# Distributed Traffic Surveillance Image Processing System with Fault Tolerance

Kandibanda Lohith
*Dept. of Computer Science and Engineering,*
*Amrita School of Computing, Bengaluru,*
*Amrita Vishwa Vidyapeetham, India, 560035.*
*bl.en.u4cse22032@bl.students.amrita.edu*

Lakshmi Priya P.
*Dept. of Computer Science and Engineering,*
*Amrita School of Computing, Bengaluru,*
*Amrita Vishwa Vidyapeetham, India, 560035.*
*bl.en.u4cse22034@bl.students.amrita.edu*

Yaswanth Kancharla
*Dept. of Computer Science and Engineering,*
*Amrita School of Computing, Bengaluru,*
*Amrita Vishwa Vidyapeetham, India, 560035.*
*bl.en.u4cse22031@bl.students.amrita.edu*

Supriya M.
*Dept. of Computer Science and Engineering,*
*Amrita School of Computing, Bengaluru,*
*Amrita Vishwa Vidyapeetham, India, 560035.*
*m_supriya@blr.amrita.edu*

***Abstract*—The growing need for real-time traffic monitoring in smart cities requires the design of scalable and fault-tolerant systems that can handle high-volume visual data streams. This paper introduces a distributed traffic surveillance system that uses a master–worker architecture to carry out parallel subtask execution for helmet violation detection, vehicle classification, and license plate recognition. The system utilizes image and video inputs, cutting frames and sending them asynchronously to domain-specific workers using RabbitMQ queues. For object detection functions, YOLOv9n and YOLOv11m are utilized, with PaddleOCR used for efficient extraction of characters in cropped regions of license plates. Redis handles distributed result cache storage and state tracking of tasks. The components of the system are completely containerized with Docker having a common base image to support fast scaling and deployment. A frontend based on Streamlit and a backend based on FastAPI interface offer an end-to-end pipeline from file upload to result display. The suggested system clearly illustrates distributed system concepts—task partitioning, queue-based communication, container orchestration—while supporting low-latency performance in traffic violation detection and urban monitoring applications.**

***Index Terms*—Distributed Systems, Master–Worker Architecture, YOLOv9n, YOLOv11m, PaddleOCR, Traffic Surveillance, RabbitMQ, Redis, Docker, Video Frame Processing**

## I. Introduction

Urban traffic monitoring plays a vital role in intelligent transportation systems (ITS), supporting road safety, enforcement, and adaptive city operations. However, conventional centralized surveillance systems face key limitations, including poor fault tolerance, constrained scalability, and inefficiency in handling large-scale visual data [4], [6]. These limitations hinder their deployment in modern smart cities, where real-time processing and modular integration are critical.

Recent advances in deep learning, particularly in the domain of object detection, have led to improved scene understanding for tasks like vehicle detection, helmet violation recognition, and automatic number plate recognition (ANPR). The YOLO family of detectors, including YOLOv5, YOLOv8, and their variants, has shown strong performance in traffic-related applications [1], [3], [14]. For example, YOLOv5-TS enhances detection of small-scale objects with feature fusion layers [3], while YOLO-BS provides improved detection accuracy for traffic signs in dynamic conditions [14]. Similarly, ANPR systems built using YOLO combined with EasyOCR have been proposed for license plate recognition on static image datasets [2], [7].

Despite their success, these systems often adopt a monolithic design—tight coupling of detection, recognition, and output generation—which restricts fault tolerance and scalability. They also typically support only static image inputs, limiting their usability in real-time deployments [8], [12]. More recent work like YOLO-CCA introduces context-aware feature fusion techniques to improve object detection using spatial relationships [17], and lightweight models such as the YOLOv4-based framework in [15] aim to reduce computational overhead for edge devices. However, such efforts still focus primarily on model-level enhancements rather than overall system scalability or modular distribution.

In parallel, several distributed systems frameworks have been introduced that make use of message brokers like RabbitMQ for decoupled task orchestration [5], [10]. These systems enable asynchronous communication and microservice architectures that are resilient and scalable. Nevertheless, the application of such principles to vision-based traffic systems remains limited. Most computer vision pipelines do not adopt master–worker architectures or support asynchronous, multi-model task splitting across containerized services [11], [13].

