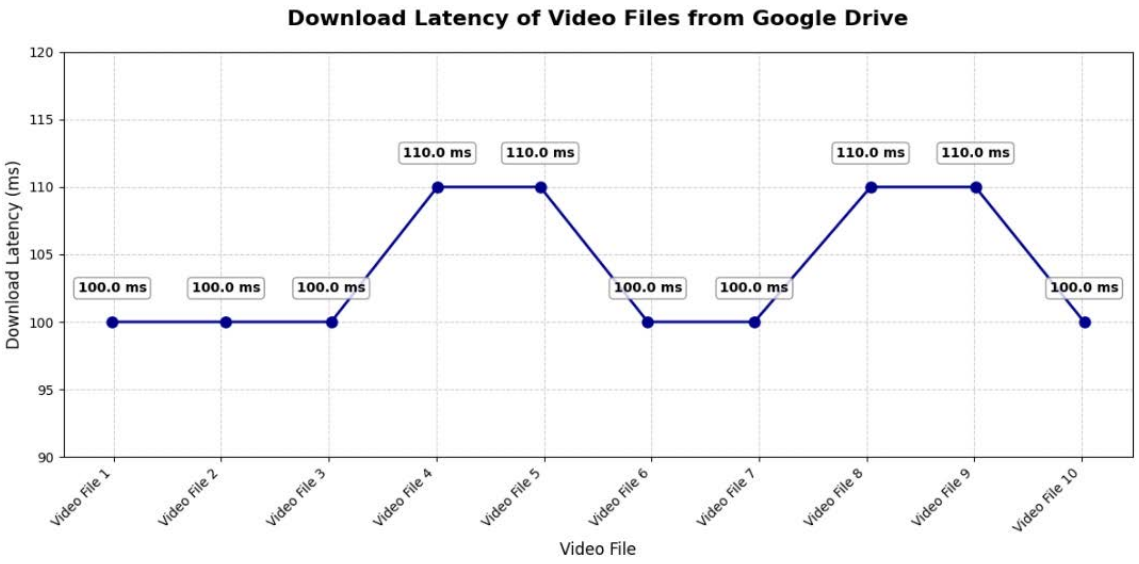


Figure 22: Download latency of video files from Google Drive

The script incorporates logging mechanisms to record each received notification, parsed file ID, and download attempt. Exception handling is implemented to detect and log issues such as missing headers, invalid request formats, or failed file downloads. This ensures traceability and simplifies debugging in case of processing errors.

Additionally, the Flask app does not respond with a `200 OK` status code until the notification has been validated. This complies with Google's API delivery model, where any response other than `2xx` triggers automatic retries [45].

To ensure idempotency and prevent duplicate processing, the system employs a text-based tracker file that records the ID or name of the last successfully processed video [47]. Upon receiving a new notification, the webhook handler queries the monitored Google Drive folder for the most recently uploaded file and compares it to the tracker. If the file has already been processed, the event is ignored.

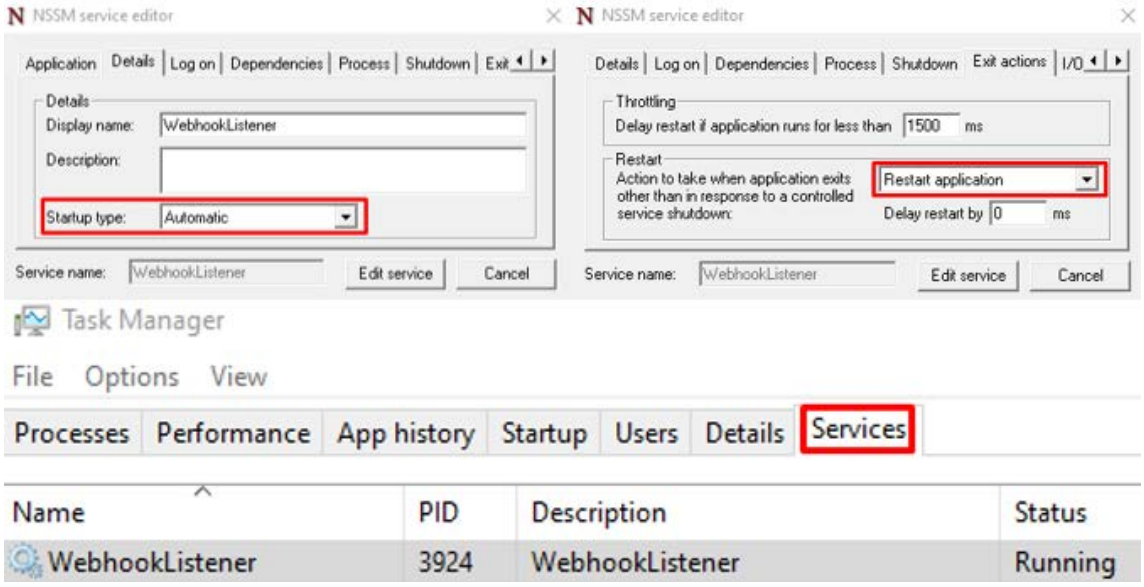
To further safeguard against race conditions, particularly during webhook events bursts, the system employs a debouncing mechanism using a threading lock to serialize execution and prevent concurrent processing. This design ensures that even if Google retries delivery due to delayed 200 OK responses, the system remains robust and avoids duplicate handling of the same file.

To ensure continuous background execution of the webhook listener, the system utilizes NSSM (Non-Sucking Service Manager), a lightweight Windows utility that allows arbitrary executables to run as persistent Windows services Unlike traditional batch scripts or manually launched processes, NSSM-managed services benefit from automatic

startup at boot, graceful recovery upon failure, and integration with the Windows Service Control Manager (SCM) [48].

In this implementation, NSSM is used to install and manage the `*webhook_listener.py*` script as a Windows service. This approach ensures the webhook endpoint remains continuously available without requiring manual user intervention. As illustrated in Figure 23, the configuration includes:

- Setting the Startup type to `Automatic`, ensuring the service starts automatically on system boot.
- Selecting the `Restart Application` policy in the Exit action tab. This instructs the system to automatically restart the service if it exits unexpectedly.



- Figure 23: NSSM webhook listener service configuration

Once configured correctly, the service appears as `Running` under the Services tab in Windows Task Manager, indicating successful deployment and operation.

In summary, the webhook listener acts as a bridge between Google Drive's event stream and the local video processing pipeline. It efficiently filters incoming events, retrieves the relevant file, and triggers further analysis without the need for manual intervention.

4.3 Server-to-Server Authentication with Google Service Account

To enable secure, unattended access to the Google Drive API from the centralized processing unit, the system employs a Google Service Account. A service account is a special type of Google account intended for automated, non-human interactions within Google Cloud. It is identified by its email address, which is unique to the account [49]. This authentication model is better suited to backend services and long-running daemons than traditional user-based OAuth 2.0 consent flows employed by the Raspberry Pi edge device.

Unlike OAuth 2.0, which requires user consent and is designed for delegated access on behalf of individual users, a service account provides server-to-server authentication using cryptographically signed credentials (JSON key files). This approach eliminates the need for interactive login flows, making it ideal for headless components such as the webhook listener.

4.3.1 Service Account Configuration

The service account used in this system is created and configured via the Google Cloud Console. After creating a new project in the Google Cloud Console and enabling the Google Drive API, a new service account is created and assigned a descriptive name. Upon creation, a private key is generated and exported in JSON format. This file (*`service_account.json`)* contains the necessary credentials, including a client ID, private key, and project metadata, required to authenticate with Google APIs from within the application codebase, and is stored securely on the processing unit [50].

To grant access to the relevant Drive content, the target folder used by the edge device for video uploads is explicitly shared with the service account's email address (e.g., webhook-trigger@project-id.iam.gserviceaccount.com) using standard Drive permissions. The service account was granted both `Editor` and `Owner` roles, as illustrated in Figure 24, allowing it to read, write, delete, and manage access to all contents within the folder. However, since no other Drive resources are shared with the account, it remains effectively isolated from the rest of the user's Drive.

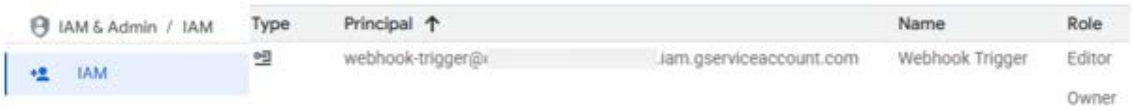


Figure 24: Service account roles configuration

The service account is provisioned with a combination of IAM (Identity and Access Management) roles and OAuth 2.0 scopes, which collectively define the boundaries of its access. IAM roles govern what actions the service account can perform within the Google Cloud project (e.g., token generation, API usage), while OAuth scopes restrict the categories of data it can access during authenticated Drive API calls. In this system, the scope `https://www.googleapis.com/auth/drive` is applied, which grants access to all Drive files the account has permission to view. In practice, this means only the shared folder is accessible [51].

4.3.2 Integration into the Webhook Workflow

Authentication is implemented using the `google-auth` and `google-api-python-client` libraries. Upon initialization, each script involved in the webhook workflow (`*webhook_listener.py*`, `*register_webhook.py*`) creates a credentials object using the `Credentials.from_service_account_file()` method, which is imported from the `google.oauth2.service_account` module [52]. This method loads the service account's private key from the locally stored `*service_account.json*` file and applies the required OAuth scope. The credential is then passed to the `build()` function to instantiate an authenticated Drive service client, which is used to perform operations such as subscribing to push notifications, retrieving metadata, or downloading uploaded videos.

This authentication process follows the JWT Profile for OAuth 2.0 Authorization Grants (RFC 7523), whereby a JSON Web Token (JWT) is signed using the service account's private key and exchanged for a short-lived access token via Google's OAuth 2.0 token endpoint [52]. These tokens expire after 1 hour [53]. This expiration model aligns well with the system's execution design, as the `*register_webhook.py*` script is automatically run every hour using Windows Task Scheduler, ensuring that a fresh token is generated each time the Drive watch channel is re-registered.

This model offers two major advantages: it avoids the complexities of interactive OAuth 2.0 flows, which are unsuitable for headless systems, and it ensures that token generation and renewal are handled automatically, without requiring additional logic or

user intervention. By integrating the service account into all webhook components, the system achieves a secure, self-sustaining, and fully automated authentication layer for Drive access. This design not only reduces operational overhead but also adheres to security best practices for scalable server-to-server communication in production environments.

5 Computer Vision Pipeline

This chapter provides a technical overview of the computer vision pipeline that drives the automated traffic monitoring system. It examines the implementation of the video processing workflow, detailing how computer vision tasks are performed on the central processing. The pipeline is executed automatically upon the arrival of new video recordings, enabling near real-time, headless operation. Its design combines efficient computer vision tasks with structured data generation, secure result delivery via MQTT, and data storage to InfluxDB. This seamless integration establishes the pipeline as the core analytical component of the system, enabling continuous and scalable traffic monitoring in real-world deployment conditions.

5.1 Computer Vision Pipeline Overview

A computer vision pipeline is a structured sequence of processing stages designed to extract meaningful information from visual data such as images or video streams. It is the core component of the traffic monitoring system, enabling the automated extraction of traffic-related metrics from recorded video footage. Its primary function is to transform raw visual data into structured, actionable metrics such as vehicle counts, estimated speeds, and traffic intensity levels. The pipeline is triggered automatically upon the detection of a new video upload, a process managed by the event-driven webhook mechanism analyzed in the previous chapter. Once initiated, the pipeline follows a sequential workflow that begins with video input handling and proceeds through a series of vision and data processing stages. The pipeline's logic is implemented in a Python script (*Traffic_Analytics.py*), which orchestrates each stage of the process

First, the latest downloaded video is loaded and processed frame-by-frame using a YOLO-based object detection model that identifies vehicles within a region of interest. Detected vehicles are then classified and assigned consistent identities across frames using a tracking algorithm, allowing for accurate counting and movement analysis. A speed

estimation stage follows, which leverages perspective transformation and temporal analysis of vehicle motion within a defined sub-region of the video frame. In parallel, vehicles crossing predefined virtual lines are counted, classified by type and direction.

