SWE30011 – IoT programming

Project topic:

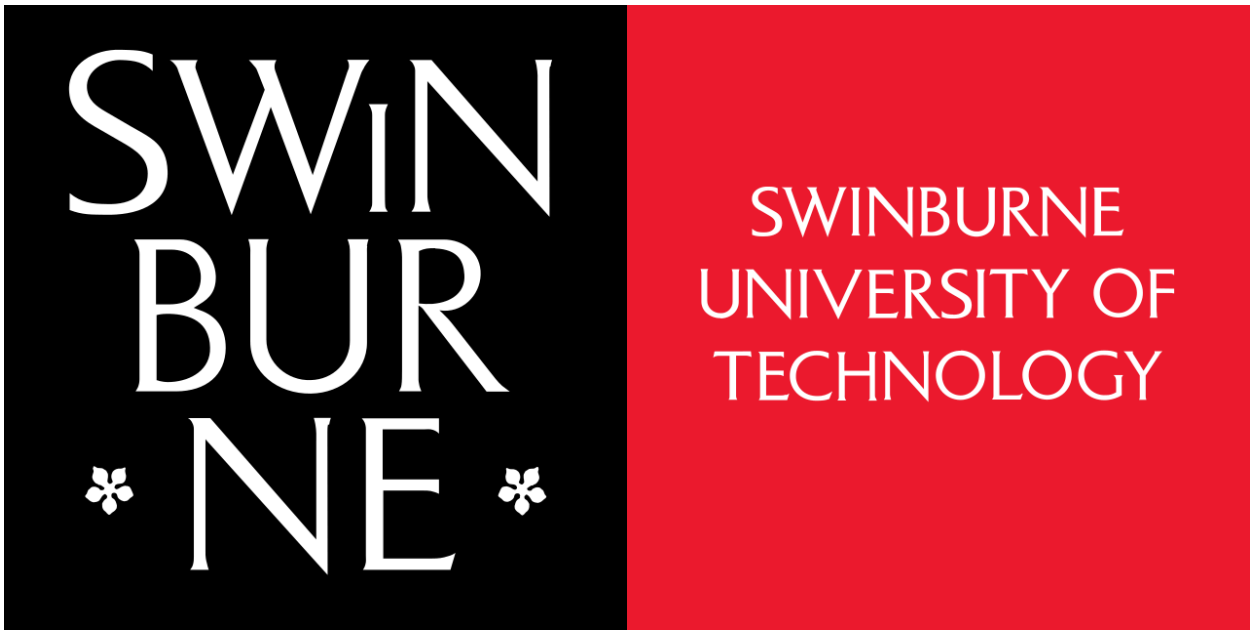
Smart Parking Optimization Using Edge Computing and Deep Learning for Autonomous Routing and Plate Recognition

Student Name:

Nguyen Dinh Dung- 104772138

Nguyen My Hanh- 104221122

Le Hoang Minh- 104997504



The demonstration video is at:
https://www.youtube.com/watch?v=Q2rw31VUKwc

Table of Contents

Introduction

Urban parking is a growing smart-city challenge: drivers can waste on average many hours per year searching for a spot, according to Khan [1]. This inefficiency leads to increased congestion, higher fuel consumption, and higher emissions. Traditional smart-parking solutions typically install an IoT sensor (e.g. ultrasonic or IR) at each parking bay to detect occupancy [2]. While straightforward, this per-slot approach is costly at scale and provides only a binary occupied/free signal, without providing contextual details (such as vehicle type, destination, etc.). Computer-vision-based systems use cameras and deep CNNs to identify free spaces [1], but they require heavy network bandwidth and centralized processing.

In contrast, we propose an edge-based deep learning approach. Each parking area is instrumented with a few ultrasonic sensors; their data is fused with a time-series prediction model (an LSTM) running on a Raspberry Pi 3 edge node. The LSTM predicts future availability based on historical data (fields like ArrivalTime, DepartureTime, BayId, AreaName). By executing the model locally, the system avoids cloud latency and bandwidth bottlenecks, thereby improving performance.

