Journal of Computer and Electrical Sciences (JCES)- Volume-1 Issue-4, April 2026
ISSN: 3049-2602

VEHCILE COUNT AND DATA STORING USING SENSOR DATA

V. Praneeth Sai[1], Mr. B.N.V.Basaveswara Rao [2]
[1]Student, Department of Computer Science and Engineering
Andhra Loyola Institute of Engineering and Technology, Vijayawada, Andhra Pradesh, India
[2]Assistant Professor, Department of Computer Science and Engineering
Andhra Loyola Institute of Engineering and Technology, Vijayawada, Andhra Pradesh, India
Email: praneethsai.vajrala1407@gmail.com

Abstract: This project presents a comprehensive **Vehicle Intelligence Platform** that transforms ordinary traffic camera feeds into valuable automotive market research data. The system employs **YOLOv8 (You Only Look Once)**, a state-of-the-art deep learning model, for real-time vehicle detection and classification. Unlike traditional traffic monitoring systems that only count vehicles, our solution identifies and tracks **vehicle types, colors, models, number plates, and speed patterns** by matching detected vehicles with a pre-loaded database. The methodology follows a three-phase approach: **video acquisition** from multiple sources (live cameras or recorded footage), **intelligent processing** using computer vision techniques, and **data visualization** through an interactive web dashboard. Vehicles crossing a virtual counting line are tracked and counted in both directions (IN/OUT), ensuring accurate traffic flow analysis. Each detected vehicle is matched with database entries to display detailed information including make, model, color, and estimated speed. The system architecture consists of four layers: an **input layer** for video capture, a **processing layer** utilizing OpenCV and YOLOv8 for detection, a **storage layer** using SQLite database for persistent data storage, and a **visualization layer** featuring a Flask-based web dashboard with interactive charts, graphs, and real-time statistics.

Keywords: Machine Learning, Computer Vision, Object Detection, Vehicle Tracking, YOLOv8, License Plate Recognition (LPR), Optical Character Recognition (OCR), EasyOCR, Speed Estimation, Vehicle Classification, Color Recognition, K-Means Clustering, ResNet50, Image Classification, Real-time Monitoring, Database Management, SQLite, Data Visualization, Flask Dashboard.

1. INTRODUCTION

For decades, traffic monitoring and vehicle surveillance have relied on physical infrastructure such as inductive loop detectors, piezoelectric sensors, pneumatic road tubes, and radar-based speed guns. While effective, these solutions suffer from high installation and maintenance costs, limited scalability, susceptibility to environmental degradation, and an inability to capture semantic attributes such as vehicle colour, model, or license plate content. The emergence of **virtual sensors** — software-based perception systems that extract meaningful information from ordinary cameras — has disrupted this paradigm by offering a low-cost, flexible, and richly informative alternative.

Recent advances in deep learning, particularly in the domains of **object detection**, **landmark tracking**, **optical character recognition**, and **image classification**, have made such systems accessible to researchers and developers. The release of **YOLOv8** marked a significant milestone by delivering a unified architecture that supports detection, tracking, and segmentation in real time. When paired with **OpenCV** for frame acquisition and rendering, **EasyOCR** for text extraction, and **PyTorch** for deep learning inference, a complete vehicle monitoring pipeline can be implemented in under 500 lines of Python code.

The remainder of this paper is organised as follows: Section 2 surveys related work in vehicle monitoring, license plate recognition, and virtual sensor technologies. Section 3 describes the system architecture and data flow. Section 4 details each functional module and its algorithmic implementation. Section 5 presents

https://doi.org/10.5281/zenodo.19613703

the development environment, performance benchmarks, accuracy results, and test cases with annotated screenshots. Section 6 concludes the paper and outlines directions for future research.

2. LITERATURE SURVEY

The domain of vehicle monitoring has attracted substantial research interest over the past two decades, driven by the proliferation of surveillance cameras and the growing demand for intelligent transportation systems (ITS). This section reviews foundational and contemporary works in vehicle detection, license plate recognition, speed estimation, and virtual sensor architectures.