To overcome these challenges, we propose a scalable vision pipeline composed of loosely coupled microservices, each responsible for a distinct inference task in the traffic anal-

TABLE I
COMPARISON OF KEY STUDIES IN TRAFFIC SURVEILLANCE SYSTEMS

| Study | Focus Area | Technologies Used | Key Findings | Challenges / Gaps | Proposed Solutions |
|---|---|---|---|---|---|
| [2] | ANPR using EasyOCR | YOLOv5, EasyOCR, OpenCV | Effective plate recognition on images | Image-only, lacks modularity or distribution | Basic OCR + preprocessing |
| [3] | Small Object Detection | YOLOv5-TS | Improved small object detection performance | Monolithic design, image-only | Enhanced detection heads, fused features |
| [5] | Microservice Architecture | RabbitMQ, REST APIs | Message queue-based distributed task handling | Not applied to computer vision workloads | Backend service orchestration via queues |
| [7] | YOLOv8 + EasyOCR ALPR | YOLOv8, EasyOCR | Accurate plate detection on static images | No frame-wise processing or task split | YOLO + OCR integrated pipeline |
| [14] | Real-time Sign Detection | YOLOv8 (YOLO-BS) | Lightweight, efficient detection | Lacks scalability and vision modularity | Efficient YOLO block tuning |
| [17] | Context-Aware Detection | YOLOv7, CCA module | Improved accuracy via multi-level fusion | High complexity, model-focused only | Contextual feature enhancement blocks |
| **Ours** | Distributed Traffic Surveillance System | YOLOv9n, YOLOv11m, PaddleOCR, RabbitMQ, Redis, Docker | Modular multi-task architecture with support for video/image input, and real-time results | Lack of fault-tolerant, scalable, containerized full-pipeline systems in literature | Master–worker architecture, asynchronous task queues, microservice deployment |

ysis workflow into three parallel subtasks—helmet violation detection, vehicle classification, and license plate recognition—each assigned to a specialized worker microservice. The architecture integrates YOLOv9n and YOLOv11m for visual detection, while PaddleOCR handles license plate text extraction from cropped regions. These components are orchestrated using RabbitMQ for message-based task delegation, and Redis is used for task state caching and result aggregation. The system supports both image and video input formats, splits video streams frame-by-frame, and returns real-time outputs through a FastAPI backend and Streamlit interface. All services are containerized via Docker and optimized using a shared base image for efficient deployment.

This work makes the following key contributions: it introduces a distributed, containerized system for real-time traffic surveillance using task-level decomposition; a multi-model architecture combining YOLOv9n, YOLOv11m, and PaddleOCR across microservices; and full support for video and image input streams with scalable deployment using Docker.Unlike previous studies focused solely on model optimization, our work showcases the integration of distributed computing principles into a real-time traffic vision system, enabling scalable, asynchronous, and containerized processing. Table I provides a comparative overview of relevant prior studies and the key innovations of our proposed system.

The remainder of this paper is organized as follows: Section II reviews related literature and system-level foundations. Section III presents the architecture and design. Section IV describes the implementation and microservices. Section V evaluates performance. Section VI concludes with future work.

## II. RELATED WORK

Deep learning-based object detection systems have been extensively applied to traffic surveillance tasks such as vehicle classification, helmet violation detection, and traffic sign recognition. Wang et al. [1] presented an improved YOLOv5 architecture that integrated an adaptive feature fusion pyramid (AF-FPN) to enhance multi-scale traffic sign detection. Their model addressed the scale variance of signs but was evaluated primarily on static image datasets and lacked modular deployment considerations. Similarly, Shen et al. [3] introduced YOLOv5-TS, which incorporated fused feature layers and optimized detection heads to reduce false negatives in small object detection. While both studies achieved improvements in detection performance, neither addressed system-level scalability, fault tolerance, or microservice-based orchestration.

Several studies have targeted automatic number plate recognition (ANPR) using OCR-enhanced YOLO pipelines. Rao et al. [2] developed a Python-based framework that combined YOLOv5 with EasyOCR and OpenCV to perform number plate detection and character recognition on image datasets. Although the system demonstrated acceptable accuracy under controlled conditions, it relied on sequential processing and lacked any asynchronous or parallel execution mechanism. Salsabila [7] extended this work by integrating YOLOv8 with EasyOCR, showing improved inference time and recognition rates, but still constrained to single-image inputs. Sushma et al. [12] evaluated YOLOv5 + EasyOCR on Indian traffic plates, highlighting performance in noisy conditions. All three works, while valuable, were limited to monolithic design and did not consider frame-level task distribution for video processing.

In terms of model optimization, various lightweight and context-aware architectures have been explored. Zhang et al. [14] proposed YOLO-BS, an enhancement over YOLOv8 aimed at real-time detection in traffic scenes with improved efficiency through structural refinements. Gu and Si [15] introduced a compact YOLOv4 variant that reduced model size for embedded applications while maintaining detection accuracy. Jiang et al. [17] introduced YOLO-CCA, a context-aware model that incorporated spatial and relational information using adaptive local context modules to improve small object detection. While these studies successfully improve accuracy or reduce inference time, they are primarily model-centric and overlook distributed system design or asynchronous task decomposition. Flores-Calero et al. [16] compiled a comprehensive systematic review of YOLO-based traffic sign detection, surveying trends across datasets, hardware platforms, and performance metrics, but did not address architectural modularity or containerization.