After inference is completed, the extracted data is aggregated into a structured JSON format aligned with smart city data models and is subsequently published to an MQTT broker for real-time consumption by external subscribers and stored in an InfluxDB time-series database for historical querying and visualization. The entire computer vision pipeline is deployed general-purpose PC without GPU acceleration, operates headlessly without the need for user intervention, and runs in a fully unattended manner. Figure 25 illustrates the workflow of the computer vision pipeline.

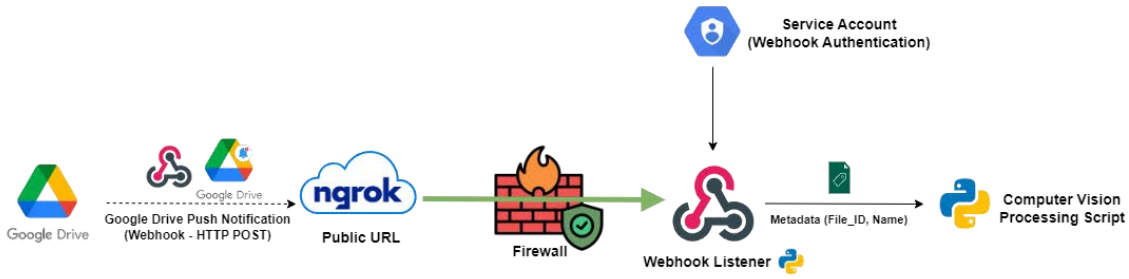


Figure 25: Computer vision pipeline workflow

5.2 Input Management and Dynamic Video Processing

The main design consideration for the computer vision pipeline was to ensure autonomous operation in response to the availability of newly recorded traffic footage. As such, the input management logic was engineered to dynamically discover and process the latest available video without requiring manual configuration. This mechanism ensures robustness in real-world deployments, where videos are continuously generated by edge devices and downloaded by the central processing unit into a shared directory.

The input video files follow a predefined naming convention that encodes the exact timestamp of their recording. Each filename adheres to the pattern `record_DD-MM-YYYY_HH-MM-SS.mp4`, which serves two purposes: it uniquely identifies each recording in chronological format, and it allows the pipeline to extract the embedded timestamp and associate it with the corresponding traffic metrics. This design choice simplifies temporal indexing and enables historical querying.

Once the pipeline is triggered, it invokes the `get_latest_video()` function, which programmatically scans the configured video input directory (./Drive_Videos). This function

uses the `glob` library to search for all files matching the `.mp4` extension and sorts them by creation time using the `os.path.getctime` method [34]. By selecting the most recently created file, the function ensures that the latest available video is always processed, without requiring user input or hardcoded filenames. If no video is found, a `FileNotFoundError` exception is raised [54].

To increase flexibility and support both automated and manual execution modes, the script provides a command-line interface (CLI) using the `argparse` module [55]. The `--video` argument allows users to explicitly specify the path to a particular video file for processing. If this argument is omitted, the script defaults to invoking `get_latest_video()` and proceeds with the most recent file in the directory. This dual-mode functionality supports both batch processing (e.g., for testing or reprocessing archived videos) and continuous deployment in unattended environments.

After identifying the video file to process, the timestamp is extracted using a regular expression defined in the `extract_timestamp()` function. This function uses Python's `re` module to match the date and time components embedded in the filename [56]. Upon successful pattern matching, the extracted values are converted into a Python datetime object, which is then formatted into ISO 8601 string format [57]. This standardized timestamp is inserted into the `results["time"]` field of the output JSON structure, providing a consistent temporal reference for downstream analytics and storage in the InfluxDB database.

5.3 Model Selection - Object Detection - Classification

The object detection task of the computer vision pipeline is implemented using the YOLO11 object detection model, executed via the ultralytics python package. Specifically, the yolo11l pre-trained model is used, as it offers a strong balance between detection accuracy and inference speed, making it well-suited for deployment on a general-purpose PC without GPU acceleration [15].

Upon initialization, the pipeline script loads the pretrained `yolo11l.pt` model file from the project's directory. This model is capable of performing multiclass object detection across the 80 classes defined in the COCO dataset. However, to ensure relevance and computational efficiency, the system is configured to selectively retain only detections associated with road vehicles. Specifically, the detection results are filtered to include only objects with COCO class IDs `[2, 3, 5, 7]`, which correspond to the categories car,

motorcycle, bus, and truck, respectively [20]. This class-based filtering is applied immediately after the model inference on each video frame, discarding irrelevant detections (such as pedestrians, or other background objects) and reducing the computational overhead in subsequent processing stages.

Each video frame is passed to the YOLO11 model, which outputs a set of bounding boxes, class labels, and additional metadata. These raw outputs are wrapped into a structured `Detections` object using the `supervision` library, an open-source computer vision toolkit designed to simplify and modularize the development of real-time vision pipelines. Developed by the Roboflow team, supervision provides high-level abstractions for working with detection results, including utilities for annotation, tracking, geometric filtering, and video frame manipulation [58]. In the context of this pipeline, it standardizes the interface between the model outputs and the subsequent post-processing operations by converting YOLO's raw detections into a format that supports filtering, spatial reasoning, and visualization.

After the initial class-based filtering, an additional spatial filtering step is applied using a polygonal region of interest. This spatial constraint is implemented through a `PolygonZone`, defined by four source points that outline the main traffic zone within the video frame. Its purpose is to exclude detections occurring outside the designated region of interest, such as those resulting from pedestrians on sidewalks, or vehicles outside the monitored lanes. The `PolygonZone` uses the geometric coordinates of each detection's anchor point to determine whether it lies within the defined region [59]. These anchor points are extracted using the `get_anchors_coordinates()` method of the `Detections` object, with the anchor position set to `BOTTOM_CENTER`, as it provides a consistent reference point that approximates the physical contact of the vehicle with the road surface [60]. This filtering is performed by invoking the `trigger()` method of the `PolygonZone` object, which applies the geometric containment test to the detection set, returning a boolean mask (`ndarray`) that ensures only in-zone detections are passed to subsequent processing stages [61]. Only detections that both belong to one of the target vehicle classes and fall within this zone are retained for tracking and analysis.

This two-stage filtering approach ensures that computational resources are focused exclusively on relevant entities. It also enhances the semantic accuracy of the extracted metrics, as only vehicles within the monitored area are considered in the final traffic analysis. This concept is illustrated in Figure 26, where only objects identified as vehicles

(i.e., those with class IDs corresponding to car, bus, truck, etc.) and whose anchor points fall within the defined polygon zone (outlined in red) are retained for further analysis. The spatial filtering zone is carefully calibrated to match the road surface under observation, excluding sidewalks, parked vehicles, or unrelated traffic outside the target area. Therefore, the selection of YOLO11 combined with this post-inference filtering logic offers a practical trade-off between precision and computational efficiency, making it well-suited for real-world traffic monitoring systems.

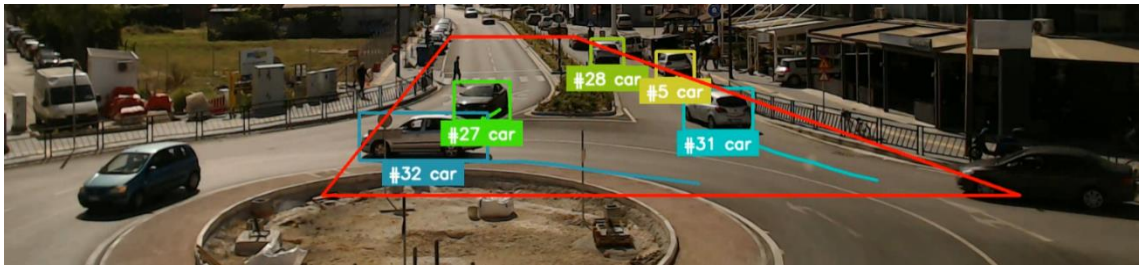


Figure 26: Polygon detection zone for spatial filtering of vehicle detections

5.4 Object Tracking

In order to maintain consistent identities for detected vehicles across video frames, a multi-object tracking mechanism based on the ByteTrack algorithm is integrated into the computer vision pipeline. ByteTrack is a lightweight and high-performance tracking-by-detection algorithm that assigns persistent IDs to detected objects across consecutive frames without requiring appearance-based features or re-identification models.

After each frame is processed and filtered, the resulting detections are passed to a tracking module powered by the `ByteTrack` class provided by the `supervision` library. The ByteTrack tracker is initialized with the video's frame rate as a parameter (`frame_rate=video_info.fps`), ensuring that its temporal association logic is aligned with the input video's characteristics.

To support trajectory-based analytics, the system maintains a coordinate history for each tracked object using a defaultdict of deque structures [62]. Each `deque` stores up to one second's worth of coordinates (`maxlen=video_info.fps`) [63]. This rolling buffer approach ensures memory efficiency while preserving sufficient temporal resolution for reliable speed estimation and motion analysis. The coordinates stored represent the transformed anchor points of each tracked vehicle, forming time-indexed trajectories used to compute velocity vectors and movement patterns.

The tracker maintains a dictionary of active tracks, indexed by object ID, and updates them based on spatial proximity and temporal continuity. By linking bounding boxes across frames based on location, motion trajectory, and object class, it assigns each vehicle a stable `tracker_id`. This persistent ID assignment is critical for both vehicle counting and speed estimation tasks, as it enables the association of each detection with a temporal history of positions [64].

For accurate temporal association, the pipeline script extracts a stable anchor point from each detection using the `get_anchors_coordinates()` method of the `Detections` object. The bottom-center of each bounding box is selected as the anchor [60]. While the same anchor point is initially used for geometric filtering during object detection, its role becomes critical in subsequent processing stages such as trajectory formation, perspective transformation, and speed estimation, where accurate spatial and temporal alignment is required.

Maintaining object-specific identities across frames further enables class-wise identity persistence. Each `tracker_id` is associated with both spatial coordinates and a semantic class label (e.g., car, bus, truck), allowing the system to compute accurate metrics over time. This is particularly important for generating detailed analytics, such as per-class vehicle counts and average speeds by vehicle type. By ensuring that a vehicle retains a consistent ID and class label across its visible trajectory, duplication is avoided, statistical reliability is improved, and result interpretability is enhanced.

5.5 Perspective Transformation and Speed Estimation

A key objective of the computer vision pipeline is to estimate the speed of vehicles observed in the video stream. In order to accurately estimate the speed of vehicles from video captured at an angle, we must first correct for the perspective distortion caused by the camera's viewing angle. The raw camera image is a projective view of the 3D world, meaning that objects farther from the camera appear smaller, and distances in pixel space do not linearly correspond to real-world distances. This distortion makes direct speed computation from image coordinates unreliable.

To resolve this, a geometry-aware method is employed, that compensates for the effects of camera perspective by projecting pixel coordinates into a transformed view that approximates real-world distances. The mechanism relies on perspective transformation—also known as planar homography—a computer vision technique that

enables the mapping of image-plane points to a rectified coordinate system where distance and displacement can be meaningfully interpreted [65].

The pipeline defines two spatial regions within the video frame: the SOURCE zone and the SUBZONE. The SOURCE polygon outlines the primary region of interest for vehicle detection and tracking. It encompasses the roadway surface on which vehicle activity occurs and serves as the spatial filter during the initial object detection phase. In contrast, the SUBZONE is a smaller, calibrated sub-region within the SOURCE zone, as illustrated in Figure 26. It is designed specifically for speed estimation and corresponds to a section of the road where vehicles move primarily in a straight line and their motion is most reliably measurable in image-space coordinates.