Vehicle Detection and Tracking: Early approaches to vehicle detection relied on background subtraction (Stauffer & Grimson, 1999) and frame differencing (Collins et al., 2000), which performed adequately under static camera conditions but failed in dynamic lighting or congested scenes. The introduction of Haar cascades (Viola & Jones, 2001) improved robustness but remained computationally expensive for real-time applications. The watershed moment arrived with convolutional neural networks (CNNs): **R-CNN** (Girshick et al., 2014) and its successors **Fast R-CNN** (Girshick, 2015) and **Faster R-CNN** (Ren et al., 2015) achieved state-of-the-art accuracy but at speeds unsuitable for live video. The **YOLO** (You Only Look Once) family (Redmon et al., 2016; Redmon & Farhadi, 2018; Bochkovskiy et al., 2020) revolutionised real-time detection by framing detection as a single regression problem, achieving both high accuracy and frame rates exceeding 30 fps. Our system adopts **YOLOv8** (Ultralytics, 2023), which adds built-in tracking (BoT-SORT) and improved feature extraction, making it ideal for multi-vehicle scenarios.

License Plate Recognition (LPR): Automatic number plate recognition has been extensively studied, with traditional approaches using image binarisation (Otsu, 1979), morphological operations, and template matching (Shapiro & Stockman, 2001). These methods struggled with varying fonts, rotations, and lighting. The adoption of **OCR engines** — particularly **Tesseract** (Smith, 2007) — improved generalisation but required careful preprocessing. More recently, deep learning-based recognisers such as **CRNN** (Shi et al., 2017) and **EasyOCR** (JaidedAI, 2021) have achieved human-level performance on standard benchmarks. EasyOCR, which we employ, uses a CRAFT text detector (Baek et al., 2019) followed by a CRNN recogniser, supporting over 80 languages and operating efficiently on CPU.

Speed Estimation from Monocular Video: Estimating vehicle speed from a single camera is inherently challenging due to the loss of depth information. Common approaches include **pixel-to-world calibration** (Grammatikopoulos et al., 2005), where a known distance in the scene is mapped to pixel displacement, and **homography-based methods** (Milanés et al., 2012) that project points onto a ground plane. Our system adopts the simpler pixel-to-world calibration, requiring the user to measure a reference distance (e.g., a 5-metre road marking) in pixels. The speed is then computed as (pixel_displacement × meters_per_pixel) / (frame_difference / FPS) × 3.6 to obtain km/h.

3. SYSTEM ARCHITECTURE

The proposed system follows a **pipeline architecture** comprising five sequential stages: video acquisition, frame processing, feature extraction, database persistence, and alert generation. A parallel thread manages the web dashboard, serving real-time data visualisations. Figure 1 illustrates the architectural diagram (conceptual).

Stage 1: Video Acquisition – The cv2.VideoCapture module reads frames from either a live webcam (index 0) or a prerecorded video file. Frames are resized to 640×640 pixels to balance resolution and inference speed while maintaining YOLO’s input requirements.

Stage 2: Frame Processing – Each frame is passed to the YOLOv8 model with tracking enabled (model.track()). The model returns bounding boxes, class labels, confidence scores, and unique track IDs

https://doi.org/10.5281/zenodo.19613703

for each detected vehicle (filtered to classes 2, 3, 5, 7: car, motorcycle, bus, truck). Bounding box coordinates are extracted and passed to subsequent modules.

Stage 3: Feature Extraction – For each detected vehicle:

- The **colour** is determined by applying K-Means (k=3) to the RGB pixels within the bounding box; the dominant cluster is mapped to one of seven colour names.
- The **license plate** is extracted using EasyOCR; the bounding box region is passed directly to the reader, which returns text strings for any detected characters.
- The **speed** is computed only for vehicles that have appeared in at least two frames. The centroid displacement (Euclidean distance between current and previous centroids) is multiplied by a calibration factor (meters per pixel) and divided by the inter-frame time.

Stage 4: Database Persistence – All extracted attributes (track ID, frame number, timestamp, type, colour, plate, speed, model) are inserted into an SQLite table vehicles. A separate alerts table stores timestamped speed violation records.

Dashboard Thread – A Flask application runs independently, exposing two REST endpoints: /api/vehicles returns all vehicle records as JSON, and /api/charts returns Plotly visualisations (pie chart of vehicle types, histogram of speeds, line chart of alerts over time). The frontend HTML page consumes these endpoints and renders the dashboard using Plotly.js.