Several studies have attempted broader system-level solutions, particularly in terms of microservice design and distributed orchestration. Catovic et al. [5] implemented a RabbitMQ-based microservice infrastructure to support asynchronous messaging in distributed web applications. Their architecture successfully demonstrated decoupled service interaction but was not extended to real-time vision processing pipelines. Jones et al. [10] conducted a performance evaluation of RabbitMQ under high-load scenarios, showing its scalability and efficiency for task-based systems. Miron and Hulea [11] proposed a lightweight, distributed traffic monitoring system suitable for edge environments. However, their design was sensor-based and did not leverage deep learning models for image or video inference.

Early distributed traffic monitoring systems, such as the one proposed by Kosonen and Bargiela [6], utilized agent-based communication to manage real-time sensor networks. While innovative at the time, the system lacked integration with vision pipelines and modern detection models. Khamidehi and Sousa [13] presented a distributed multi-agent approach using reinforcement learning for UAV-based traffic monitoring. Their framework emphasized task allocation and route planning rather than computer vision-based detection or OCR integration. Zhang et al. [8] also proposed improvements to YOLOv5 using attention blocks to enhance traffic sign detection, but like other model-centric works, it remained limited to a static, single-service design.

In contrast to these prior works, our proposed system introduces a fully modular and distributed master–worker architecture for real-time traffic scene analysis. Unlike previous efforts, it integrates task-specific YOLO models (YOLOv9n for helmet detection and YOLOv11m for vehicle classification) and PaddleOCR for license plate recognition, all deployed as isolated microservices. Using RabbitMQ for asynchronous message delegation and Redis for result caching, the system supports frame-wise processing of both video and image inputs. All components are containerized using Docker with a shared base image, enabling scalable and reproducible deployment. This unified architecture bridges the gap between vision model performance and distributed system design, offering a robust platform for real-time smart city traffic monitoring.

## III. System Architecture

### A. System Overview

The system adopts a microservice-oriented architecture tailored for responsive traffic scene analysis in dynamic urban environments. It follows a master–worker paradigm, where a central controller handles input reception, task delegation, and result aggregation, while dedicated worker services execute specialized vision tasks in parallel. The architecture is designed to be scalable, resilient, and deployable across containerized infrastructures. All services operate statelessly and communicate asynchronously through a central message queue.

Figure 1 illustrates the complete layout of the system. The user initiates the process by uploading an image or video through a web interface. The backend server, built with FastAPI, orchestrates all preprocessing and queue assignment logic. For video inputs, the system performs frame-by-frame extraction before forwarding each frame to the appropriate processing queues. RabbitMQ serves as the message broker, enabling asynchronous and decoupled communication between services.

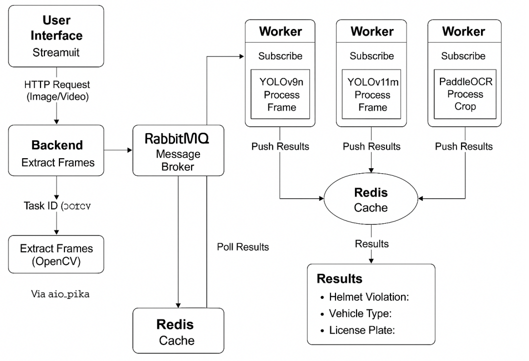


Fig. 1. System architecture of the distributed traffic surveillance platform.

### B. Component Breakdown

The frontend interface is implemented using Streamlit, providing a lightweight interface for user uploads and result visualization. The backend server developed in FastAPI manages preprocessing logic, task assignment, and communication with the message broker. RabbitMQ coordinates task distribution via separate queues for helmet detection, vehicle classification, and license plate recognition. Each queue is consumed by an independent worker container optimized for its designated task.

The helmet detection worker uses the YOLOv9n model to identify motorcyclists and verify helmet usage. The vehicle detection worker leverages YOLOv11m to detect and classify vehicle types, and it extracts candidate regions containing license plates. These cropped regions are then sent to the plate recognition worker, which employs PaddleOCR to extract and interpret the alphanumeric characters. Each of these workers operates as a stateless microservice, packaged into a standalone Docker container, and is horizontally scalable.

Redis is used as a shared in-memory store for caching intermediate results from each worker. This facilitates asynchronous result aggregation and reduces coupling between components. The backend monitors Redis for task completion and compiles the results into a structured output. The final detection report is rendered via the Streamlit frontend, including annotated frames and violation summaries.

### C. Data Flow and Task Execution

When a user uploads a video file, the system extracts its frames and assigns each to a unique task identifier. The backend then publishes the frames to multiple RabbitMQ queues corresponding to the task categories. Each queue operates independently and can buffer messages in case of worker unavailability. The helmet detection queue receives full frames for motorcyclist evaluation, while the vehicle detection queue processes the same frames to identify vehicles and extract potential license plate regions. These cropped regions are then sent to the plate recognition queue for OCR processing.

Each worker listens to its respective queue and performs inference on incoming tasks. Upon completion, the results are stored in Redis, using a combination of frame ID and task label as the key. The master service continuously checks Redis to determine if all three processing stages have been completed for each frame. Once available, the results are aggregated into a structured format that includes bounding boxes, violation flags, vehicle types, and recognized license plate numbers. This output is then relayed to the frontend for user review.