Our approach innovatively combines real-time occupancy detection from a minimal number of well-placed ultrasonic sensors with future state prediction, enabling autonomous and optimized routing to available parking bays. The implemented model achieves a notably low mean absolute error (MAE) of 9 spots and root mean squared error (RMSE) of 13 spots, representing only 5 to 6% prediction error across more than 500 parking spots per area. Furthermore, the entire routing operation executes within a 3-5 second latency, demonstrating practical usability for smart city applications.

To manage the entry and exit, our system implements an Automatic License Plate Recognition (ALPR) subsystem specifically designed for cars with Vietnamese plates. This subsystem is two-stage, with YOLOv8 handling detection and LPRnet, a customized Optimal Character Recognition (OCR), recognizing plates. This module runs at the parking area's entry and departure points, automatically detecting vehicles, recording timestamps, and updating the occupancy status in real time.. High recognition accuracy in real-world situations is ensured by the Vietnamese-specific design, which takes into account the distinctive alphanumeric patterns and formatting conventions of regional license plates. In order to enable real-time vehicle identification without causing delays at entry/exit gates, our system aims for 95% recognition accuracy under typical lighting conditions with processing latency maintained between 100 and 150 milliseconds.

Furthermore, our solution has a complete monitoring dashboard that provides an overall review of system performance. This dashboard monitors crucial data like ALPR recognition accuracy, LSTM prediction error (MAE and RMSE), end-to-end routing latency, and sensor reliability. The

dashboard allows administrators to monitor system health, detect bottlenecks, and take data-driven actions to better optimize the system.

This hybrid approach (sensor fusion + edge inference + automated vehicle management + performance monitoring) differs from traditional cloud-hosted or sensor-only solutions. For example, previous cloud-centric systems send each bay's sensor signal to the cloud via a Pi [1], whereas our system evaluates data on the Pi to provide real-time routing direction while also regulating vehicle flow via ALPR. Modern research demonstrates that deep learning (e.g., LSTM) may efficiently capture the temporal patterns of parking usage while producing low prediction errors (MAE, RMSE) [3]. Similarly, YOLOv8 and LPRNet have shown cutting-edge performance in license plate detection and identification tasks, making them perfect for automated access control in parking lots.

This project addresses the pressing urban problem of inefficient parking search and presents a novel, scalable, and effective solution that surpasses existing systems by combining edge computing, sensor fusion, time-series deep learning, automated vehicle management, and intelligent monitoring. Such advancements are crucial for reducing urban congestion, lowering emissions, and enhancing driver experience amidst accelerated urbanization and growing demands on parking infrastructure.