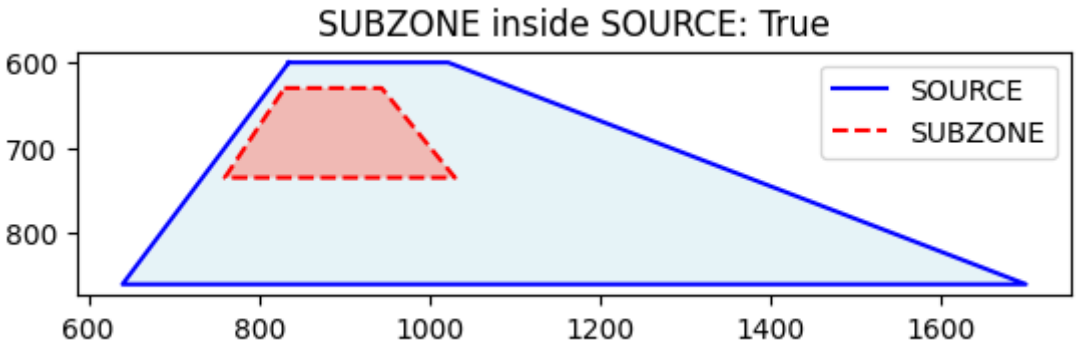


Figure 27: SUBZONE within SOURCE zone

The speed estimation relies on the computation of a homography matrix, which defines a projective transformation between the image plane and a simplified target plane. Homography enables the mapping of points from the original camera view to a rectified plane where distances and displacements can be meaningfully interpreted. This transformation assumes that vehicle motion occurs on a flat, planar surface and allows the system to recover real-world spatial relationships from the perspective-distorted image.

The matrix is computed using the `cv2.getPerspectiveTransform` function in OpenCV, which takes as input two sets of four points (quadrilaterals): one set corresponding to the vertices of the SUBZONE in the image, and the other defining the corners of a fixed rectangular area in the transformed (target) space [66]. In the implemented system, the target region is a rectangular area, whose dimensions correspond to the actual physical dimensions of the road segment used for speed estimation.

In the computer vision pipeline script, this logic is encapsulated in the `ViewTransformer` class, which computes the homography matrix upon initialization and provides a

method to apply it to arbitrary image coordinates. The source and target quadrilaterals are represented as NumPy arrays, where each row corresponds to the coordinates of a vertex. These arrays are passed to `getPerspectiveTransform` to compute the homography matrix. The `transform_points` method then applies this matrix using OpenCV's `perspectiveTransform`, effectively projecting detection points from the image plane to the rectified coordinate system [67].

It is important however to note that the source and target quadrilaterals must be carefully selected and calibrated for each individual camera view. Any change in camera height, tilt, or orientation requires recalibrating these zones and recomputing the transformation matrix [67]. The results of the perspective transformation on the specified zone are presented in Figure 28.

Figure 28: Perspective transformation

This calibration ensures that pixel coordinates within the SUBZONE, when transformed correspond to metric distances in the TARGET space. The resulting transformed y-coordinates are then used to infer motion along the vertical axis of the rectified plane, allowing for direct interpretation of motion in meters and speeds in km/h.

To compute speed, the pipeline tracks the movement of each vehicle’s anchor point across frames. For each unique vehicle (as identified by the `tracker_id`), a temporal history of its transformed y-coordinates is stored in a fixed-length `deque`. This deque allows efficient insertion of new coordinates while maintaining a sliding window of past

positions. Once a sufficient number of data points is collected for a vehicle, the vertical displacement (Δ_y) is calculated as the difference between its earliest and latest y-coordinates in the deque. The time interval (Δ_t) is derived from the number of frames covered, divided by the video frame rate. The instantaneous speed is then estimated using the equation:

$$Speed\ (km/h)\ = \frac{\Delta_y}{\Delta_t} \cdot 3.6$$

where Δ_y is the vertical distance in the target space, Δ_t is the time elapsed in seconds, and the factor 3.6 converts meters per second to kilometers per hour. To ensure stability and accuracy in speed estimation, the system delays computation until each vehicle's deque contains a sufficient number of coordinate samples. This approach addresses a common issue in object detection known as bounding box flickering, which is a subtle frame-to-frame jitter in objects detection outputs that, while often not noticeable to the human eye, can significantly distort speed measurements when speed is calculated over very short time intervals. By accumulating position data over approximately one second of footage, the system effectively smooths out these micro-movements. As a result, the measured displacement reflects meaningful vehicle motion rather than noise introduced by detection variability, yielding more accurate speed estimates [67].

Figure 29 illustrates the output of the speed estimation module, where detected vehicles are visualized with bounding boxes and annotations that indicate the assigned `tracker_id`, their class label, and their estimated speed. As specified, speed is calculated only for vehicles inside the SUBZONE (green trapezoid).

Figure 29: Speed estimation within SUBZONE

While the speed estimation module performs reliably under controlled conditions, several practical challenges may arise when applying this technique in real-world deployments. These factors can significantly affect speed estimation accuracy and must be accounted for during system design and calibration.

To begin with, the effectiveness of the adopted speed estimation method relies heavily on the positioning of the camera relative to the monitored traffic zone. In the context of the proposed system, the camera is installed in a fixed, elevated position opposite the direction of vehicle movement, creating a favorable viewpoint for applying a homography-based perspective transformation. This setup allows the system to simplify vehicle motion analysis by focusing primarily on displacement along the vertical axis of the image plane, effectively ignoring lateral (x-axis) movement. Under these conditions, the vertical displacement of anchor points across frames can be reliably mapped to real-world distances using a calibrated region of interest. However, this simplification becomes inadequate in more complex traffic scenarios, particularly when vehicles exhibit diagonal or lateral movement within the frame. In such cases, the current model's perspective assumptions lead to projection errors and reduced accuracy in speed estimation. Addressing these limitations requires more advanced techniques, such as optical flow-based motion analysis, multi-camera calibration methods that incorporate depth estimation, or 3D reconstruction with stereo vision, all of which offer more precise handling of arbitrary motion vectors and diverse camera geometries. However, they also introduce significant computational overhead, making them less suitable for lightweight system deployments.

Another key consideration revolves around occlusions and incomplete detections. The stability of the bounding box around a vehicle is critical for accurate speed estimation. Temporary occlusions, such as one vehicle passing in front of another, can distort the detected size or position of the bounding box. Even minor inconsistencies across frames may introduce substantial errors in the estimated speed, especially when these changes occur within a short temporal window.

One more challenge concerns the reliability of anchor points under varying environmental conditions. The system uses the bottom-center of each bounding box as a spatial anchor for tracking and speed computation, a method that works well under favorable conditions such as daylight and clear weather. However, under adverse

conditions like rain, fog, or low-light environments, consistently identifying this anchor point becomes more difficult, potentially degrading overall accuracy.

Lastly, the assumption of a flat road surface may not hold true in real deployments. Road inclination can alter the perceived movement of vehicles within the image plane, distorting speed measurements if unaccounted for. To mitigate this, speed estimation should ideally be limited to road segments that are relatively level, or alternatively, road slope information should be incorporated into the geometric transformation model [67].

5.6 Detected Vehicles Counting

In addition to detection, tracking, and speed estimation, the computer vision pipeline incorporates a line-based vehicle counting mechanism designed to quantify directional traffic flow. This functionality is implemented using the `LineZone` class provided by the `supervision` library, which allows the definition of a virtual line within the video frame to monitor object crossings. Vehicles that intersect this line are counted in real-time, enabling the system to distinguish between inbound and outbound movements relative to a predefined orientation.

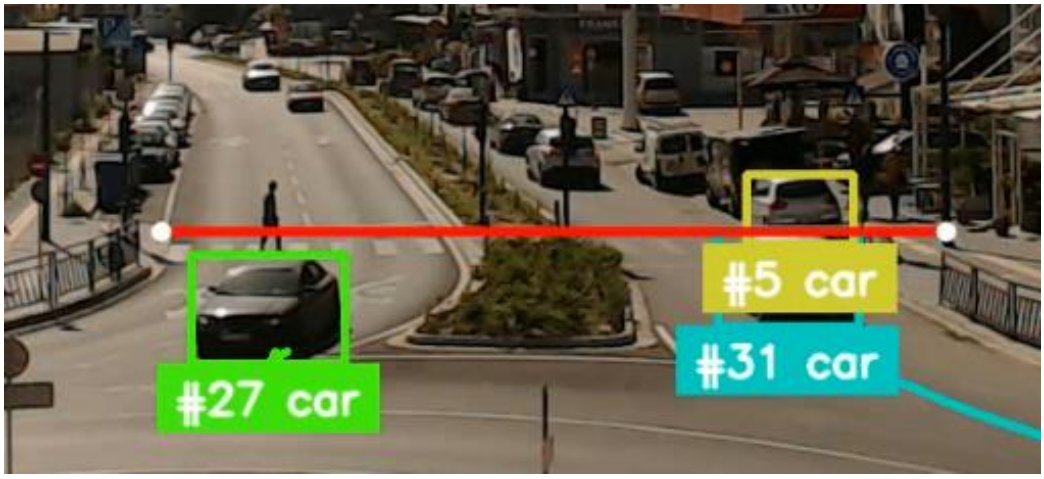


Figure 30: Virtual counting line

The virtual counting line is defined programmaticaly via the `LineZone` constructor, which takes two endopoinds (`LINE_START` and `LINE_END`) that span the desired counting region. These endpoints are specified in pixel coordinates, and in the current setup, the line is drawn horizontally across the road lane of interest, as illustrated in Figure 30. The line direction is used to infer whether vehicles are entering or leaving a zone (e.g., a roundabout). The line crossing logic is based on the analysis of anchor point trajectories,

which are updated frame-by-frame via the tracking module. The `triggering_anchors` parameter is set to `sv.Position.CENTER`, meaning the center point of the object's bounding box is used to detect crossing events [68].

As tracked detections are updated frame-by-frame, the pipeline invokes the `trigger()` method of the `LineZone` to evaluate which objects have crossed the line. The line zone maintains internal counters: `in_count` and `out_count`, as well as `in_count_per_class` and `out_count_per_class`, which allow directional and class-wise accumulation. Each vehicle is counted only once per direction based on its `tracker_id`, preventing duplication due to overlapping frames or momentary occlusions.

These counts are stored in a hierarchical data structure under the `_internal_data["traffic_analytics"]` dictionary. The counts are separated into `entering_roundabout` and `leaving_roundabout` categories, each of which includes the total number of vehicles and a breakdown by class using YOLO class names. This is performed by the `update_results()` function, which aggregates the current state of the `LineZone` and prepares the structured JSON output accordingly.

An additional metric derived from counting logic is traffic intensity, defined as the number of vehicles passing through the monitored area per hour. The pipeline computes this value dynamically at the end of each video processing cycle using both real-time observations and recent historical data to extract a smooth and context-aware result.

The baseline estimate (`default_intensity`) is calculated by summing the number of vehicles that crossed the virtual line in either direction (in_count +out_count) and scaling that count to an hourly rate. Given that the processed videos represent one minute of traffic activity, the vehicle count is multiplied by a factor of 60:

$$Default\ Intensity = (in\_count + out\_count) \cdot 60$$

However, to improve the robustness of this estimation and account for short-term fluctuations, outliers, or anomalies (e.g., false positives, sudden occlusions, or dropped frames), the system attempts to retrieve previously stored intensity measurements from the time-series database. This is achieved by querying the InfluxDB database using the `influx_fetch.fetch_intensity()` function, which retrieves previous intensity values recorded up to 1 hour prior to the current timestamp.

If at least three historical measurements are found, the system constructs a new set of values consisting of:

- The three most recent past intensity values (if more than three are available, only the last three are used), and
- The newly computed default intensity from the current video.

The average of this combined set is then calculated and rounded to the nearest integer:

$$Smoothed\ Intensity\ =\ round\left(\frac{\sum(recent\ values\ +\ current)}{N}\right)$$

If fewer than three historical values are available, the system defaults to using the instantaneous estimate only, which ensures continued functionality in cases of limited data or initial deployment.

Finally, the computed intensity is assigned to the `results["object"]["intensity"]` field in the output JSON. This value also serves as the basis for determining the congestion status. If the resulting intensity exceeds a predefined threshold (e.g., 1000 vehicles/hour), the congested flag is set to 1 to indicate high traffic load; otherwise, it remains 0. This simple congestion detection mechanism provides a simple but effective method to assess traffic conditions in near real-time.