### D. Scalability and Fault Tolerance

The system supports horizontal scaling through the replication of worker containers. Docker enables the deployment of multiple instances of each processing service, allowing load balancing across tasks. RabbitMQ ensures that if a worker instance fails during processing, the message is retained in the queue and can be reprocessed by another available instance. This approach guarantees fault isolation at the service level and ensures consistent task completion even under partial failure.

Redis is configured to operate as a shared in-memory store that acts as both a buffer and a synchronization mechanism. By using task identifiers and atomic operations, the system avoids race conditions and supports concurrent processing without data inconsistency. This design decouples the processing and aggregation stages and enables robust handling of variable workloads without affecting frontend responsiveness.

### E. System Deployment

All system components are containerized using Docker, enabling platform-independent deployment and isolation. A shared base image is used across all vision worker containers to reduce build redundancy and ensure environment consistency. Docker Compose is employed for orchestrating the services, defining dependencies, environment variables, and network configurations within a single configuration file. The system can be deployed locally for testing or scaled to a cloud environment with minimal configuration changes. The microservice architecture, combined with container-based isolation, enables elastic scalability and simplifies maintenance, debugging, and future extension of system functionalities.

## IV. Implementation

### A. Technology Stack

The implementation of the distributed traffic surveillance system leverages a combination of open-source technologies, deep learning models, and containerization tools to achieve real-time performance, modularity, and scalability. The system is primarily developed using Python, which serves as the foundation for both backend services and model-based inference components.

For object detection, the YOLOv9n and YOLOv11m models are employed. YOLOv9n is optimized for lightweight helmet detection, while YOLOv11m provides enhanced accuracy for multi-class vehicle classification. PaddleOCR is used for license plate recognition, offering a robust optical character recognition framework capable of handling diverse fonts and backgrounds with minimal preprocessing. These models are integrated into task-specific microservices.

The backend is built using FastAPI, a high-performance asynchronous web framework that handles image and video uploads, frame processing, and communication with message queues. Streamlit is used for the frontend, providing an interactive interface for users to upload data and visualize results. For inter-process communication, RabbitMQ serves as the message broker, enabling asynchronous task dispatching across multiple workers. Redis is used as an in-memory key–value store to cache intermediate results and support stateful result aggregation.

Docker is used to containerize all components, ensuring environment consistency and simplifying deployment. Each worker service, as well as the API and message broker, is encapsulated within individual Docker containers. The system orchestration is handled via Docker Compose, enabling scalable and reproducible deployments across local or cloud-based infrastructures.

### B. Worker Microservices

The core of the system's distributed processing logic is built around three task-specific worker microservices, each designed to perform a distinct vision task independently and concurrently. These workers subscribe to their respective RabbitMQ queues and operate in a stateless manner, allowing multiple instances to be scaled horizontally based on processing load.

The helmet detection worker is responsible for identifying helmet usage among motorcyclists in each video frame. It runs a pre-trained YOLOv9n model, chosen for its balance between speed and accuracy on lightweight object detection tasks. Upon receiving a frame from its queue, the worker performs inference and generates bounding boxes indicating the presence or absence of helmets. The output is serialized and written to Redis, tagged with the frame ID and task label.

The vehicle detection worker utilizes YOLOv11m to detect and classify vehicles such as two-wheelers, cars, and trucks. This worker also performs an additional operation of cropping detected regions that potentially contain license plates, which are later passed to the plate recognition worker. Each cropped image is linked to the corresponding original frame ID and sent as a task to a dedicated RabbitMQ queue for OCR processing.

The plate recognition worker uses PaddleOCR to extract alphanumeric content from the cropped license plate regions. The OCR output includes both the recognized text and the confidence scores, which are then stored in Redis for further aggregation. Due to variations in lighting, angle, and occlusion, PaddleOCR's robustness and multilingual capabilities make it suitable for diverse traffic environments.

All three workers are implemented as independent Python-based services using asynchronous consumers for RabbitMQ. Their modularity enables easy upgrades, independent testing, and dynamic scaling. Each worker is containerized using Docker, and all share a common base image that includes shared dependencies such as OpenCV, PyTorch, PaddleOCR runtime, and messaging libraries. This approach significantly reduces redundancy during deployment and ensures consistent behavior across service replicas.

## *C. Backend Server and Task Routing*

The backend server is implemented using FastAPI, an asynchronous Python framework that handles core routing, request parsing, and task coordination across the system. It serves as the central communication layer between the user-facing interface, the message queues, and the Redis cache. The API endpoints accept user uploads in the form of images or video files, which are processed based on the input type.

For image inputs, the file is temporarily stored and directly enqueued into all three RabbitMQ queues corresponding to the helmet detection, vehicle detection, and plate recognition tasks. For video inputs, the backend performs frame extraction using OpenCV. Frames are sampled at regular intervals, assigned a unique frame identifier, and stored temporarily for asynchronous processing. Each extracted frame is then published to the helmet and vehicle queues with a unique task ID and metadata.