Conceptual Design

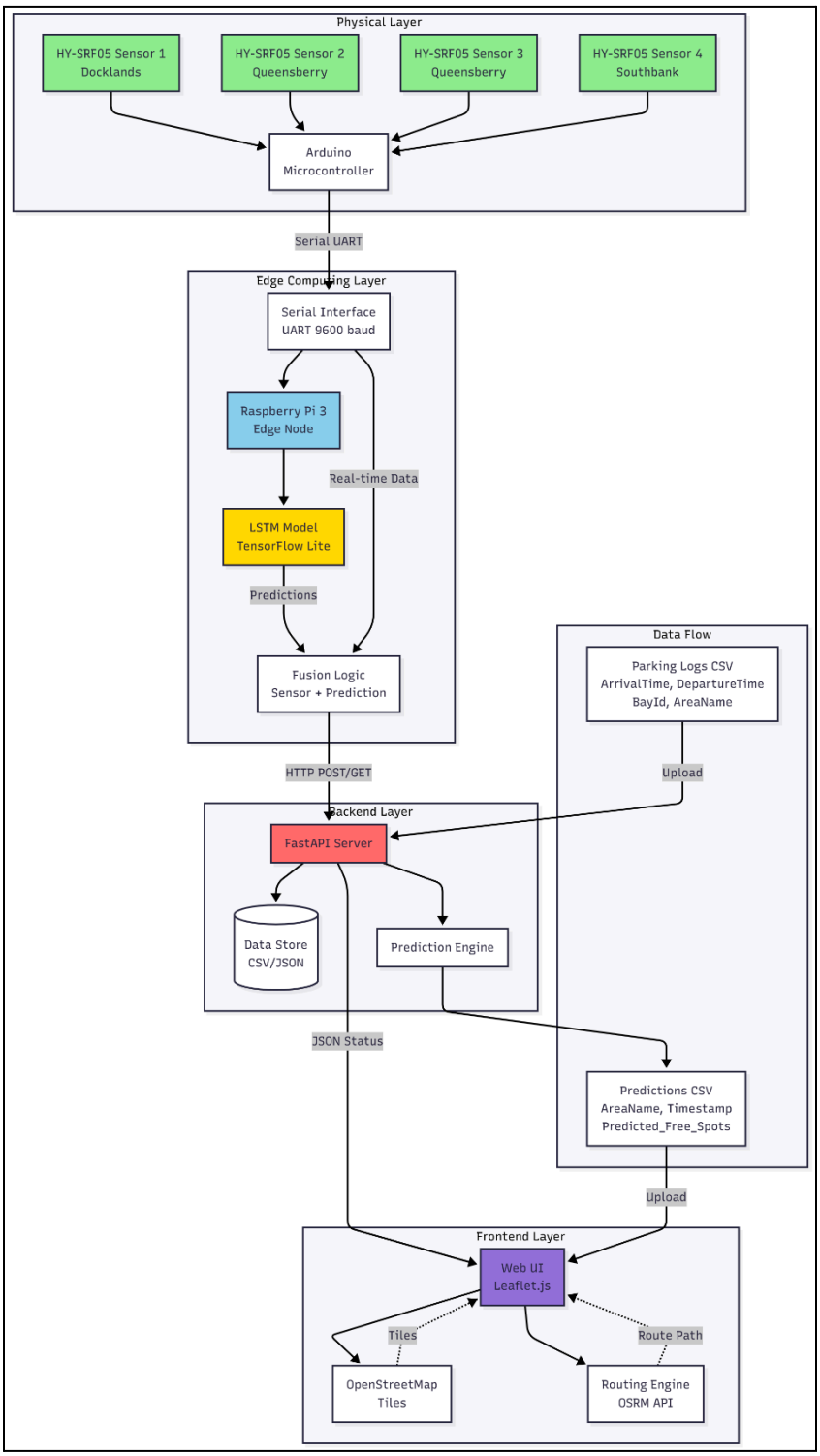


Figure 1. Overall system design

The system architecture comprises physical sensors, an edge node (Raspberry Pi 3), a prediction model (LSTM), a backend API, and a web-based routing frontend. Ultrasonic distance sensors (HY-SRF05) are placed at key parking bays (Docklands: 1 sensor; Queensberry: 2; Southbank: 1). These sensors detect vehicle presence via sonar. An Arduino reads each HY-SRF05 in sequence and sends occupancy signals to the Pi over a serial link (Figure 1). On the Pi, the LSTM model (converted to TensorFlow Lite) periodically forecasts parking availability. A logic module fuses the live sensor state with the model's output: if a sensor reports occupied, that bay is marked busy; otherwise, the model's predicted occupancy informs routing decisions. The Pi runs a FastAPI server to upload data (sensor states and predictions) as CSV/JSON to a cloud backend.