5.7 Computer Vision Data Annotation

To enhance interpretability and support both debugging and visual reporting, the pipeline's computer vision script includes comprehensive annotations that overlay detection and tracking results directly onto the original video frames. This process provides a visual summary of the system's inference at each frame and serves as a valuable tool for verification, monitoring, and system demonstrations. The annotation functionality is built using `supervision`, which offers a variety of modular annotators for drawing bounding boxes, labels, trajectory traces, and incorporating custom counters directly onto video frames.

Each frame processed by the pipeline is annotated in multiple layers. The first layer consists of bounding boxes, which are drawn around detected vehicles using the `BoxAnnotator` class. The bounding boxes are color-coded rectangles based on the `tracker_id`, allowing for clear visual differentiation between tracked instances. This highlights which objects have been detected, localized, and persistently tracked by the model and tracker [60].

Next, the pipeline overlays trajectory traces, which represent the recent motion history of each tracked object. Implemented via the `TraceAnnotator`, this layer draws curved or

linear trails behind each object to reflect recent movement across a configurable number of frames. The trace duration is set relative to the video frame rate [58]. These visual traces help assess object behavior over time and are useful for evaluating tracking consistency.

The system also overlays text labels on each bounding box using the `LabelAnnotator` [60]. These labels display the vehicle's tracking ID, class name, and, where available, the estimated speed in kilometers per hour. These values are dynamically updated as the video progresses. Labels are positioned beneath each bounding box and rendered with adaptive scaling and thickness settings to remain legible at various video resolutions.

In addition to per-object annotations, a multiclass vehicle counter is incorporated in the upper right corner of the frame. This component is implemented using the `CustomCounterAnnotator`, a custom extension of supervision's `LineZoneAnnotatorMulticlass`, which displays a summary of the number of vehicles detected per class and per direction (e.g., entering or leaving a zone) [60]. It aggregates the current state of the `LineZone` and provides a compact visual summary of traffic flow statistics on the frame itself.

Figure 31: Annotated traffic monitoring system

Figure 31 illustrates the annotation functionality, showing color-coded bounding boxes around detected vehicles with unique tracker IDs, trajectory traces indicating recent movement traces, estimated speed information overlaid on labels, class-specific vehicle

identification, and a real-time vehicle counter displaying traffic flow data for vehicles entering and leaving the roundabout.

All annotations are rendered on a working copy of each video frame, preserving the original data for further processing or archival. The annotated frames are passed to a `cv2.VideoWriter` instance, configured to encode the output using the mp4v codec and to match the resolution and frame rate of the input video. This ensures smooth playback and temporal synchronization in the resulting video [33].

The pipeline also supports real-time visualization using OpenCV's `cv2.imshow` function [33]. This feature displays annotated frames live in a dedicated window during execution, allowing users to verify detection accuracy, inspect visual artifacts, and monitor system behavior as it runs. The display loop includes keyboard-based termination, enabling runtime control over the execution process.

5.8 Result Serialization and Data Distribution

Upon completion of video processing and metric computation, the computer vision pipeline performs result serialization and distributes the outputs to external systems. This final stage ensures that the inference results are both persistently stored for long-term analysis and immediately available for real-time consumption. The pipeline achieves this through a dual-channel data distribution mechanism: exporting a structured JSON file and publishing the results to an MQTT broker and an InfluxDB time-series database.

All traffic metrics produced by the pipeline are first consolidated [69] into a structured Python dictionary called `results`. This dictionary is designed to closely mirror the schema of a `TrafficFlowObserved` entity, as defined in the FIWARE Smart Data Models initiative for the Transportation domain. The FIWARE model provides a standardized schema for describing traffic flow observations in a way that is semantically interoperable and suitable for use in NGSI-LD–compliant smart city systems.

The `results` object includes several core attributes recommended by the data model. The `time` field corresponds to the `dateObserved` attribute, indicating the exact timestamp of the captured traffic conditions. The `deviceInfo` dictionary contains identifiers that link the observation to a specific processing node (e.g., `TrafficFlowObserved-Larissa`). Geospatial metadata is included in the `rxInfo` structure, which aligns with the `location` property defined in the model, specifying coordinates as a GeoJSON-like point with latitude and longitude values.

The `object` section encapsulates the core traffic metrics and is structured according to the key attributes defined by the `TrafficFlowObserved` entity, including:"

- `intensity`: the total number of vehicles detected during the observation period.
- `averageVehicleSpeed`: the computed average speed of all vehicles detected during the observation period.
- `congested`: a Boolean flag indicating whether there was a traffic congestion during the observation period.
- `address`: contains `streetAddress`, `addressLocality`, and `addressCountry`, providing geographic context. [70]

The `results` dictionary is then serialized into a JSON-formatted file using Python's built-in `json` module [71]. The output file is saved as `*Traffic_Flow_Observed.json*` and written to the processing unit's local disk. Once generated, the JSON file is distributed through two independent channels. The first is real-time publication via MQTT (Message Queuing Telemetry Transport), a lightweight publish-subscribe messaging protocol commonly used in IoT systems. The `publish_json_to_mqtt()` function handles this process by reading the JSON file and publishing its content as a payload to a preconfigured topic on an MQTT broker. To ensure secure transmission, the connection is established over TLS (Transport Layer Security), with certificate-based authentication [72]. Specifically, both the broker and the clients are configured with X.509 certificates issued by a trusted Certificate Authority (CA), enabling mutual authentication and encrypted message exchange. This is critical in the context of traffic analytics, where sensitive real-time data such as vehicle speeds and congestion indicators must be protected against interception or manipulation. By employing TLS-secured MQTT, the system guarantees that only authorized clients can publish or subscribe to traffic data topics, while ensuring that the transmitted data remains confidential and tamper-proof during transit.

The MQTT message is sent with a Quality of Service (QoS) level of 1, which guarantees that the message is delivered at least once. This mechanism makes the results instantly accessible to any subscribed client, including FIWARE IoT agents and dashboards.

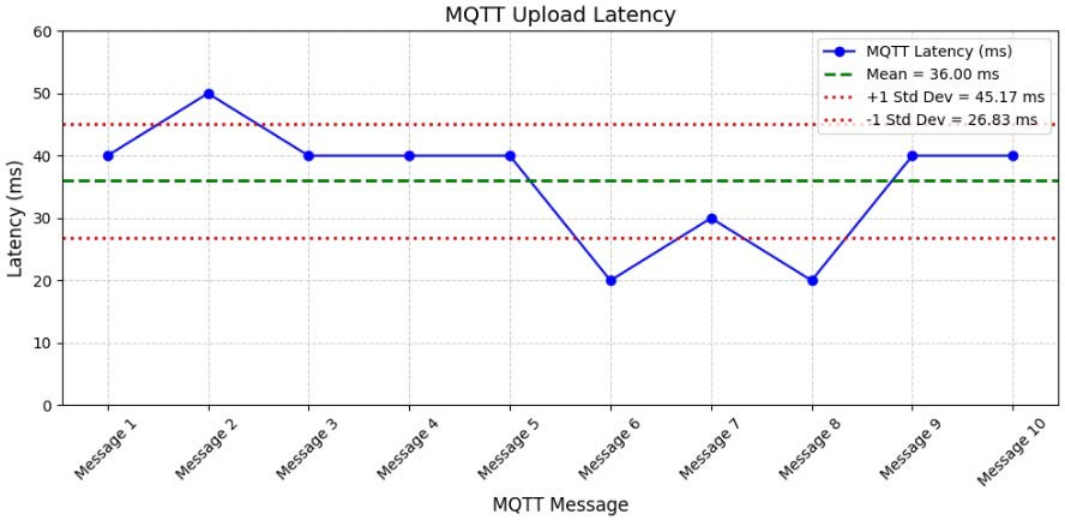


Figure 32: MQTT messages upload latency

To assess the communication performance of the messaging subsystem, the MQTT upload latency was measured. This latency is defined as the time required to publish a telemetry message from the central processor to the MQTT broker. As shown in Figure 32, the MQTT latencies exhibit extremely low values, ranging from 20 to 50 ms, with a mean latency of 36 ms. This consistency confirms that the use of MQTT as a lightweight messaging protocol is well-suited for low-latency, real-time telemetry transmission in the proposed system architecture. The minimal variation further supports the reliability of MQTT for time-sensitive communication between system components.

The second distribution channel involves historical storage in InfluxDB, a high-performance database optimized for handling time-series data. To evaluate the performance of the database access layer, the latency associated with both reading from and writing to the InfluxDB time-series database was measured. In the proposed architecture, once a video has been processed and vehicle counts have been computed, the system queries historical data from InfluxDB to estimate the traffic flow over the preceding hour. As Figure 33 shows, this fetch operation consistently exhibits low latency, typically ranging from 10 to 30 ms, with an average retrieval time of approximately 23 ms. Following this, the computed traffic metrics are written to the database, with write latencies ranging from 40 to 50 ms and a mean write time of 46 ms. These results indicate that InfluxDB read/write operations impose minimal overhead, enabling the system to perform time-sensitive, data-driven computations without delay.

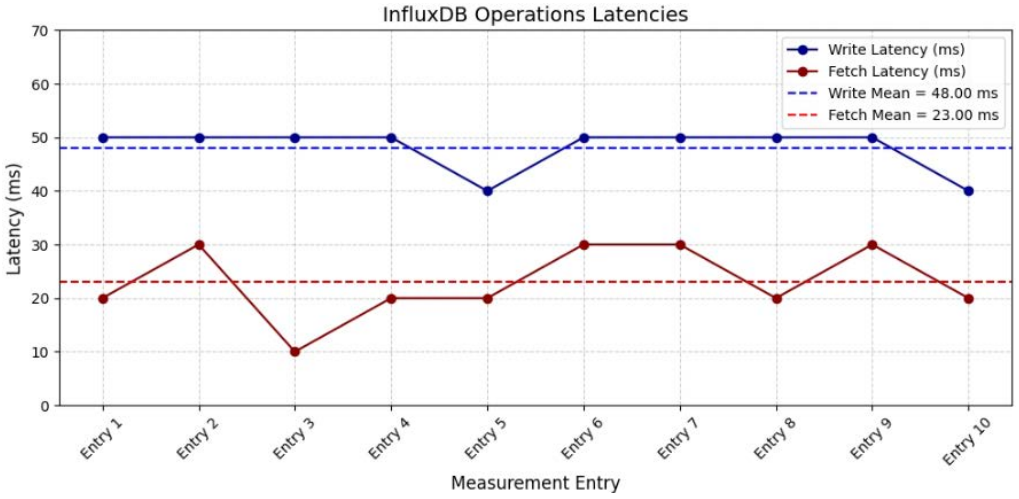


Figure 33: Database operations latencies

Functionally, the `influx_write.write_to_influx()` function parses the same JSON file and reformats its contents into one or more data points before writing them to the database [73]. This enables the system to maintain a structured historical record of traffic observations, supporting time-based windowed queries, statistical trend analysis, and external visualization. To facilitate this, a custom Grafana dashboard was created and connected to InfluxDB to display key traffic metrics such as hourly vehicle flow and average speed [74]. The dashboard, as shown in Figure 34, includes time-series graphs and gauge indicators that provide intuitive, real-time insights into traffic conditions and support retrospective analysis of urban mobility patterns.

Figure 34: Traffic monitoring system’s Grafana dashboard

This dual-distribution design ensures decoupling between the perception layer (video analysis) and the application or storage layers. Such decoupling enhances system reliability and modularity, allowing either output channel to operate independently or be reconfigured without impacting the core inference logic.

5.9 End-to-End System Performance Analysis

To assess the overall responsiveness of the proposed architecture, the end-to-end latency, defined as the total time required for the complete execution of the system pipeline, was-measured. As demonstrated in Figure 35, end-to-end latencies range from 80.81 to 83.62 seconds, with a mean of 81.88 seconds. The values are tightly clustered, with most falling within a narrow band around the mean, indicating high predictability across the full system pipeline. This level of consistency suggests that neither the network layer nor the computational stages introduce significant variability under normal operating conditions. As such, the measured end-to-end latency confirms that the integrated architecture achieves stable and timely performance, validating its suitability for near-real-time traffic monitoring and its readiness for deployment in real-world environments.