Task publishing is performed using persistent RabbitMQ channels to ensure message delivery even in the event of service restarts. For plate recognition, the system adopts a two-stage flow: the vehicle worker sends cropped plate regions to the plate queue, which the backend monitors for result association using mapped frame-task keys. This design enables intermediate outputs to be rerouted or reprocessed without blocking downstream services.

As the workers process tasks and push results into Redis, the backend concurrently queries Redis to check for task completion status. Using Redis' key–value store and hash data structures, results are organized by frame ID and task type. Once all expected outputs for a given frame are available, the backend aggregates the data into a unified response object that includes detection results, annotated metadata, and OCR text.

This aggregated output is then passed to the frontend interface or returned via API response, depending on the client. The use of asynchronous routing and non-blocking I/O in FastAPI ensures that the system can handle high input rates without performance degradation. The backend also implements basic error handling and retry logic to recover from temporary queue or cache unavailability.

## *D. Redis Integration*

Redis serves as the system's centralized, in-memory key–value store for managing intermediate and final results generated by the worker services. Its lightweight and low-latency characteristics make it ideal for storing and retrieving high-frequency data, especially in asynchronous, multi-worker environments where task synchronization is critical.

Each worker, upon completing its task, serializes its output and writes it to Redis using a unique composite key that combines the frame ID and the task type (e.g., "frame_12:vehicle" or "frame_45:helmet"). These entries are stored using hash data structures that allow fast lookup, partial updates, and efficient memory usage. This structure ensures that partial results for each frame can be independently accessed and updated without conflict, supporting concurrent processing by different services.

The backend process continuously polls Redis to determine when all expected results for a frame have been received. Once the entries for helmet detection, vehicle classification, and plate recognition are present for a given frame ID, the system aggregates the results into a unified response object. This response includes bounding box coordinates, classification labels, OCR text, and metadata such as confidence scores and timestamps.

Redis also plays a crucial role in managing task status and coordination in failure scenarios. In the event of a worker crash or delayed response, the absence of expected keys acts as a signal to either retry or reschedule the task. The backend applies configurable timeouts and validity checks to identify incomplete or stale entries, ensuring that only consistent and complete results are returned to the user.

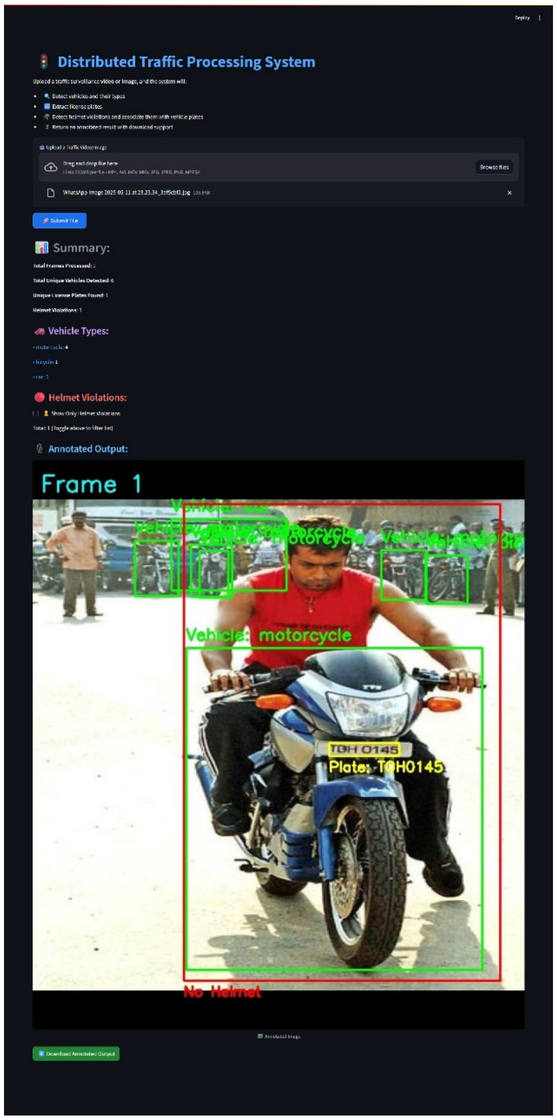
Overall, Redis functions as a fast-access synchronization layer between loosely coupled services, enabling real-time communication and state management without introducing bottlenecks. Its integration into the system is essential for ensuring that the distributed architecture remains cohesive, consistent, and responsive under varying load conditions.

## *E. Frontend Interface*

The frontend of the system is implemented using Streamlit, an open-source Python library designed for building interactive web applications with minimal overhead. It provides a lightweight interface for user interaction, including file uploads, task initiation, and visualization of real-time inference results. Streamlit's reactive rendering engine enables seamless updates as backend processes complete their tasks.