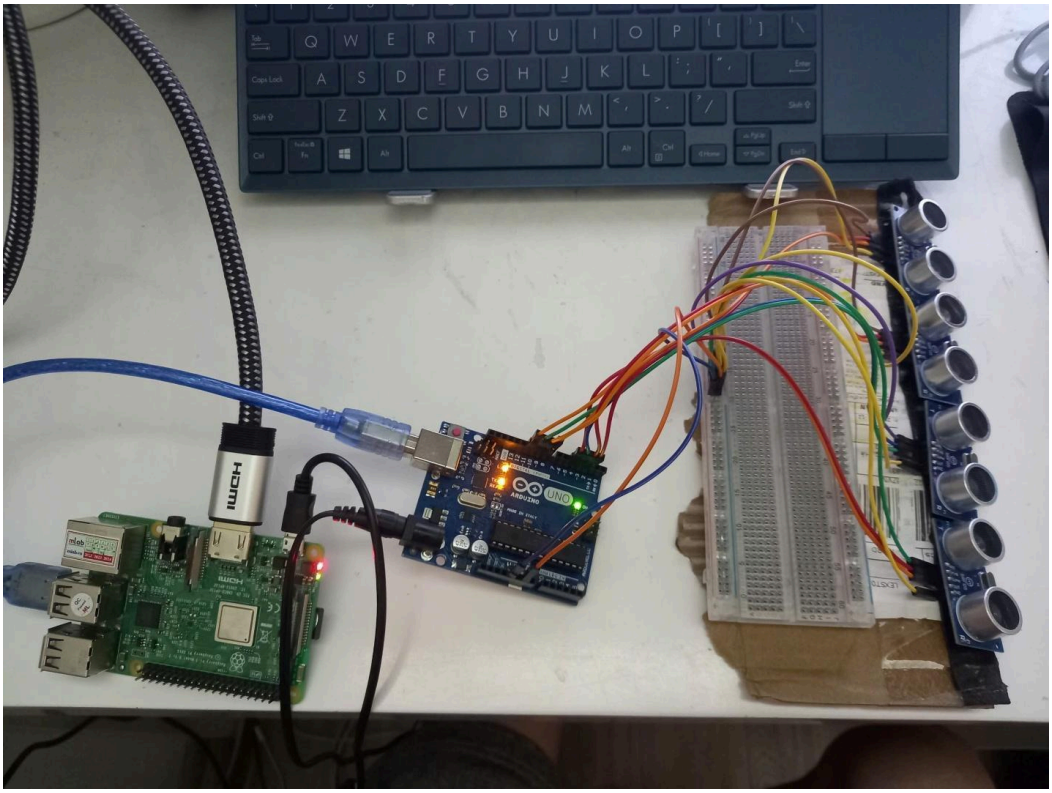


Figure 2. Hardware Architecture

The backend API stores and serves the current occupancy data. A Leaflet.js web application fetches this data and displays color-coded markers on an OpenStreetMap:

Green: >200 slots

Yellow: <200 slots

Red: <100 slots

Black: out of slots

Drivers interact with the frontend: they can click markers for details or choose “auto-route” to let the system pick the nearest available bay. The routing uses OSM street data to compute a driving path. This overall design leverages edge computing to minimize latency [4] and fuses sensor inputs with predictive modeling for smarter routing.

Sensors feed real-time occupancy status into the microcontroller, which sends data via serial to the Raspberry Pi. The LSTM model predicts each area’s number of free slots based on historical usage data. The rule “any sensor triggered ⇒ area full” ensures that a single detected car causes the whole area to be marked as occupied. The Web UI then displays one map marker per area with a color-coded status.

Task Breakdown

Key project tasks are summarized in the table below, organized by AI (prediction) and IoT (sensing/backend) modules:

Name	Component/Module	Task	Details/Description
Nguyen Dinh Dung	Dataset (AI)	Data Collection	Provided raw logs: ArrivalTime, DepartureTime, BayId, AreaName.
	Preprocessing (AI)	Time-series Aggregation	Aggregate logs into 5-minute interval occupancy by bay per area.
	Model Training (AI)	LSTM Development	Two-layer LSTM network trained on historical data, using 5-fold CV.
	Evaluation (AI)	Metrics Computation	Compute Mean Absolute Error (MAE) and RMSE overall and per area [3].
	Deployment (AI)	TFLite Conversion	Convert the trained model to TensorFlow Lite for Raspberry Pi inference.
	Sensors (IoT)	Hardware Installation	Deploy HY-SRF05 ultrasonic sensors (Docklands:1, Queensberry:2, Southbank:1).
	Sensor Interface	Arduino Connection	Connect sensors to Arduino; implement a scan loop to avoid crosstalk.
	Edge Node (IoT)	Raspberry Pi Setup	Install OS, Python, TensorFlow Lite runtime on Pi for model inference.