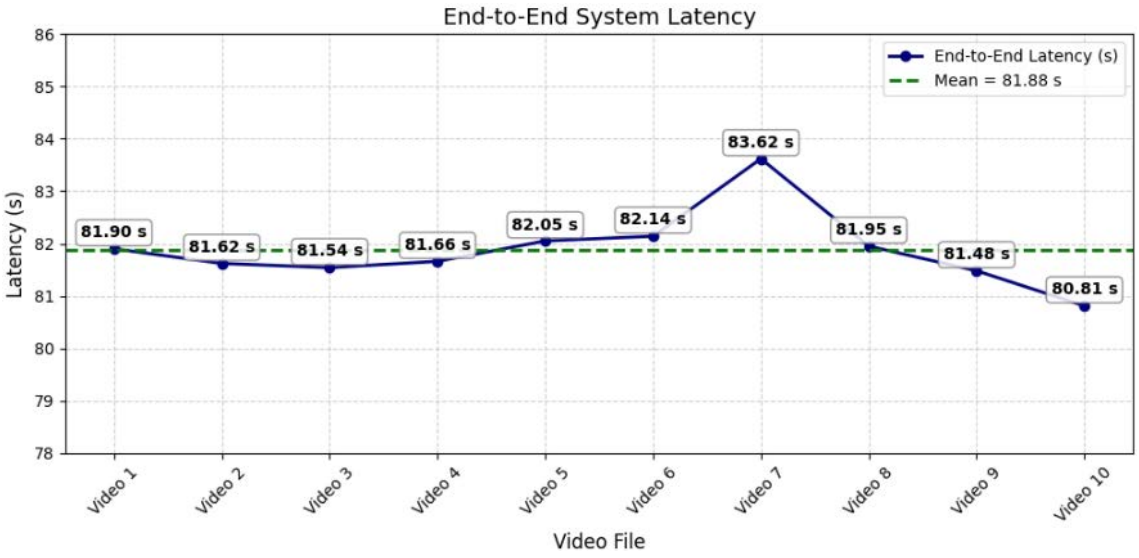


Figure 35: End-to-end system latency

To evaluate the efficiency of the proposed computer vision pipeline in terms of computational resource usage, the average CPU and RAM utilization during the inference process on the central processing unit was monitored. The processing system is a general-purpose desktop PC without discrete GPU acceleration, equipped with an AMD Ryzen 5 5600G processor (6-core, 12-thread, with integrated graphics), 16 GB DDR4 RAM, and running the Windows operating system.

The measurements illustrated in Figure 36 indicate that the pipeline imposes a moderate CPU load, utilizing just under two-thirds (64.72%) of the available processing capacity. This level of utilization suggests that the application efficiently leverages the par-

allelism provided by the processor's multi-core architecture, while still maintaining sufficient headroom for concurrent tasks. This ensures stable operation without risking CPU saturation.

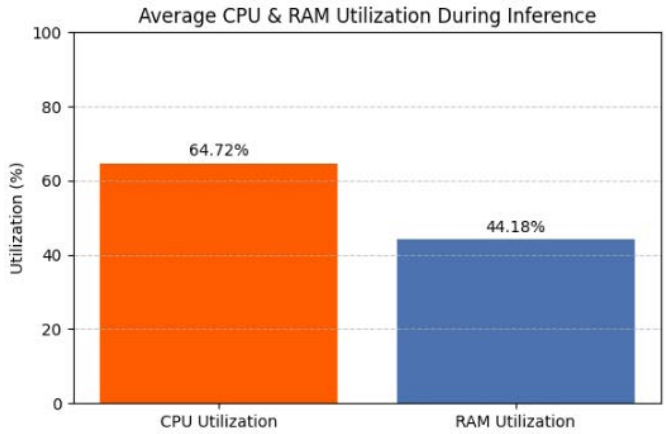


Figure 36: Average CPU and RAM utilization during inference

The RAM usage, remaining below 50%, reflects effective memory management within the application. This is important for long-running or continuously operating systems, where excessive or poorly managed memory usage could lead to performance degradation over time. The low memory footprint indicates that the pipeline is well-suited for deployment on systems with moderate specifications, without requiring high-end or specialized hardware.

Ultimately, the decision to rely solely on CPU-based inference, without the use of discrete GPUs, underscores the lightweight nature of the application and its adaptability to cost-effective, scalable deployments. These findings confirm the system's ability to operate reliably within resource-constrained environments while maintaining consistent processing performance.

6 Conclusions

This thesis focused on the design and deployment of a fully automated, headless IoT system for traffic monitoring, integrating edge computing, cloud infrastructure, and computer vision technologies into a functional and scalable architecture. The system was built with the goal of operating autonomously in real-world conditions, without requiring manual intervention and specialized hardware.

The system was implemented using a decoupled edge-to-cloud, event-driven approach. An embedded edge device (Raspberry Pi) recorded traffic videos at scheduled intervals and uploaded them to Google Drive. A webhook mechanism triggered a centralized processing pipeline, which executed vehicle detection, classification, tracking, speed estimation, and congestion analysis using a pre-trained YOLO model. The extracted traffic metrics were serialized into a structured JSON format, published via MQTT, and stored in a time-series database for historical analysis and visualization through Grafana.

One of the main challenges was ensuring robust operation in environments with limited connectivity and hardware capabilities. The decision to avoid real-time streaming and instead work with discrete video segments proved to be an effective trade-off, allowing for simpler implementation, better resource utilization, and more structured processing. Additional difficulties included integrating the Google Drive API and maintaining persistent webhook communication, both of which required careful handling of authentication and retry mechanisms. These challenges were ultimately addressed through scripting automation, custom scheduling, and the use of lightweight libraries that suited the system's constraints.

Overall, the system demonstrated stable end-to-end performance with low variance across all components. Evaluation showed consistent processing times and efficient resource usage, confirming the system's suitability for constrained, cost-effective deployments.

Future work could focus on two key directions: first, enhancing detection accuracy by training a custom object detection model tailored to urban traffic scenes using context-aware features and localized data, and second, to migrate the processing pipeline to GPU-

accelerated hardware to reduce processing latency and support more complex models. These steps aim to enhance scalability, enable real-time performance across multiple streams, and strengthen integration into broader smart city infrastructures.

Bibliography

[1] "Edge Computing with Raspberry Pi," [Online]. Available: https://medium.com/@thebinayak/edge-computing-with-raspberry-pi-ec28912bc3df. [Accessed 23 6 2025].

[2] "What is edge computing?," [Online]. Available: https://www.ibm.com/think/topics/edge-computing. [Accessed 23 6 2025].

[3] "Understanding the Synergy between IoT and Edge Computing: Unlocking New Horizons of Connectivity and Efficiency," [Online]. Available: https://hashstudioz.medium.com/understanding-the-synergy-between-iot-and-edge-computing-unlocking-new-horizons-of-connectivity-32c9847a0a5a. [Accessed 23 6 2025].

[4] "Smart Traffic Management Systems: IoT Solutions for Urban Mobility," [Online]. Available: https://www.aeologic.com/blog/smart-traffic-management-systems-iot-solutions-for-urban-mobility/. [Accessed 23 6 2025].

[5] "Affordable IoT Solutions for Urban Mobility Challenges," Medium, [Online]. Available: https://medium.com/%40AllGEOGRAPHY/affordable-iot-solutions-for-urban-mobility-challenges-6248487563b4. [Accessed 23 6 2025].

[6] "Ultralytics Glossary - Computer Vision," [Online]. Available: https://www.ultralytics.com/glossary/computer-vision-cv. [Accessed 23 6 2025].

[7] "Enhancing Traffic Management with Computer Vision: Applications and Benefits," [Online]. Available: https://www.augmentedstartups.com/blog/enhancing-traffic-management-with-computer-vision-applications-and-benefits?srsltid=AfmBOoqXzvsAJ6HS3oQa3YdA5laSWYRHi5jRizPW0IAkH1QfL8WAUjPQ. [Accessed 23 6 2025].

[8] A. Drozdov, "Computer Vision for Traffic Management and Analysis," [Online]. Available: https://yellow.systems/blog/computer-vision-for-traffic-management. [Accessed 23 6 2025].

[9] R. Kundu, "YOLO: Algorithm for Object Detection Explained [+Examples]," V7, [Online]. Available: https://www.v7labs.com/blog/yolo-object-detection. [Accessed 23 6 2025].

[10] "Performance Metrics Deep Dive," [Online]. Available: https://docs.ultralytics.com/guides/yolo-performance-metrics/. [Accessed 23 6 2025].

[11] "Ultralytics Glossary - Intersection over Union (IoU)," [Online]. Available: https://www.ultralytics.com/glossary/intersection-over-union-iou. [Accessed 23 6 2025].

[12] "Ultralytics Glossary - Mean Average Precision (mAP)," [Online]. Available: https://www.ultralytics.com/glossary/mean-average-precision-map. [Accessed 6 23 2025].

[13] J. Ebner, "The Best Way to Plot a Precision-Recall Curve in Python," [Online]. Available: https://sharpsight.ai/blog/plot-precision-recall-curve-in-python/. [Accessed 23 6 2025].

[14] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] R. Khanam and M. Hussain, "YOLOV11: AN OVERVIEW OF THE KEY ARCHITECTURAL," 2024.

[16] Z. Keita, "YOLO Object Detection Explained," [Online]. Available: https://www.datacamp.com/blog/yolo-object-detection-explained. [Accessed 23 6 2025].

[17] J. Gallagher, "What is YOLOv11? An Introduction," [Online]. Available: What is YOLOv11? An Introduction. [Accessed 23 6 2025].

[18] A. Ghosh, "YOLO11: Redefining Real-Time Object Detection," [Online]. Available: https://learnopencv.com/yolo11/#aioseo-yolo11-architecture-and-whats-new-in-yolo11. [Accessed 23 6 2025].

[19] S. N. Rao, "YOLOv11 Architecture Explained: Next-Level Object Detection with Enhanced Speed and Accuracy," [Online]. Available: https://medium.com/@nikhil-rao-20/yolov11-explained-next-level-object-detection-with-enhanced-speed-and-accuracy-2dbe2d376f71. [Accessed 6 23 2025].

[20] "COCO Dataset," [Online]. Available: https://docs.ultralytics.com/datasets/detect/coco/. [Accessed 23 6 2025].

[21] D. Shah, "COCO Dataset: All You Need to Know to Get Started," [Online]. Available: https://www.v7labs.com/blog/coco-dataset-guide. [Accessed 6 23 2025].

[22] "What is High Level Design? – Learn System Design," [Online]. Available: https://www.geeksforgeeks.org/system-design/what-is-high-level-design-learn-system-design/. [Accessed 6 23 2025].

[23] "What is Edge to Cloud?," [Online]. Available: https://www.scalecomputing.com/resources/edge-to-cloud-computing-integration. [Accessed 6 23 2025].

[24] "What is Edge to Cloud?," [Online]. Available: https://www.hpe.com/emea_europe/en/what-is/edge-to-cloud.html. [Accessed 23 6 2025].

[25] "What is event-driven architecture?," [Online]. Available: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture. [Accessed 23 6 2025].

[26] "Advantages of the event-driven architecture pattern," [Online]. Available: https://developer.ibm.com/articles/advantages-of-an-event-driven-architecture/. [Accessed 23 6 2025].

[27] "What is event-driven architecture?," [Online]. Available: https://www.ibm.com/think/topics/event-driven-architecture. [Accessed 23 6 2025].

[28] "Raspberry Pi 4 Tech Specs," [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. [Accessed 23 6 2025].

[29] "C922 PRO HD STREAM WEBCAM," [Online]. Available: https://www.logitech.com/el-gr/shop/p/c922-pro-stream-webcam.960-001088. [Accessed 23 6 2025].

[30] "Raspberry Pi Connect," [Online]. Available: https://www.raspberrypi.com/documentation/services/connect.html. [Accessed 23 6 2025].