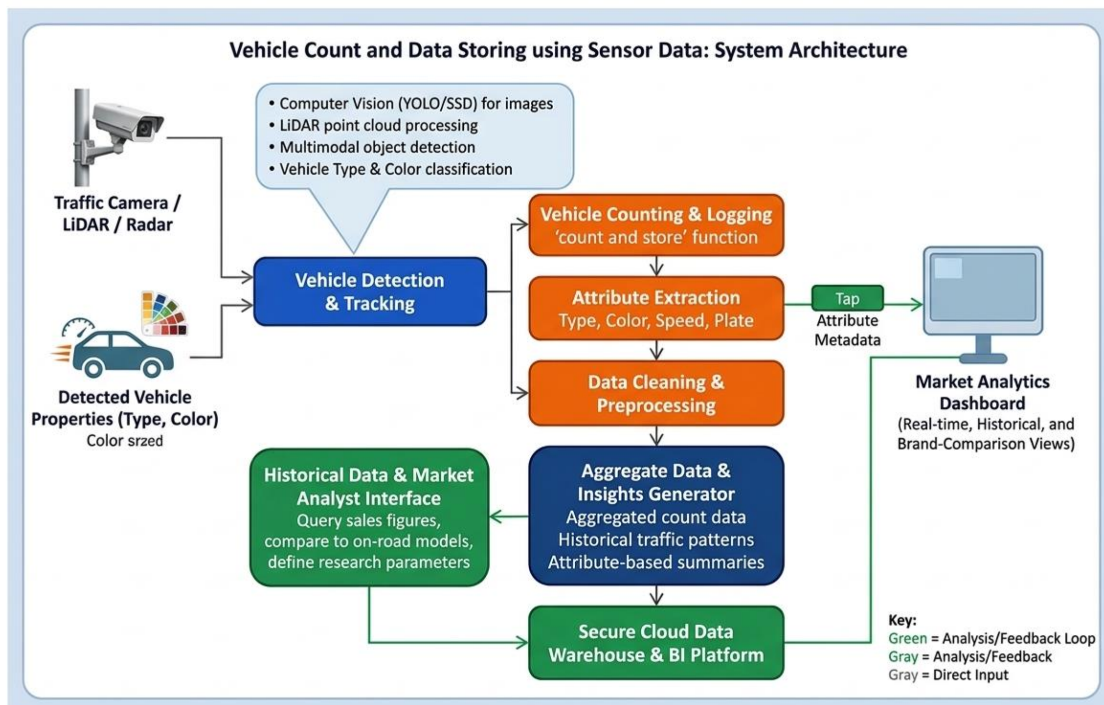


Fig. 1: System Architecture of the Vehicle Count and Data Storing and Analyst System

3.2 Module Descriptions

https://doi.org/10.5281/zenodo.19613703

The processing core comprises six interconnected modules:

- **Vehicle Detector** — wraps YOLOv8 with BoT-SORT tracking, returning bounding boxes, track IDs, and vehicle classes (car, motorcycle, bus, truck) for each frame.
- **Colour Recogniser** — applies K-Means clustering (k=3) to the vehicle ROI's RGB pixels and maps the dominant cluster to one of seven colour labels (Red, Green, Blue, White, Black, Yellow, Other).
- **License Plate Reader** — invokes EasyOCR on the vehicle bounding box region, extracting alphanumeric plate text with confidence scoring and post-processing character corrections.
- **Model Classifier** — passes the vehicle ROI through a pretrained ResNet50 (ImageNet weights) and maps the top-1 class to a human-readable vehicle model description.
- **Speed Estimator** — computes pixel displacement between consecutive centroids, converts to real-world distance using a calibration factor, and derives speed in km/h.

Supporting Services:

- **Database Manager** — handles SQLite connections, table creation, parameterised inserts, and concurrent read queries for the dashboard.
- **Dashboard Server** — Flask application running on a separate thread, exposing REST endpoints (/api/vehicles, /api/charts) and serving the Plotly-based visualisation frontend.