Upon launching the application, users are presented with options to upload either image files or video recordings. Once an input is selected and submitted, the frontend initiates a request to the FastAPI backend via a secure HTTP endpoint. While processing occurs asynchronously in the background, the interface displays task status updates and progress indicators, providing feedback on the number of frames processed and the status of each detection task.

For output visualization, the frontend renders annotated image frames, showing bounding boxes for helmets, vehicles, and license plates. In the case of license plate recognition, the recognized text and associated confidence scores are displayed adjacent to the corresponding cropped regions. All detection results are presented in a structured format, allowing users to inspect each frame independently, as illustrated in Figure 2.



Distributed Traffic Processing System

Upload a traffic surveillance video or image, and the system will:

- Detect vehicles and their types
- Extract license plates
- Detect helmet violations and associate them with vehicle plates
- Return an annotated result with download support

Upload a Traffic Video/Image

Drag and drop file here

WhatsApp Image 2025-05-11 at 23.23.34_3ef5cbf2.jpg

Summary:

Total Frames Processed: 1

Total Unique Vehicles Detected: 6

Unique License Plates Found: 1

Helmet Violations: 1

Vehicle Types:

- motorcycle: 4
- bicycle: 1
- car: 1

Helmet Violations:

Show Only Helmet Violations

Total: 1 (Toggle above to filter list)

Annotated Output:

Fig. 2. Custom-developed Streamlit UI displaying annotated frames and OCR outputs.

To enhance usability, the frontend is designed to support paginated result browsing and reprocessing triggers in case of failures. Additionally, the system provides an option to download the final JSON result containing all detection and OCR outputs, which can be used for external audit, integration, or archival purposes.

The frontend maintains loose coupling with the backend, relying solely on API responses and periodic polling for result availability. This separation enables the UI to remain responsive even under varying backend loads and supports potential future migration to alternative front-end frameworks if required.

### F. Containerization and Deployment

To ensure modularity, portability, and ease of scaling, the entire system is deployed using Docker containers. Each microservice—including the FastAPI backend, RabbitMQ broker, Redis cache, and all three vision workers—is encapsulated within an independent Docker container. This approach provides consistent runtime environments across development, testing, and deployment stages, minimizing configuration conflicts and dependency issues.

A shared base image is used for all three worker services, consolidating common libraries such as OpenCV, PyTorch, PaddleOCR runtime, and the messaging client. This reduces image size, accelerates build times, and promotes efficient layer caching during container creation. The shared image architecture also simplifies updates to shared dependencies and enforces version consistency across worker containers.

The system is orchestrated using Docker Compose, which defines service relationships, environment variables, network settings, and volume bindings in a single configuration file. This allows the system to be launched or scaled with minimal manual setup. Each service is assigned a specific role and hostname, enabling clean inter-container communication through an internal virtual network.

Additionally, Docker Compose facilitates horizontal scalability by allowing multiple replicas of each worker container to be instantiated. This is particularly beneficial for video processing tasks, where parallelism at the frame level can significantly reduce latency. Load balancing is inherently managed by RabbitMQ queues, which distribute tasks evenly among active worker replicas.

The containerized deployment model also enables portability to cloud platforms and edge devices. With minor configuration changes, the entire stack can be deployed in remote environments or Kubernetes-based orchestration systems. This flexibility ensures that the system remains adaptable to a variety of real-world deployment scenarios without compromising performance or maintainability.

## V. Results and Evaluation

This section presents a comprehensive evaluation of the proposed distributed traffic surveillance system, emphasizing its functional performance, system efficiency, and deployment feasibility. The evaluation focuses on three key components—helmet detection, vehicle classification, and license plate recognition—and demonstrates the system's ability to operate under real-time constraints using image and video inputs.

### A. Helmet Detection (YOLOv9n)

The helmet detection module leverages the lightweight YOLOv9n model, selected for its low latency and strong detection accuracy in high-density scenes. During evaluation, the system was able to accurately distinguish motorcyclists and identify whether helmets were worn. Even in complex traffic frames with partial occlusion and multiple riders, the detector consistently flagged helmet violations. As illustrated in Fig. 3, the annotated outputs were seamlessly integrated into the frontend display, enabling end-users to visualize violations in real time.

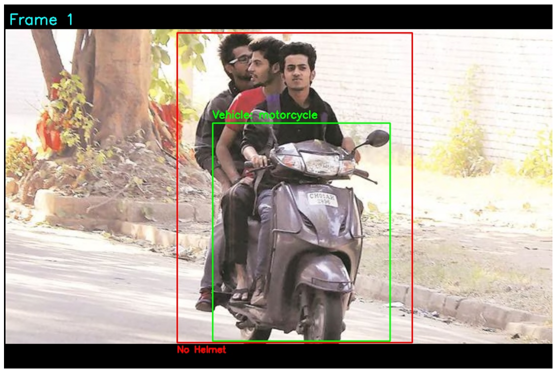


Fig. 3. Detected Helmet Violation

## B. Vehicle Classification (YOLOv11m)

Vehicle detection and classification were handled using YOLOv11m, a moderately heavier model optimized for multi-class object recognition. The worker successfully differentiated between two-wheelers, cars, trucks, and buses, even in densely populated frames. In addition to classification, the module also extracted license plate regions from detected vehicles and dispatched them to the OCR pipeline. The classification output, as shown in Fig. 4, remained stable across various lighting conditions and camera angles, ensuring robustness in real-world settings.