Name	Component/Module	Task	Details/Description
	Inference (IoT)	On-Pi Prediction	Run LSTM inference on-device; apply override logic with live sensor data.
	Backend (IoT)	FastAPI Server	Develop a REST API to receive data (CSV upload of predictions) and serve JSON to the UI.
	Frontend (UI)	Map Visualization	Leaflet.js map with color-coded bay markers (green=free, red=occupied).
	Routing Logic (UI)	Navigation Controls	Manual route: click bay; Auto-route: compute nearest free bay (using OSM).
	Detect and capture plates	Integrated 2 models: Yolo and LPRNet	Integrated 2 models onto Raspberry Pi 3, synchronized the camera with the sensor data
Le Hoang Minh	Dataset (AI)	Prepare dataset	Find a suitable dataset and separate it into train, test, and validation folders with annotations
	Model training (AI)	YOLOv8 and LPRNet development	Two-stage system that detects the plate, crops, and then recognizes the text on the cropped image.
	Evaluation (AI)	Metrics computation	Compile the test file to test the model against new data for generalization.
	Database	Schema and connection	Use SQLite to run the script and connect to the database. This ensures the system can upload new records to it.
	Dashboard	Table creation	Visualize the output of the system (4 tables in Database, Implementation) by calling the API.
Nguyen My Hanh	Monitoring Dashboard	Visualize system status	Flask web app route / generates interactive charts for browser: 10-min step chart for IoT vs AI status, discrepancy bar chart per area, and user feedback bar chart.

Name	Component/Module	Task	Details/Description
	Data Loader & Normalizer	Load CSV and clean JSON files	Reads parking records, AI predictions, and user feedback. Standardizes column names, converts timestamps to datetime, and fills missing area IDs.
	Visualization Module	Generate charts	Uses Plotly Express: step chart for occupancy timeline (horizontal-vertical steps to show status changes), bar charts for discrepancy and feedback metrics, converted to HTML for Flask.
	IoT vs AI Comparison	Aggregate and compare occupancy	Computes IoT occupancy from parking CSV, resamples AI predictions to 10-minute intervals, merges datasets, and calculates discrepancies and simplified accuracy.
	Accuracy & Feedback Analysis	Evaluate performance	Computes simplified IoT vs AI accuracy (1 – discrepancy rate), presents numeric user feedback averages, all displayed in the dashboard for system monitoring.

Table 1: Tasks breakdown

In particular, the AI tasks involved training a sequence model on parking logs. We aggregated raw arrival/departure records into a regular time series (5-minute bins) for each BayId and AreaName, then trained a 2-layer LSTM using 5-fold cross-validation. Performance is measured by MAE and RMSE on held-out folds. For the IoT tasks, the sensor layout spanned three areas with a total of 4 sensors (one or two per area). A FastAPI-based backend was implemented to accept data from the edge node. The LSTM model was deployed in TensorFlow Lite on the Raspberry Pi, enabling on-device inference. Edge processing also includes basic logic-based routing: for example, if a bay's sensor reports occupied, the slot is excluded even if the model had predicted it free. Finally, the system generates CSV outputs (predicted occupancy by area), which can be uploaded via the frontend to update the dataset or share predictions.

Implementation

Edge-based LSTM parking prediction system

The implementation spans AI, hardware integration, edge software, backend services, and frontend logic:

AI: employs the On-street Car Parking Sensor Data (Melbourne, 2025) [5], a large real-world dataset containing high-resolution temporal logs of parking bay activity (over 20 million rows). Each record includes a unique parking bay identifier, GPS coordinates, sensor-detected arrival and departure timestamps, and vehicle-presence indicators derived from embedded ground sensors. To construct a reliable forecasting pipeline (Figure 3), the raw timestamps were first converted into standardized datetime objects, with invalid entries removed to ensure temporal consistency. The busiest geographical area was then identified automatically using frequency analysis, allowing the model to focus on a high-density subset with larger temporal variability. Within this area, arrival and departure times were aligned to a five-minute grid, and occupancy events were aggregated to compute a continuous time series of total occupied and free parking bays. Additional temporal features—hour of day, day of week, and weekend indicators—were incorporated to capture behavioural and cyclical patterns inherent in urban parking dynamics. All features were then normalized using MinMax scaling, and sliding windows were generated to form supervised learning sequences for short-term forecasting.