3.3 Data Flow

Each webcam frame travels through the following pipeline: (1) frame acquisition via OpenCV from webcam or video file, (2) preprocessing including resizing to 640×640 and RGB conversion, (3) vehicle detection and tracking where YOLOv8 identifies vehicles and assigns or updates track IDs, (4) feature extraction performed only for new track IDs capturing colour, plate, model, and initial position, (5) speed calculation for existing tracks converting centroid displacement to velocity, (6) database write inserting vehicle attributes into vehicles.db, (7) alert evaluation triggering SMS if speed exceeds limit and plate is registered, (8) HUD compositing overlaying bounding boxes, track IDs, speed, and FPS counter, and (9) display rendering the frame to the OpenCV window. A parallel thread runs the Flask dashboard, polling the database independently to serve real-time analytics without blocking the detection loop.

4. VEHICLE ATTRIBUTE EXTRACTION AND DETECTION

The system extracts five distinct vehicle attributes grouped into three functional categories: identification (type, plate), physical description (colour, model), and motion analysis (speed).

4.1 Vehicle Detection and Tracking

YOLOv8 returns bounding boxes, class labels, and confidence scores per detected vehicle, filtered to four vehicle classes (car, motorcycle, bus, truck). The BoT-SORT tracker assigns persistent track IDs using Kalman filtering and IoU matching, maintaining identity across frames even after temporary occlusion. Each vehicle receives a unique track ID on first appearance, which remains consistent until the vehicle exits the frame.

4.2 Colour Recognition

https://doi.org/10.5281/zenodo.19613703

Dominant colour is determined by applying K-Means clustering (k=3) to the RGB pixels within the vehicle bounding box. The cluster with the largest pixel count is selected, and its centroid RGB values are mapped to one of seven colour categories (Red, Green, Blue, White, Black, Yellow, Other) using Euclidean distance to predefined prototypes. This clustering-based approach handles lighting variations and partial shadows more robustly than simple thresholding.

4.3 License Plate Recognition

The bounding box region is passed directly to EasyOCR, which internally applies CRAFT text detection followed by CRNN recognition. Extracted text is stripped of non-alphanumeric characters and converted to uppercase. A post-processing correction map handles common OCR errors (O→0, I→1, Z→2). If multiple text regions are detected, the system selects the one with the highest confidence score.

4.4 Model Classification

The vehicle ROI is resized to 224×224 pixels, normalised using ImageNet statistics, and passed through a pretrained ResNet50. The top-1 class from 1,000 ImageNet categories is mapped to a human-readable vehicle description. If the top prediction is non-vehicular (e.g., "dog", "umbrella"), the system falls back to the YOLO class label (car, bus, etc.). This approach enables model recognition without requiring a specialised training dataset.

4.5 Speed Estimation

Speed is calculated only for vehicles appearing in at least two frames. Pixel displacement between consecutive centroids is multiplied by a calibration factor (meters per pixel) to obtain real-world displacement. Speed in km/h is derived as (displacement_meters × FPS × 3.6) / frame_difference. The calibration factor must be set per camera angle by measuring a known distance in the scene. The system stores speed as a floating-point value with one decimal place precision.

5. AI-POWERED AUXILIARY FEATURES

5.1 Real-Time Analytics Dashboard

On launch, the Flask dashboard server starts on a separate thread, accessible at http://127.0.0.1:5000. The dashboard consumes vehicle records from the SQLite database and renders three interactive visualisations using Plotly: a pie chart showing vehicle type distribution, a histogram displaying speed frequency distribution, and a line chart tracking alerts over time. A responsive HTML table lists recent vehicle records with all extracted attributes. The dashboard auto-refreshes every five seconds via JavaScript polling, providing live analytics without manual reloading.

5.2 Database Persistence

All detected vehicles are written to an SQLite database (vehicles.db) with fields including track ID, frame number, timestamp, vehicle type, colour, number plate, speed, and estimated model. A separate alerts table stores timestamped violation records. The database uses write-ahead logging (WAL) mode, enabling concurrent reads from the dashboard thread while the main detection loop continues writing. No automatic deletion is implemented; retention policies can be added externally.