[31] "Raspberry Pi OS," [Online]. Available: https://www.raspberrypi.com/documentation/computers/os.html. [Accessed 23 6 2025].

[32] "What Is a Cron Job: Understanding Cron Syntax and How to Configure Cron Jobs," [Online]. Available: https://www.hostinger.com/tutorials/cron-job. [Accessed 23 6 2025].

[33] "Reading and Writing Videos using OpenCV," [Online]. Available: https://opencv.org/blog/reading-and-writing-videos-using-opencv/. [Accessed 23 6 2025].

[34] "os.path — Common pathname manipulations," [Online]. Available: https://docs.python.org/3/library/os.path.html.

[35] "Google Drive API overview," [Online]. Available: https://developers.google.com/workspace/drive/api/guides/about-sdk.

[36] "Configure the OAuth consent screen and choose scopes," Google Workspace, [Online]. Available: https://developers.google.com/workspace/guides/configure-oauth-consent.

[37] "OAuth 2.0," [Online]. Available: https://oauth.net/2/.

[38] "PyDrive2 documentation," [Online]. Available: https://docs.iterative.ai/PyDrive2/.

[39] "Create and manage files," [Online]. Available: https://developers.google.com/workspace/drive/api/guides/create-file.

[40] "Choose Google Drive API scopes," [Online]. Available: https://developers.google.com/workspace/drive/api/guides/api-specific-auth.

[41] "Manage OAuth Clients," [Online]. Available: https://support.google.com/cloud/answer/15549257?visit_id=638842899673442942-3873846125&rd=1.

[42] "Raspberry Pi 4 specs and benchmarks," [Online]. Available: https://magazine.raspberrypi.com/articles/raspberry-pi-4-specs-benchmarks. [Accessed 23 6 2025].

[43] "Upload file data," [Online]. Available: https://developers.google.com/workspace/drive/api/guides/manage-uploads. [Accessed 23 6 2025].

[44] "What is a webhook?," [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-a-webhook. [Accessed 23 6 2025].

[45] "Notifications for resource changes," [Online]. Available: https://developers.google.com/workspace/drive/api/guides/push. [Accessed 23 6 2025].

[46] S. Panker, "Ngrok," [Online]. Available: https://surajpanker82.medium.com/ngrok-96fd33ca869c. [Accessed 23 6 2025].

[47] L. Gupta, "Idempotent REST API," [Online]. Available: https://restfulapi.net/idempotent-rest-apis/. [Accessed 23 6 2025].

[48] "NSSM - the Non-Sucking Service Manager," [Online]. Available: https://nssm.cc/.

[49] "Service accounts overview," [Online]. Available: https://cloud.google.com/iam/docs/service-account-overview.

[50] "Using OAuth 2.0 for Server to Server Applications," [Online]. Available: https://developers.google.com/identity/protocols/oauth2/service-account#overview.

[51] "Roles and permissions," [Online]. Available: https://cloud.google.com/iam/docs/roles-overview.

[52] "google.oauth2.service_account module," [Online]. Available: https://google-auth.readthedocs.io/en/latest/reference/google.oauth2.service_account.html.

[53] "Service account credentials," [Online]. Available: https://cloud.google.com/iam/docs/service-account-creds.

[54] "Built-in Exceptions," [Online]. Available: https://docs.python.org/3/library/exceptions.html.

[55] "argparse — Parser for command-line options, arguments and subcommands," [Online]. Available: https://docs.python.org/3/library/argparse.html.

[56] "re — Regular expression operations," [Online]. Available: https://docs.python.org/3/library/re.html.

[57] "ISO 8601 - Date and Time Format," [Online]. Available: https://www.iso.org/iso-8601-date-and-time-format.html.

[58] "Supervision," [Online]. Available: https://supervision.roboflow.com/latest/.

[59] "Ultralytics Glossary - Anchor-Based Detectors," [Online]. Available: https://www.ultralytics.com/glossary/anchor-based-detectors. [Accessed 23 6 2025].

[60] "Supervision - Annotators," [Online]. Available: https://supervision.roboflow.com/annotators/.

[61] "Supervision - Polygon Zone," [Online]. Available: https://supervision.roboflow.com/detection/tools/polygon_zone/.

[62] "collections — Container datatypes," [Online]. Available: https://docs.python.org/3/library/collections.html.

[63] "Supervision - Video," [Online]. Available: https://supervision.roboflow.com/utils/video/.

[64] "Supervision - Trackers," [Online]. Available: https://supervision.roboflow.com/trackers/.

[65] "Perspective Transformation," [Online]. Available: https://www.sciencedirect.com/topics/computer-science/perspective-transformation#:~:text=A%20perspective%20transformation%20is%20a,and%20photographs%20taken%20with%20lenses.

[66] "Geometric Image Transformations," [Online]. Available: https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html.

[67] P. Skalski, "How to Estimate Speed with Computer Vision," [Online]. Available: https://blog.roboflow.com/estimate-speed-computer-vision/. [Accessed 23 6 2025].

[68] "Supervision - Line Zone," [Online]. Available: https://supervision.roboflow.com/detection/tools/line_zone/.

[69] P. Powell and I. Smalley, "What is data consolidation?," [Online]. Available: https://www.ibm.com/think/topics/data-consolidation. [Accessed 23 6 2025].

[70] "FIWARE Smart Data Models - Entity: TrafficFlowObserved".

[71] "json — JSON encoder and decoder," [Online]. Available: https://docs.python.org/3/library/json.html.

[72] "Mosquitto MQTT Broker SSL Configuration Using Own Certificates," [Online]. Available: http://www.steves-internet-guide.com/mosquitto-tls/. [Accessed 23 6 2025].

[73] "influxdata Documentation - Python client library," [Online]. Available: https://docs.influxdata.com/influxdb/v2/api-guide/client-libraries/python/.

[74] "Get started with Grafana and InfluxDB," [Online]. Available: https://grafana.com/docs/grafana/latest/getting-started/get-started-grafana-influxdb/. [Accessed 23 6 2025].