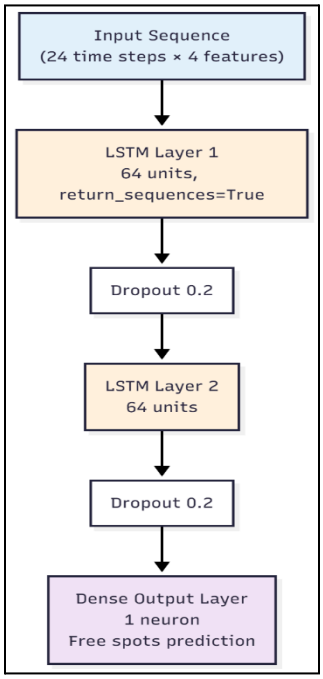


Figure 3. Model architecture

The prediction model was constructed using a stacked Long Short-Term Memory (LSTM) neural network designed to capture short- and long-range temporal dependencies. The architecture consisted of two LSTM layers with dropout regularization, followed by a densely connected output layer predicting the number of free parking spots 30 minutes ahead (Figure 3). A time-series-aware cross-validation scheme (TimeSeriesSplit) was employed to prevent information leakage and ensure robust performance assessment. After model tuning and validation across five folds, a final model was trained on the full training set for 50 epochs. Evaluation on held-out test data demonstrated strong predictive accuracy, achieving a mean absolute error (MAE) of approximately 9.12 free spots and a root mean squared error (RMSE) of 13.11, indicating that the LSTM model effectively learned the temporal occupancy patterns and can support real-time parking availability prediction in dense urban environments.

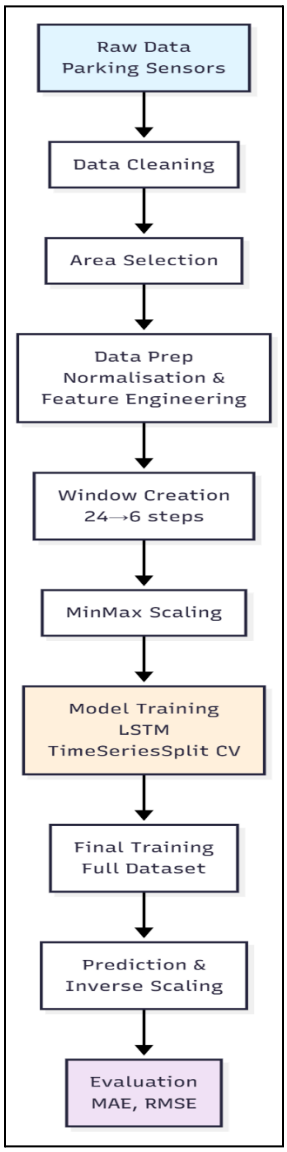


Figure 4. Workflow Diagram (Data → Preprocessing → Modeling → Evaluation)

IoT Architecture: 4 HY-SRF05 ultrasonic sensors are wired to a common Arduino microcontroller. The Arduino sequentially triggers each sensor (to avoid echo interference) and reads the distance. If the measured distance is below a threshold, the bay is considered occupied. These readings are sent over a serial (UART) connection to the Raspberry Pi 3 (Edge Node). This setup follows standard IoT parking designs [4]. The Pi polls the Arduino, receives occupancy signals (Figure 4), and publishes them to the backend via HTTP.