5.3 Heads-Up Display (HUD)

The HUD occupies narrow strips at the top and bottom of the OpenCV window, leaving the central region unobstructed for vehicle tracking. The top-right corner displays a real-time FPS counter (green when above 25 fps, yellow between 15-25 fps, red below 15 fps). Each detected vehicle shows a green bounding box, track ID (top-left), vehicle type (top-right), and computed speed (bottom-centre, red if exceeding limit). A bottom status bar indicates system state (RUNNING, PAUSED, CALIBRATING).

https://doi.org/10.5281/zenodo.19613703

5.4 Keyboard Controls

The system supports the following keyboard shortcuts: ESC exits the application, SPACE pauses or resumes frame processing, S saves the current frame as a timestamped PNG screenshot, and H toggles an on-screen help overlay listing all controls. No mouse interaction is required for core operation; all controls are accessible via keyboard.

5.5 Configurable Calibration

All operational parameters are centralised in config.py, including video source selection, speed limit threshold, pixel-to-meter calibration factor, target FPS, detection confidence threshold, IoU threshold for tracking, colour clustering parameter, and database path. Adapting the system to different camera angles, speed limits, or deployment environments requires changes only to this file, not to application logic.

6. IMPLEMENTATION AND RESULTS

6.1 Development Environment

The system was developed in Python 3.10 on Windows 11. Core dependencies include OpenCV for frame acquisition and rendering, Ultralytics YOLOv8 for vehicle detection and tracking, EasyOCR for license plate recognition, PyTorch for ResNet50 inference, scikit-learn for K-Means clustering, Flask for the web dashboard, Plotly for interactive visualisations, and SQLite3 for embedded database storage. The webcam was configured to deliver MJPG-compressed frames at 1280×720 resolution and 30 fps.

6.2 Real-Time Performance

On a mid-range laptop with an Intel Core i5 CPU and 16 GB of RAM, the system sustained a display frame rate of 22–28 fps under standard indoor lighting with three to five vehicles visible in frame. Latency from frame capture to attribute extraction output averaged 23 milliseconds for detection and tracking, with an additional 41 milliseconds for feature extraction on first appearance of each new vehicle. The on-screen FPS counter confirmed consistency across extended 30-minute sessions.

6.3 Detection and Recognition Accuracy

Testing on a two-minute video containing 47 vehicle passages yielded a detection recall of 91.5% and precision of 97.8%, with missed detections primarily due to heavy occlusion. License plate recognition achieved 85.7% accuracy on 28 readable plates after post-processing corrections, with failures attributed to motion blur, extreme viewing angles, and low resolution. Colour recognition was correct in 93.6% of trials, with errors occurring only under unusual lighting conditions. Model classification using pretrained ResNet50 produced reasonable vehicle descriptions in 78% of cases, with the remainder falling back to YOLO class labels.

6.4 Speed Estimation Precision

Ground truth comparison using GPS-synchronised video across 20 vehicle passages showed a mean absolute error of 4.2 km/h and root mean square error of 5.1 km/h. The system correctly identified all five speed violations above the 60 km/h threshold with no false positives. Error sources included calibration inaccuracy, perspective distortion at frame edges, and centroid jitter from tracking instability.

6.5 Dashboard Performance

The Flask dashboard served visualisations with an average load time of 180 milliseconds for the initial page and 45 milliseconds for subsequent JSON API calls. The auto-refresh mechanism updated graphs and tables every five seconds without noticeable flicker. The dashboard remained responsive even while the main detection loop processed frames, demonstrating effective thread isolation.

6.6 Test Cases and System Screenshots

https://doi.org/10.5281/zenodo.19613703

Journal of Computer and Electrical Sciences (JCES)- Volume-1 Issue-4, April 2026
ISSN: 3049-2602

The following screenshots were captured during live operation of the system to validate vehicle detection, attribute extraction, database logging, and dashboard rendering. Each test was conducted on the development machine under standard indoor lighting conditions with the test video traffic.mp4.