Appendix A

Computer Vision Pipeline Script → *Traffic_Analytics.py*

```
import argparse
import glob
import os
import re
import json
from collections import defaultdict, deque
from datetime import datetime
import cv2
import numpy as np
import paho.mqtt.client as mqtt
import supervision as sv
from ultralytics import YOLO
import influx_config
import influx_fetch
import influx_write
from custom_counter_annotator import CustomCounterAnnotator

#MQTT Broker Configuration
BROKER = "[BROKER_ADDRESS]"
PORT = 8883
TOPIC = "[TOPIC_ID]"
TLS_CA_CERT = "[PATH_TO_CA_CERT]"
TLS_CLIENT_CERT = "[PATH_TO_CLIENT_CERT]"
TLS_CLIENT_KEY = "[PATH_TO_CLIENT_KEY]"
USE_TLS = True  # Set to True when using port 8883 (TLS)


def publish_json_to_mqtt(json_path):
    """Publish the JSON file to the specified MQTT topic."""
    client = mqtt.Client()
    client.on_connect = lambda client, userdata, flags, rc: None  #
Suppress connection logging
    client.tls_set(TLS_CA_CERT, TLS_CLIENT_CERT, TLS_CLIENT_KEY)
    client.connect(BROKER, PORT)

    with open(json_path, 'r') as json_file:
        json_data = json_file.read()

    client.publish(TOPIC, json_data, qos=1)
    client.disconnect()


# Main detection polygon
SOURCE = np.array([[835, 620], [1022, 620], [1700, 860], [640, 860]])

# Subzone polygon (a subarea within SOURCE for speed estimation)
SUBZONE = np.array([[836, 625], [944, 625], [1030, 735], [760, 735]])

# Road Length in the target (transformed) space
TARGET_WIDTH = 6
TARGET_HEIGHT = 40
```

```
# Perspective Transformation target points (for speed estimation)
TARGET = np.array(
    [
        [0, 0],
        [TARGET_WIDTH - 1, 0],
        [TARGET_WIDTH - 1, TARGET_HEIGHT - 1],
        [0, TARGET_HEIGHT - 1]
    ]
)

CLASSES = [1, 2, 3, 5, 7]


class ViewTransformer:
    def __init__(self, source: np.ndarray, target: np.ndarray):
        source = source.astype(np.float32)
        target = target.astype(np.float32)
        self.m = cv2.getPerspectiveTransform(source, target)

    def transform_points(self, points: np.ndarray) -> np.ndarray:
        if points.size == 0:  # Check if the array is empty
            return np.array([])  # Return an empty array if there are
no points to transform

        reshaped_points = points.reshape(-1, 1, 2).astype(np.float32)
        transformed_points = cv2.perspectiveTransform(reshaped_points,
self.m)
        return transformed_points.reshape(-1, 2)


# Define a line for vehicle counting
LINE_START = sv.Point(1260, 675)
LINE_END = sv.Point(820, 675)

LINE_ZONE = sv.LineZone(
    start=LINE_START,
    end=LINE_END,
    triggering_anchors=(sv.Position.CENTER,)
)

# Load YOLO model
model = YOLO("yolo11l.pt")

# Results dictionary for TrafficFlowObserved JSON output format
results = {
    "time": {},
    "deviceInfo": {
        "devEui": "[DEVICE_IDENTIFIER]"
    },
    "object": {
        "intensity": 0,
        "congested": 0,
        "averageVehicleSpeed": 0,
        "address":{
            "streetAddress": "Wimots",
            "addressLocality": "Larissa",
            "addressCountry": "GR"
        }
    },
    "rxInfo": [
```

```
        {
            "location": {
                "latitude": [LATITUDE],
                "longitude": [LONGITUDE]
            }
        }
    ]
}

# Hidden internal tracking dictionaries (not for JSON output)
_internal_data = {
    "average_speed_per_type": {},  # Dictionary to hold average speeds
per vehicle type
    "traffic_analytics": {
        "entering_roundanout": {
            "total_in": 0,
            "vehicle_type": {}  # Will hold counts for each vehicle
type
        },
        "leaving_roundanout": {
            "total_out": 0,
            "vehicle_type": {}  # Will hold counts for each vehicle
type
        },
    },
}

# Function to update results dictionary
def update_results():
    """Update total counts and class wise counts"""
    _internal_data["traffic_analytics"]["entering_roundanout"]["to-
tal_in"] = LINE_ZONE.in_count
    _internal_data["traffic_analytics"]["leaving_roundanout"]["to-
tal_out"] = LINE_ZONE.out_count

    _internal_data["traffic_analytics"]["entering_roundanout"]["vehi-
cle_type"] = {
        model.names[class_id]: count
        for class_id, count in LINE_ZONE.in_count_per_class.items()
    }
    _internal_data["traffic_analytics"]["leaving_roundanout"]["vehi-
cle_type"] = {
        model.names[class_id]: count
        for class_id, count in LINE_ZONE.out_count_per_class.items()
    }

    # Calculate vehicle counter
    total_in = LINE_ZONE.in_count
    total_out = LINE_ZONE.out_count
    vehicle_counter = total_in + total_out

    # Calculate a default intensity (fallback if historical data isn't
available)
    default_intensity = vehicle_counter * 60

    # Try to get historical data and compute a more sophisticated in-
tensity value
    try:
        # Get historical intensity values from InfluxDB
        historical_values = influx_fetch.fetch_intensity(re-
sults["time"])
```

```
        # If we have at least 4 measurements from the previous hour,
use their average
        if len(historical_values) >= 3:
            # Take up to the 4 most recent measurements
            recent_values = historical_values[-3:] if len(histori-
cal_values) >= 4 else historical_values
            # Include the current measurement in the average
            all_values = recent_values + [default_intensity]
            average_intensity = round(sum(all_values) / len(all_val-
ues)) #Round to integer
            print(f"Using historical-based intensity calculation: {av-
erage_intensity}")
            results["object"]["intensity"] = average_intensity
        else:
            # Not enough historical data, use the simple calculation
            print(f"Using default intensity calculation: {default_in-
tensity}")
            results["object"]["intensity"] = default_intensity
    except Exception as e:
        print(f"Error querying historical data: {e}")
        # Fallback to default calculation
        results["object"]["intensity"] = default_intensity

    # Update congested state based on the average intensity
    results["object"]["congested"] = int(results["object"]["inten-
sity"] > 1000)

    # Calculate average speeds per type (rounded to 2 decimal places)
    for vehicle_type, speeds in vehicle_speeds.items():
        if speeds:
            _internal_data["average_speed_per_type"][vehicle_type] =
round(
                sum(speeds) / len(speeds), 2
            )

    # Calculate overall average speed
    total_weighted_speed = 0
    total_vehicle_count = 0

    for vehicle_type, avg_speed in _internal_data["aver-
age_speed_per_type"].items():
        # Retrieve counts for the specific vehicle type
        count_in = _internal_data["traffic_analytics"]["entering_roun-
dabout"]["vehicle_type"].get(vehicle_type, 0)
        count_out = _internal_data["traffic_analytics"]["leaving_roun-
dabout"]["vehicle_type"].get(vehicle_type, 0)
        vehicle_count = count_in + count_out

        # Add to the weighted sum and total vehicle count
        total_weighted_speed += avg_speed * vehicle_count
        total_vehicle_count += vehicle_count

    results["object"]["averageVehicleSpeed"] = round(
        total_weighted_speed / total_vehicle_count, 2
    ) if total_vehicle_count > 0 else 0


def extract_timestamp(filename):
    """ Extract timestamp pattern from filename """
    pattern = r'record_(\d+)-(\d+)-(\d+)_(\d+)-(\d+)-(\d+)\.mp4'
    match = re.search(pattern, filename)
```

```
    if not match:
        raise ValueError(f"Unable to extract timestamp from filename:
{filename}")

    day, month, year, hour, minute, second = map(int, match.groups())
    # Create datetime object and format as ISO
    timestamp = datetime(year, month, day, hour, minute, second)
    return timestamp.strftime("%Y-%m-%dT%H:%M:%S")


DOWNLOAD_DIR = "./[VIDEO_DIRECTORY]"  # Directory where videos are
downloaded
DEFAULT_OUTPUT_VIDEO = "./annotated_result.mp4"  # Default output
video path


def get_latest_video(directory):
    """Retrieve the latest video file in the specified directory."""
    videos = sorted(glob.glob(os.path.join(directory, "*.mp4")),
key=os.path.getctime, reverse=True)
    return videos[0] if videos else None

if __name__ == "__main__":
    # Parse command-line arguments
    parser = argparse.ArgumentParser(description="Run YOLO vehicle de-
tection.")
    parser.add_argument("--video", type=str, help="Path to the source
video file.")
    parser.add_argument("--output", type=str, de-
fault=DEFAULT_OUTPUT_VIDEO, help="Path to save the output video.")
    args = parser.parse_args()

    # Determine the source video path
    if args.video:
        source_video_path = args.video
    else:
        source_video_path = get_latest_video(DOWNLOAD_DIR)
        if not source_video_path:
            raise FileNotFoundError("No videos found in the download
directory.")

    video_filename = os.path.basename(source_video_path)
    try:
        timestamp = extract_timestamp(video_filename)
        # Update the timestamp in the results dictionary
        results["time"] = timestamp
    except ValueError as e:
        print(f"Warning: {e}")
        print("Processing will continue but timestamp in results will
not be updated.")

    target_video_path = args.output

    # Initialize video details
    video_info = sv.VideoInfo.from_video_path(source_video_path)
    frame_generator = sv.get_video_frames_genera-
tor(source_path=source_video_path)

    # Initialize the perspective transformer for detection/tracking
(using SOURCE)
```

```
    view_transformer = ViewTransformer(source=SOURCE, target=TARGET)

    # Initialize a second perspective transformer for speed estimation
(using SUBZONE)
    subzone_transformer = ViewTransformer(source=SUBZONE, tar-
get=TARGET)

    # Polygon zone for perspective filtering (main detection zone uses
SOURCE)
    polygon_zone = sv.PolygonZone(SOURCE)

    # Initialize trackers and coordinates
    byte_track = sv.ByteTrack(frame_rate=video_info.fps)
    coordinates = defaultdict(lambda: deque(maxlen=video_info.fps))

    # Dictionary to store speeds at specific points per vehicle type
    vehicle_speeds = defaultdict(list)

    # Calculate optimal annotation sizes
    thickness = sv.calculate_optimal_line_thickness(resolu-
tion_wh=video_info.resolution_wh)
    text_scale = sv.calculate_optimal_text_scale(resolu-
tion_wh=video_info.resolution_wh)

    # Set up annotators
    bounding_box_annotator = sv.BoxAnnotator(
        thickness=thickness,
        color_lookup=sv.ColorLookup.TRACK
    )

    label_annotator = sv.LabelAnnotator(
        text_scale=0.8,
        text_thickness=2,
        text_position=sv.Position.BOTTOM_CENTER,
        color_lookup=sv.ColorLookup.TRACK
    )

    trace_annotator = sv.TraceAnnotator(
        thickness=thickness,
        trace_length=video_info.fps * 2,
        position=sv.Position.BOTTOM_CENTER,
        color_lookup=sv.ColorLookup.TRACK
    )

    counter_annotator_multiclass =  CustomCounterAnnotator(
        position=sv.Position.TOP_RIGHT,
        color=sv.Color.WHITE,
        text_color=sv.Color.BLACK,
        text_scale=0.75,
        text_thickness=1,
        padding=10,
        margin=20
    )

    line_zone_annotator = sv.LineZoneAnnotator(
        thickness=thickness,
        text_thickness=2,
        text_scale=text_scale,
        color=sv.Color.RED,  # Color of the line
        text_color=sv.Color.WHITE,  # Color of the text
        text_offset=50,  # Offset for the text from the line
```

```
    )

    # Set up the VideoWriter to save the annotated video
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Codec for mp4 format
    output_fps = video_info.fps  # Use the same FPS as the input video
    output_resolution = video_info.resolution_wh
    video_writer = cv2.VideoWriter(target_video_path, fourcc, out-
put_fps, output_resolution)

    for frame in frame_generator:
        if frame is None:  # Skip if no frame is available
            continue

        result = model(frame, verbose=False)[0]
        detections = sv.Detections.from_ultralytics(result)
        detections = detections[np.isin(detections.class_id, CLASSES)]
        detections = detections[polygon_zone.trigger(detections)]
        detections = byte_track.update_with_detections(detections=de-
tections)

        # Get anchor points (e.g. bottom-center) from detections
        # Note: For detection/tracking I use view_transformer, but for
speed estimation I apply the subzone perspective transformer.
        points = detections.get_anchors_coordinates(anchor=sv.Posi-
tion.BOTTOM_CENTER)
        points = subzone_transformer.trans-
form_points(points=points).astype(int)

        labels = []
        for tracker_id, point, class_id in zip(detections.tracker_id,
points, detections.class_id):
            vehicle_type = model.names[class_id]  # Get vehicle type
based on class_id

            # If no point is available, continue without speed compu-
tation
            if point.size == 0:
                labels.append(f"#{tracker_id} {vehicle_type}")
                continue

            #  Check that the transformed point is within the target
bounds.
            x, y = point
            if not (0 <= x < TARGET_WIDTH and 0 <= y < TARGET_HEIGHT):
                labels.append(f"#{tracker_id} {vehicle_type}")
                continue

            # If the detection is within the subzone (after transfor-
mation), record the y-coordinate for speed calculation.
            coordinates[tracker_id].append(y)

            # Wait until enough points are collected before computing
speed
            if len(coordinates[tracker_id]) < video_info.fps / 2:
                labels.append(f"#{tracker_id} {vehicle_type}")
            else:
                coordinate_start = coordinates[tracker_id][-1]
                coordinate_end = coordinates[tracker_id][0]
                distance = abs(coordinate_start - coordinate_end)
                time_elapsed = len(coordinates[tracker_id]) /
video_info.fps
```

```
                speed = distance / time_elapsed * 3.6  # Convert from
m/s to km/h

                # Store computed speed per vehicle type
                vehicle_speeds[vehicle_type].append(speed)
                labels.append(f"#{tracker_id} {vehicle_type}
{int(speed)} km/h")

        # Calculate average speeds per vehicle type
        for vehicle_type, speeds in vehicle_speeds.items():
            if speeds:
                _internal_data["average_speed_per_type"][vehicle_type]
= sum(speeds) / len(speeds)

        # Trigger line zone
        LINE_ZONE.trigger(detections=detections)

        # Annotate frame
        annotated_frame = frame.copy()
        annotated_frame = trace_annotator.annotate(scene=anno-
tated_frame, detections=detections)
        annotated_frame = bounding_box_annotator.annotate(scene=anno-
tated_frame, detections=detections)
        annotated_frame = label_annotator.annotate(scene=anno-
tated_frame, detections=detections, labels=labels)
        annotated_frame = counter_annotator_multiclass.annotate(anno-
tated_frame, LINE_ZONE)

        # Draw the main detection polygon (SOURCE) on the annotated
frame
        cv2.polylines(annotated_frame, [SOURCE.astype(np.int32)], is-
Closed=True, color=(0, 0, 255), thickness=3)

        # Draw the SUBZONE on the original frame for debugging:
        cv2.polylines(annotated_frame, [SUBZONE.astype(np.int32)], is-
Closed=True, color=(0, 255, 0), thickness=2)

        # Add this new line for line zone annotation:
        annotated_frame = line_zone_annotator.annotate(anno-
tated_frame, line_counter=LINE_ZONE)

        # Display the annotated frame
        cv2.imshow("annotated_frame", annotated_frame)

        # Write the annotated frame to the target video
        video_writer.write(annotated_frame)

        if cv2.waitKey(1) == ord("q"):
            break

    try:
        # Save results to JSON
        update_results()
        json_path = "Traffic_Flow_Observed.json"
        with open("Traffic_Flow_Observed.json", "w") as json_file:
            json.dump(results, json_file, indent=4)
        print("Results saved to Traffic_Flow_Observed.json")

        # Publish the JSON to MQTT
        publish_json_to_mqtt(json_path)
```

```
        # Store data to InfluxDB
        influx_write.write_to_influx("Traffic_Flow_Observed.json")

        # Release the video writer and close windows
        video_writer.release()
        cv2.destroyAllWindows()

    finally:
        # Close InfluxDB connection
        influx_config.close_client()
```

Fetch Data from InfluxDB → *influx_fetch.py*

```
from datetime import datetime, timedelta, timezone
from influx_config import query_api

def fetch_intensity(timestamp_str):
    """
    Query InfluxDB for historical intensity values from previous
hours.
    Returns a list of intensity values.
    """
    try:
        print(f"Attempting to query historical data for timestamp:
{timestamp_str}")

        # Parse the timestamp from string to datetime
        try:
            current_time = datetime.fromisoformat(timestamp_str.re-
place('Z', '+00:00'))
            current_time = current_time.replace(tzinfo=timezone.utc)
            print(f"Parsed current_time: {current_time}")
        except ValueError as e:
            print(f"Error parsing timestamp: {e}")
            current_time = datetime.now(timezone.utc)
            print(f"Using current time instead: {current_time}")

        # Calculate the start time for our query (24 hours before cur-
rent time)
        start_time = current_time - timedelta(hours=24)

        # Format timestamps for InfluxDB query
        start_time_str = start_time.strftime('%Y-%m-%dT%H:%M:%SZ')
        current_time_str = current_time.strftime('%Y-%m-%dT%H:%M:%SZ')

        # Query for intensity values
        intensity_query = f'''
        from(bucket: "Mobility Data")
            |> range(start: {start_time_str}, stop: {cur-
rent_time_str})
            |> filter(fn: (r) => r._field == "intensity")
        '''

        # Execute the query
        tables = query_api.query(query=intensity_query)

        # First find the most recent data point time
        most_recent_time = None
        for table in tables:
            for record in table.records:
```

```
                record_time = record.get_time()
                if most_recent_time is None or record_time > most_re-
cent_time:
                    most_recent_time = record_time

        # Extract intensity values within the last 2 hours of the most
recent data point
        intensity_values = []
        if most_recent_time:
            for table in tables:
                for record in table.records:
                    record_time = record.get_time()
                    # Check if the record is from the last 1+ hour
relative to the most recent data
                    if record_time >= (most_recent_time - time-
delta(minutes=65)):
                        value = record.get_value()
                        intensity_values.append(value)
                        print(f"Found recent intensity value: {value}
at time {record_time}")

        print(f"Found {len(intensity_values)} intensity values from
previous hour")
        return intensity_values

    except Exception as e:
        print(f"Error in fetch_historical_intensity: {e}")
        return []  # Return empty list on error
```