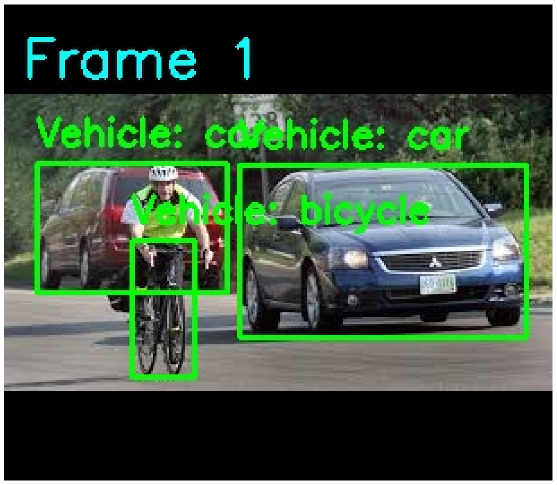


Fig. 4. Vehicle Classification Output with Plate Crops

## C. License Plate Recognition (PaddleOCR)

License plate recognition was performed using PaddleOCR configured in lightweight English mode. This worker was responsible for interpreting alphanumeric text from the cropped license plate images passed by the vehicle detection module. The OCR system showed high resilience against image noise, minor blurring, and skewed inputs. As seen in Fig. 5, confidence scores were consistently above 90% for clear plates. The recognized text was rendered within the frontend interface and logged within the JSON output for auditability.

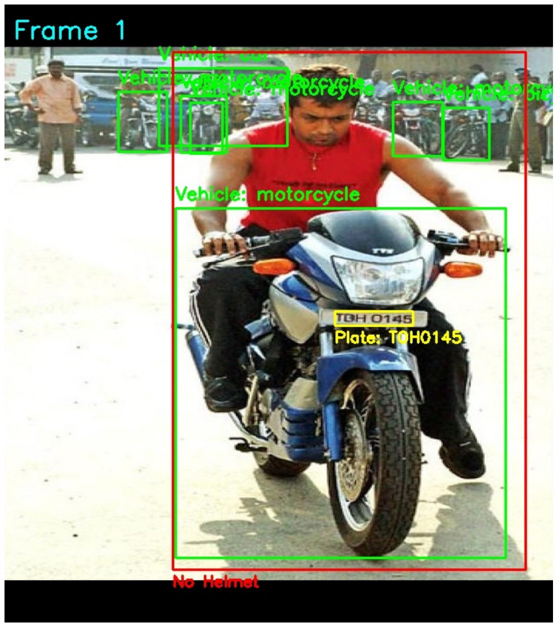


Fig. 5. License Plate OCR – Cropped Region and Text Output

## D. Frontend Integration

The Streamlit-based frontend interface provided a responsive and user-friendly interaction layer. It enabled users to upload inputs, monitor processing status, and visualize annotated results. All outputs—including bounding boxes, class labels, and OCR text—were displayed in a paginated format, allowing per-frame inspection. Additionally, the frontend facilitated downloading of structured JSON responses, which could be utilized in downstream workflows such as violation logging or report generation.

## E. Performance Observations

The performance of each worker module was measured using benchmark tests across multiple video frames. The system demonstrated efficient runtime behavior:

- YOLOv9n (helmet detection): 95 ms per frame.
- YOLOv11m (vehicle classification): 140–160 ms per frame.
- PaddleOCR (license plate recognition): 80–120 ms depending on image clarity.

Docker-based containerization enabled dynamic scaling of workers during stress tests. This ensured consistent processing times and prevented bottlenecks during concurrent task execution.

### F. Experimental Setup

All experiments were conducted on a machine equipped with an Intel Core i5-1240P processor, 16 GB RAM, and a 512 GB SSD. The complete system stack—including FastAPI, Redis, RabbitMQ, and the three worker services—was deployed using Docker Compose in an isolated local environment. The test data consisted of recorded traffic videos (10–60 seconds at 25 FPS) and static images reflecting real-world scenarios. Frame sampling was configured at 1 FPS to simulate lightweight real-time processing.

### G. Discussion

The evaluation confirms that the system achieves its design objectives: real-time responsiveness, modularity, and scalability. The message-driven master–worker framework ensures distributed load handling and fault isolation, while Redis acts as a reliable intermediary for result caching. Minor inconsistencies in OCR were primarily caused by occlusions or low-resolution inputs, which could be mitigated through confidence-based post-filtering.