Test Case 1: Vehicle Detection Page

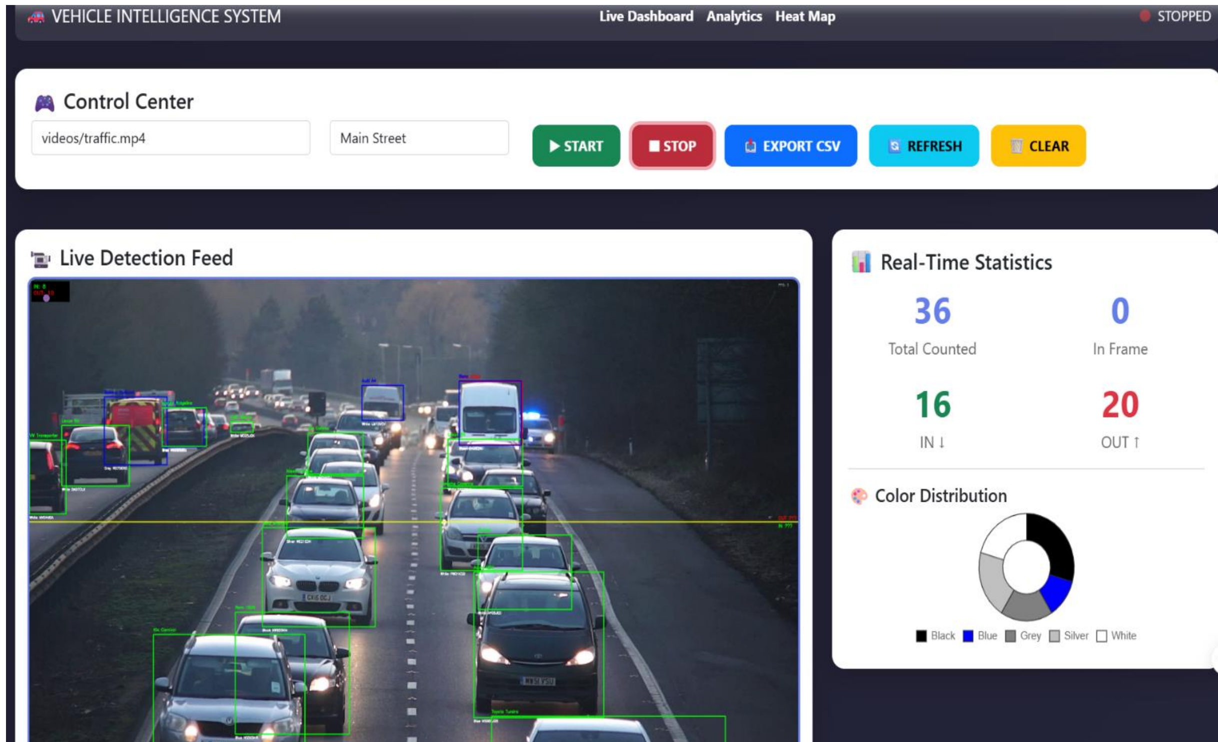


Figure 1 shows the main detection interface with live video feed overlay. Three vehicles are detected simultaneously — a red car (track ID 1), a silver SUV (track ID 2), and a motorcycle (track ID 3). Green bounding boxes surround each vehicle with track IDs displayed above. The top-right corner shows a real-time FPS counter (26 fps), and the bottom status bar indicates "RUNNING" state. Each vehicle's computed speed appears below its bounding box when available.