Appendix B

Webhook Listener → *webhook_listener.py*

```
import os
import glob
import json
import sys
import subprocess
import threading
from flask import Flask, request
from googleapiclient.discovery import build
from google.oauth2.service_account import Credentials
from googleapiclient.http import MediaIoBaseDownload

app = Flask(__name__)

# Service Account Key
SERVICE_ACCOUNT_FILE = "service_account.json"
creds = Credentials.from_service_account_file(SERVICE_ACCOUNT_FILE,
scopes=["https://www.googleapis.com/auth/drive"])

# Initialize Google Drive API service
service = build("drive", "v3", credentials=creds)

# Google Drive Folder ID where videos are uploaded
FOLDER_ID = "[GOOGLE_DRIVE_FOLDER_ID]"

# Download Directory
DOWNLOAD_DIR = "./[VIDEO_DIRECTORY]"

# Last Processed File Tracker
LAST_PROCESSED_FILE = "last_processed.txt"

# Active Webhook Data
ACTIVE_CHANNELS_FILE = "active_channels.json"

# Lock for Debouncing
processing_lock = threading.Lock()


@app.route("/", methods=["GET"])
def home():
    return "Webhook listener is running!", 200


def get_latest_video():
    """Fetch the latest video from the specified folder in Google
Drive."""
    try:
        results = service.files().list(
            q=f"'{FOLDER_ID}' in parents and mimeType='video/mp4'",
            orderBy="createdTime desc",
            pageSize=5,
            fields="files(id, name, createdTime)"
```

```
        ).execute()

        files = results.get("files", [])
        if files:
            return files[0]["id"], files[0]["name"]
        return None, None
    except Exception as e:
        print(f"Error fetching video: {e}")
        return None, None


def download_video(file_id, file_name):
    """Download the video from Google Drive."""
    try:
        # Ensure the download directory exists
        os.makedirs(DOWNLOAD_DIR, exist_ok=True)

        file_path = os.path.join(DOWNLOAD_DIR, file_name)
        request = service.files().get_media(fileId=file_id)
        with open(file_path, 'wb') as f:
            downloader = MediaIoBaseDownload(f, request)
            done = False
            while not done:
                status, done = downloader.next_chunk()
                print(f"Download {int(status.progress() * 100)}%.")

        # Clean up older videos after downloading the latest
        cleanup_old_videos()

        return file_path
    except Exception as e:
        print(f"Error downloading video: {e}")
        return None


def run_computer_vision(video_path):
    """Run the computer vision script with the video path."""
    try:
        output_video_path = "./annotated_result.mp4"
        print(f"Running with Python: {sys.executable}")  # Log Python
executable
        python_executable = sys.executable  # Ensures the correct Py-
thon interpreter is used
        command = [python_executable, 'Traffic_Analytics.py', '--vid-
eo', video_path, '--output', output_video_path]
        subprocess.run(command, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Error running computer vision script: {e}")


def cleanup_old_videos():
    """Delete all videos except the latest one."""
    videos = sorted(glob.glob(os.path.join(DOWNLOAD_DIR, "*.mp4")),
key=os.path.getctime, reverse=True)
    if len(videos) > 1:  # Keep only the latest video
        for video in videos[1:]:
            os.remove(video)
            print(f"Deleted old video: {video}")


def get_last_processed_file():
```

```
    """Retrieve the name of the last processed file."""
    if os.path.exists(LAST_PROCESSED_FILE):
        with open(LAST_PROCESSED_FILE, 'r') as f:
            return f.read().strip()
    return None


def update_last_processed_file(file_name):
    """Update the last processed file tracker."""
    with open(LAST_PROCESSED_FILE, 'w') as f:
        f.write(file_name)


@app.route('/webhook', methods=['POST'])
def webhook():
    """Handle webhook notifications."""
    print(f"Headers: {request.headers}")

    if request.data:
        print(f"Body: {request.data}")

    if request.is_json:
        data = request.json
        print("Webhook received (JSON):", json.dumps(data, indent=2))

    # Debounce webhook notifications
    if processing_lock.locked():
        print("Processing is already ongoing. Ignoring this webhook
notification.")
        return '', 200

    with processing_lock:
        # Fetch the latest video
        file_id, file_name = get_latest_video()
        if file_id and file_name:
            last_processed = get_last_processed_file()
            if file_name == last_processed:
                print(f"File {file_name} has already been processed.
Skipping.")
                return '', 200  # Exit early if already processed

            print(f"Latest video found: {file_name}")
            video_path = download_video(file_id, file_name)
            if video_path:
                print(f"Video downloaded to: {video_path}")

                # Automatically run the computer vision script
                run_computer_vision(video_path)

                # Update last processed file
                update_last_processed_file(file_name)
        else:
            print("No video found.")

    return "", 200

if __name__ == "__main__":
    app.run(debug=True, port=5000)
```

Register Webhook → *register_webhook.py*

```
from googleapiclient.discovery import build
from google.oauth2.service_account import Credentials
import uuid
import json

# Service Account Key
SERVICE_ACCOUNT_FILE = "service_account.json"  # Ensure this file is
in the same directory

# File to track active webhooks
ACTIVE_CHANNELS_FILE = "active_channels.json"

# Webhook settings
CHANNEL_ID = str(uuid.uuid4())
WEBHOOK_URL = "[WEBHOOK_ENDPOINT_URL]"
FOLDER_ID = "[GOOGLE_DRIVE_FOLDER_ID]"


def load_credentials():
    """Load service account credentials."""
    creds = Credentials.from_service_ac-
count_file(SERVICE_ACCOUNT_FILE, scopes=["https://www.goog-
leapis.com/auth/drive"])
    return creds


def load_active_channels():
    """Load the list of active channels from a file."""
    try:
        with open(ACTIVE_CHANNELS_FILE, 'r') as file:
            return json.load(file)
    except FileNotFoundError:
        return {}


def save_active_channel(channel_id, resource_id):
    """Save the active webhook channel and expiration timestamp."""
    active_channels = {"id": channel_id, "resourceId": resource_id}

    with open(ACTIVE_CHANNELS_FILE, 'w') as file:
        json.dump(active_channels, file)


def stop_old_channel():
    """Stop the old webhook channel before creating a new one."""
    creds = load_credentials()
    service = build('drive', 'v3', credentials=creds)

    active_channels = load_active_channels()
    if "id" in active_channels and "resourceId" in active_channels:
        try:
            service.channels().stop(
                body={"id": active_channels["id"], "resourceId": ac-
tive_channels["resourceId"]}).execute()
            print(f"Stopped old webhook channel: {active_chan-
nels['id']}")
        except Exception as e:
            print(f"Error stopping old webhook: {e}")
```

```
def setup_webhook():
    """Set up a new webhook for the Google Drive folder."""
    creds = load_credentials()
    service = build('drive', 'v3', credentials=creds)

    # Stop any existing webhook before creating a new one
    stop_old_channel()

    request_body = {
        "id": CHANNEL_ID,
        "type": "web_hook",
        "address": WEBHOOK_URL
    }

    try:
        response = service.files().watch(fileId=FOLDER_ID, body=re-
quest_body).execute()
        save_active_channel(CHANNEL_ID, response["resourceId"])
        print("Webhook set up successfully:", response)
    except Exception as e:
        print(f"Error setting up webhook: {e}")


if __name__ == "__main__":
    setup_webhook()
```

Appendix C

Edge Device Functionality → *camera_recording.py*

```
import cv2
import os
import time
from datetime import datetime
from pydrive2.auth import GoogleAuth, RefreshError
from pydrive2.drive import GoogleDrive

# Ensure the working directory is the script's directory
os.chdir(os.path.dirname(os.path.abspath(__file__)))

# Function to record a video with a timestamped filename
def record_video():
    timestamp = datetime.now().strftime('%d-%m-%Y_%H-%M-%S')
    filename = f'record_{timestamp}.mp4'

    cap = cv2.VideoCapture(0)
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = 5  # Frames per second
    duration = 60  # Recording duration in seconds

    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(filename, fourcc, fps, (frame_width,
frame_height))

    start_time = time.time()
    while time.time() - start_time < duration:
        ret, frame = cap.read()
        if ret:
            out.write(frame)
        else:
            print("Failed to capture frame. Exiting.")
            break

    cap.release()
    out.release()
    print(f"Video saved as {filename}")
    return filename

# Function to upload a file to a specific folder in Google Drive
def upload_to_drive(filename, folder_id):
    gauth = GoogleAuth()
    try:
        # Load saved credentials if they exist, otherwise authenticate
        if os.path.exists('credentials.json'):
            gauth.LoadCredentialsFile('credentials.json')
            if gauth.credentials is None or gauth.access_token_ex-
pired:
                if gauth.credentials and hasattr(gauth.credentials,
'refresh_token') and gauth.credentials.refresh_token:
                    gauth.Refresh()
```

```
                else:
                    gauth.LocalWebserverAuth()
                gauth.SaveCredentialsFile('credentials.json')
            else:
                gauth.Authorize()  # Use saved credentials if valid
        else:
            # Initialize flow properly for offline access
            gauth.LoadClientConfigFile('client_secrets.json')
            gauth.GetFlow()
            gauth.flow.params['access_type'] = 'offline'
            gauth.flow.params['approval_prompt'] = 'force'
            gauth.LocalWebserverAuth()
            gauth.SaveCredentialsFile('credentials.json')

        drive = GoogleDrive(gauth)
        file = drive.CreateFile({'title': filename, 'parents': [{'id':
folder_id}]})
        file.SetContentFile(filename)
        file.Upload()
        print(f"Uploaded {filename} to folder ID {folder_id}")

        # Maintain only the latest video in the folder
        manage_drive_files(drive, folder_id)

    except RefreshError as e:
        print(f"Error refreshing credentials: {e}")
        gauth.LocalWebserverAuth()
        gauth.SaveCredentialsFile('credentials.json')
        print("Re-authentication completed.")

# Function to manage and delete older videos on Google Drive
def manage_drive_files(drive, folder_id):
    file_list = drive.ListFile({'q': f"'{folder_id}' in parents and
trashed=false"}).GetList()
    video_files = [file for file in file_list if file['title'].starts-
with('record_') and file['title'].endswith('.mp4')]

    # Sort files by creation date (or modified date)
    video_files.sort(key=lambda x: x['createdDate'], reverse=True)

    # Keep only the latest
    if len(video_files) > 1:
        for file_to_delete in video_files[1:]:
            print(f"Deleting {file_to_delete['title']} from Google
Drive")
            file_to_delete.Delete()

# Function to manage local video files and keep only the latest
def manage_local_files(directory='.'):
    video_files = [f for f in os.listdir(directory) if f.starts-
with('record_') and f.endswith('.mp4') and os.path.isfile(f)]
    video_files.sort(key=lambda x: os.path.getmtime(os.path.join(di-
rectory, x)), reverse=True)

    # Keep only the latest
    if len(video_files) > 1:
        for file_to_delete in video_files[1:]:
            try:
                print(f"Deleting {file_to_delete} from local direc-
tory")
                os.remove(os.path.join(directory, file_to_delete))
```

```
            except OSError as e:
                print(f"Error deleting {file_to_delete}: {e}")

# Ensure the current working directory is correct when calling this
function
if __name__ == '__main__':
    folder_id = "[GOOGLE_DRIVE_FOLDER_ID]"
    filename = record_video()
    upload_to_drive(filename, folder_id)
    manage_local_files()  # Call to clean up local directory after up-
loading
```