The use of Docker containers and RabbitMQ enables horizontal scaling, allowing the system to adapt to fluctuating workloads without impacting performance. Future enhancements may include GPU acceleration, multilingual OCR support, and adaptive model retraining based on environmental feedback.

In conclusion, the system demonstrates effective integration of distributed computing and computer vision for urban traffic monitoring. It meets real-time operational demands and provides a flexible base for deployment in law enforcement and smart city infrastructure.

## VI. Conclusion and Future Work

This paper presented a scalable traffic surveillance system that performs helmet detection, vehicle class identification, and license plate recognition using a distributed microservice architecture. With a master–worker model task assignment using RabbitMQ, Redis-backed outcome aggregation, and containerized deployment with Docker, the system achieves high performance, fault tolerance, and deployability. The combination of a Streamlit frontend and FastAPI backend offers a simple-to-use interface and ordered output delivery. Evaluation revealed strong frame-wise processing, low-latency inference, and strong support for image and video inputs. Loosely coupled architecture allows for horizontal scaling and modular updates without disturbing the core pipeline.

Future work includes supporting multilingual OCR, optimizing inference through GPU acceleration, and adding automatic violation reporting. Enhancements such as confidence-based filtering, adaptive thresholds, and feedback-driven learning could further improve performance in real-world deployments. The proposed system offers a practical foundation for vision-based traffic enforcement in smart city infrastructures.

## References

[1] Wang, J., Chen, Y., Dong, Z. and Gao, M., 2023. Improved YOLOv5 network for real-time multi-scale traffic sign detection. Neural Computing and Applications, 35(10), pp.7853-7865.

[2] Rao, T., Rawat, S., Gupta, A., Mehra, N., Dhondiyal, S.A. and Kapil, D., 2025. Comprehensive study on automatic number plate recognition with python using OpenCV and EasyOCR. In Challenges in Information, Communication and Computing Technology (pp. 501-505). CRC Press.

[3] Shen, J., Zhang, Z., Luo, J. and Zhang, X., 2023. YOLOv5-TS: Detecting traffic signs in real-time. Frontiers in Physics, 11, p.1297828.

[4] Kekevi, U. and Aydın, A.A., 2022. Real-time big data processing and analytics: Concepts, technologies, and domains. Computer Science, 7(2), pp.111-123.

[5] Catovic, A., Buzadija, N. and Lemes, S., 2022. Microservice development using RabbitMQ message broker. Science, engineering and technology, 2(1), pp.30-37.

[6] Kosonen, I. and Bargiela, A., 1999. A distributed traffic monitoring and information system. Journal of Geographic Information and Decision Analysis, 3(1), pp.31-40.

[7] Salsabila, N., 2024. Enhancing Automated Vehicle License Plate Recognition with YOLOv8 and EasyOCR. Journal of Information Systems and Informatics, 6(3).

[8] Zhang, R., Zheng, K., Shi, P., Mei, Y., Li, H. and Qiu, T., 2023. Traffic sign detection based on the improved YOLOv5. Applied Sciences, 13(17), p.9748.

[9] Christodoulou, C. and Kolios, P., 2020, May. Optimized tour planning for drone-based urban traffic monitoring. In 2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring) (pp. 1-5). IEEE.

[10] Jones, B., Luxenberg, S., McGrath, D., Trampert, P. and Weldon, J., 2011. RabbitMQ performance and scalability analysis. project on CS, 4284.

[11] Miron, R. and Hulea, M., Lightweight Architecture for Distributed Road Traffic Monitoring.

[12] R. Sushma, M. R. Devi, N. Maheshwaram, and S. Bhukya, "Automatic License Plate Recognition with YOLOv5 and Easy-OCR method," Int. J. Innov. Res. Technol., vol. 9, no. 1, pp. 1243–1247, Jun. 2022.

[13] Khamidehi, B. and Sousa, E.S., 2021, September. Distributed deep reinforcement learning for intelligent traffic monitoring with a team of aerial robots. In 2021 IEEE International Intelligent Transportation Systems Conference (ITSC) (pp. 341-347). IEEE.

[14] Zhang, H., Liang, M. and Wang, Y., 2025. YOLO-BS: a traffic sign detection algorithm based on YOLOv8. Scientific Reports, 15(1), p.7558.

[15] Gu, Y. and Si, B., 2022. A novel lightweight real-time traffic sign detection integration framework based on YOLOv4. Entropy, 24(4), p.487.

[16] Flores-Calero, M., Astudillo, C.A., Guevara, D., Maza, J., Lita, B.S., Defaz, B., Ante, J.S., Zabala-Blanco, D. and Armingol Moreno, J.M., 2024. Traffic sign detection and recognition using YOLO object detection algorithm: A systematic review. Mathematics, 12(2), p.297.

[17] Jiang, L., Zhan, P., Bai, T. and Yu, H., 2024. YOLO-CCA: A Context-Based Approach for Traffic Sign Detection. arXiv preprint arXiv:2412.04289.