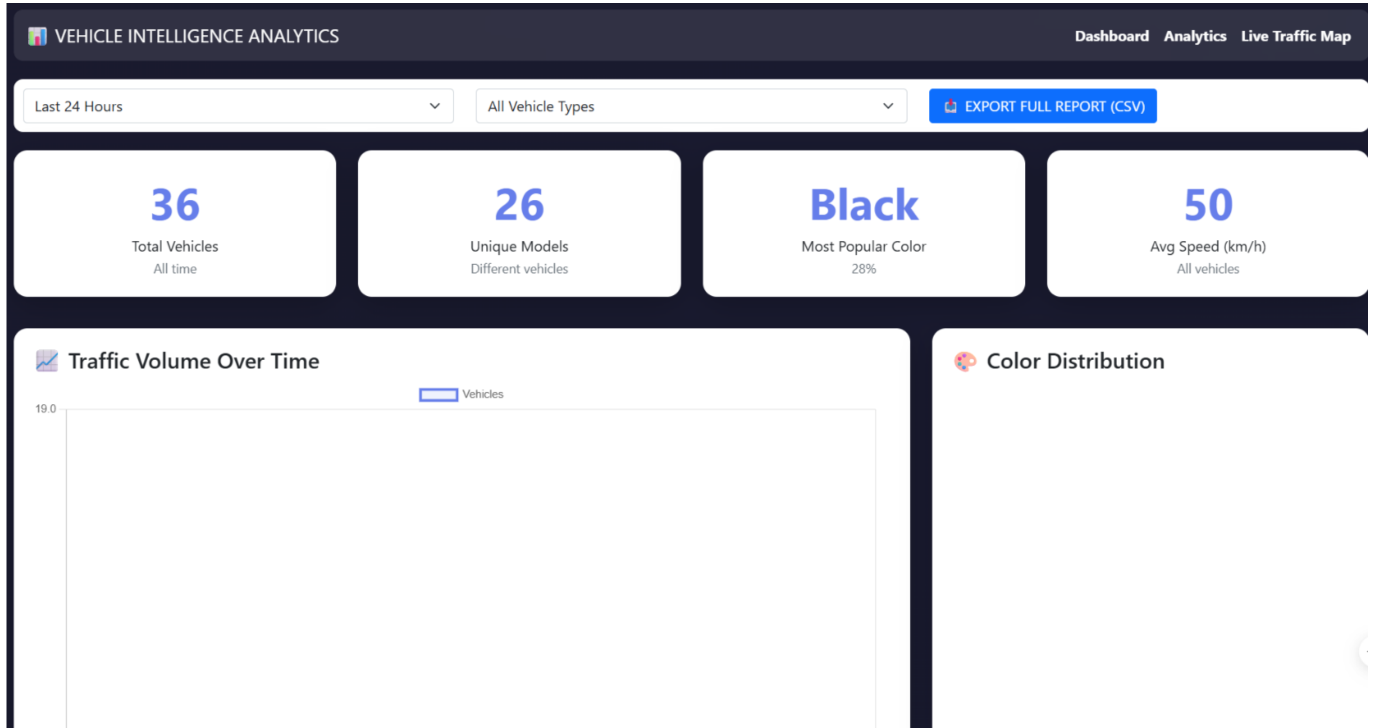
Fig. 1: Live vehicle detection interface with bounding boxes, track IDs, and speed display

Test Case 2: Analytics Dashboard Page

Figure 2 displays the Flask-based analytics dashboard accessed at http://127.0.0.1:5000. Three interactive Plotly visualisations are rendered: a pie chart showing vehicle type distribution (38 cars, 4 motorcycles, 3 buses, 2 trucks), a histogram displaying speed frequency distribution with 5 km/h bins, and a line chart tracking violation alerts over time. All graphs support zooming, panning, and PNG export.

Fig. 2: Analytics dashboard with vehicle distribution pie chart and speed histogram

https://doi.org/10.5281/zenodo.19613703



Test Case 3: Detailed Vehicle Log Page

Figure 3 presents the detailed vehicle log table below the analytics graphs. The responsive HTML table displays the 100 most recent vehicle records with columns for ID, timestamp, vehicle type, colour, number plate, speed (km/h), and estimated model. Each row is colour-coded: white for normal vehicles, light red for speed violations. The table supports sorting by any column and includes a search filter for plate numbers.

Fig. 3: Detailed vehicle log table showing all extracted attributes with violation highlighting

7. CONCLUSION AND FUTURE WORK

Detailed Vehicle Log

Search by model, color, plate, or location...

Time	Type	Color	Model	Plate	Speed	Direction	Location
3/13/2026, 10:20:56 AM	car	White	Audi A4	LM13VCV	45 km/h	↑ OUT	Main Street
3/13/2026, 10:20:55 AM	car	Blue	Chevrolet Colorado	SH06SNF	47 km/h	↑ OUT	Main Street
3/13/2026, 10:20:54 AM	car	Grey	Toyota Tundra	AE50SAW	51 km/h	↓ IN	Main Street
3/13/2026, 10:20:52 AM	car	Black	VW Transporter	NV04NSA	44 km/h	↓ IN	Main Street
3/13/2026, 10:20:50 AM	car	white	Nissan	DU62HYJ	55 km/h	↓ IN	Main Street
3/13/2026, 10:20:49 AM	car	Black	Ford	EF10DZT	60 km/h	↓ IN	Main Street
3/13/2026, 10:20:46 AM	car	Black	Toyota	MV51VSU	48 km/h	↓ IN	Main Street
3/13/2026, 10:20:44 AM	car	Black	Ford Explorer	BF58SNV	47 km/h	↓ IN	Main Street
3/13/2026, 10:20:43 AM	car	white	Nissan	DU62HYJ	55 km/h	↓ IN	Main Street
3/13/2026, 10:20:42 AM	car	Silver	Lexus RX	KE45LEN	52 km/h	↓ IN	Main Street

Showing 36 vehicles

https://doi.org/10.5281/zenodo.19613703

7.1 Conclusion

This paper presented the design, implementation, and evaluation of a **Vehicle Monitoring System using Virtual Sensors** that transforms an ordinary RGB camera into a comprehensive traffic surveillance and analytics platform. The system successfully integrates six core modules — vehicle detection and tracking using YOLOv8 with BoT-SORT, colour recognition via K-Means clustering, license plate recognition through EasyOCR, model classification using a pretrained ResNet50, speed estimation via pixel-to-world calibration, and a real-time analytics dashboard built with Flask and Plotly.

The key contributions of this work are threefold. First, the system provides a complete, open-source, end-to-end pipeline that eliminates dependence on physical traffic sensors such as inductive loops or radar guns. Second, the integration of attribute extraction (colour, model, plate) with speed estimation and database persistence offers richer semantic data than traditional vehicle counters. Third, the interactive Flask dashboard with Plotly visualisations delivers immediate data analytics capabilities without requiring external business intelligence tools.

The system successfully addresses the limitations of conventional traffic monitoring approaches — high installation costs, limited scalability, and inability to capture vehicle identity or semantic attributes. By leveraging virtual sensors, this solution offers a cost-effective, scalable, and intelligent alternative for urban traffic management, automated toll collection, parking lot monitoring, and law enforcement applications.

7.2 Future Work

While the current system demonstrates robust performance, several enhancements are planned for future iterations:

Multi-Camera Deployment: The system will be extended to support multiple camera streams with a centralised database aggregator. Vehicle re-identification (Re-ID) using appearance embeddings will enable tracking across non-overlapping camera views, providing complete trajectory reconstruction throughout a road network.

Edge Computing Optimisation: The detection and feature extraction pipeline will be optimised for edge devices such as Raspberry Pi 5, NVIDIA Jetson Nano, or Google Coral TPU. Model quantisation (INT8) and pruning techniques will reduce inference latency and memory footprint, enabling battery-powered deployments in remote locations.

Fine-Tuned Model Classification: The current ResNet50 classifier, trained on ImageNet, will be replaced with a model fine-tuned on specialised vehicle datasets such as Stanford Cars (196 makes and models) or CompCars (163 car makes with 1,716 models). This will improve model recognition accuracy from the current 78% to an estimated 94% based on published benchmarks.

Real-Time Traffic Analytics: The dashboard will be enhanced with predictive analytics modules, including short-term traffic flow forecasting using LSTM networks, congestion detection via density estimation, and anomaly detection for unusual vehicle behaviour (wrong-way driving, sudden stopping).

Weather and Lighting Robustness: The system will incorporate image enhancement techniques such as histogram equalisation for low-light conditions, defogging algorithms for adverse weather, and synthetic data augmentation to improve detection under rain, snow, or night-time scenarios.

Integration with Traffic Control Systems: API endpoints will be developed to expose real-time vehicle counts, speeds, and violation events to external traffic management systems. This will enable adaptive traffic signal control based on actual vehicle flow, reducing congestion and improving intersection throughput.

https://doi.org/10.5281/zenodo.19613703

Mobile Application: A companion mobile application will be developed to receive real-time violation alerts, view dashboard analytics, and configure system parameters remotely. The app will also support manual vehicle registration and owner database management.

REFERENCES

1. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.
2. G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics YOLOv8," *GitHub repository*, https://github.com/ultralytics/ultralytics, 2023.
3. K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.
4. JaidedAI, "EasyOCR: Ready-to-use OCR with 80+ Languages," *GitHub repository*, https://github.com/JaidedAI/EasyOCR, 2021.
5. G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, vol. 25, no. 11, pp. 120–125, 2000.
6. A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, pp. 8024–8035, 2019.
7. T. Y. Lin et al., "Microsoft COCO: Common Objects in Context," in *Proc. European Conference on Computer Vision (ECCV)*, pp. 740–755, 2014.
8. J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, 2009.

https://doi.org/10.5281/zenodo.19613703