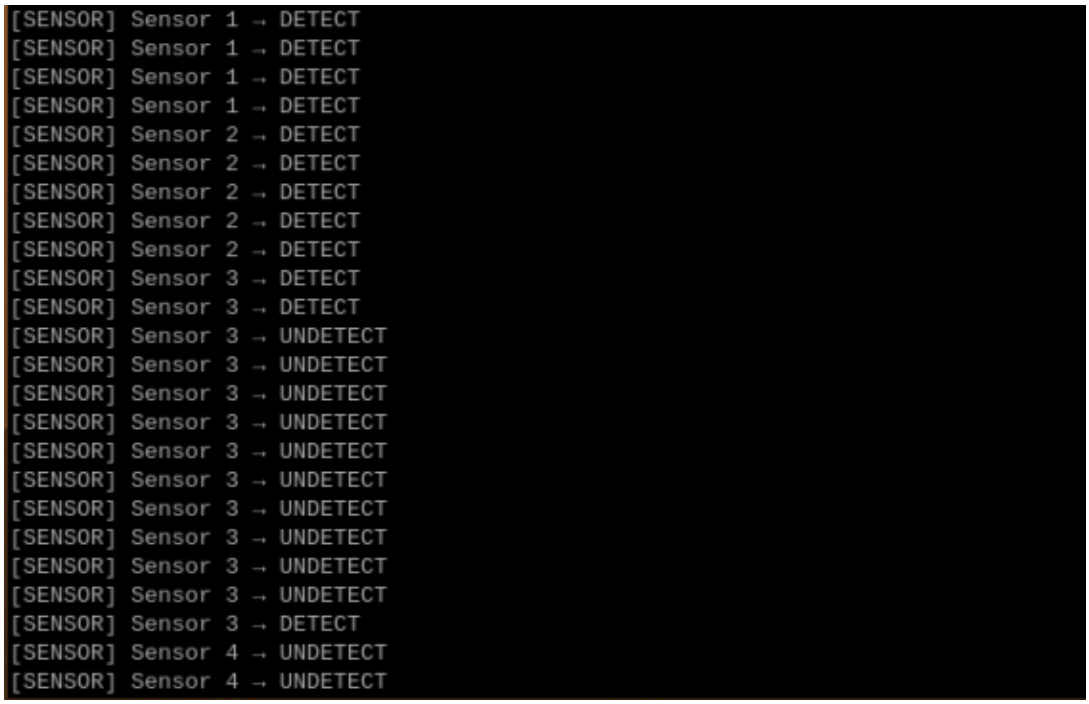


Figure 5. Sensor data received on Pi

Edge Inference & Override Logic: On the Pi, a Python process runs the TensorFlow Lite (Figure 6) version of the trained LSTM model. The model predicts parking availability for each area (e.g. number of free slots left). Prediction runs in a loop (e.g. every 5 minutes) or is triggered by new data. After getting predictions, the Pi applies override logic: if the real-time sensor reports a bay as occupied, it forces that bay to “occupied” regardless of the prediction. This fusion ensures reliability: sensor data always takes precedence for the immediate state. The fused state is then formatted (as CSV or JSON) and sent via the FastAPI REST API.

Figure 6. TFLite integration

```
# ============= CONFIGURATION =============
BASE_DIR = os.path.abspath(os.path.dirname(__file__))
CONFIG_PATH = os.path.join(BASE_DIR, "config.json")
MODEL_PATH = os.path.join(BASE_DIR, "lstm_parking_model.keras")
SCALER_PATH = os.path.join(BASE_DIR, "scaler_parking.save")
AREAS_GEO = os.path.join(BASE_DIR, "areas_geo.csv")
PRED_CSV = os.path.join(BASE_DIR, "predicted_free_spots_top3areas.csv")

with open(CONFIG_PATH, "r") as f:
    cfg = json.load(f)

SERIAL_PORT = cfg.get("serial_port", "/dev/ttyACM0")
BAUDRATE = cfg.get("baudrate", 9600)
SENSOR_TO_AREA = cfg.get("sensor_to_area", {})
```

Figure 7. Trained LSTM model on backend

Sensing and Actuation: The HY-SRF05 sensors use sonar pulses to measure distance to the nearest object. They have separate Trigger and Echo lines (handled by Arduino). The Arduino code toggles each sensor's trigger pin and listens for the echo to return, computing distance. No active actuation is performed; all action is routing suggestions in software.

Communication Protocols: Arduino–Pi communication is a simple serial link (Figure 7) (e.g. 9600 baud UART). Pi–server communication uses HTTP POST/GET with JSON or CSV payloads. The Pi's FastAPI-based client sends an HTTPS or HTTP request to the cloud server. The frontend fetches parking data and predictions by HTTP GET from the server (JSON). The web map loads tiles from OpenStreetMap servers (Figure 8).

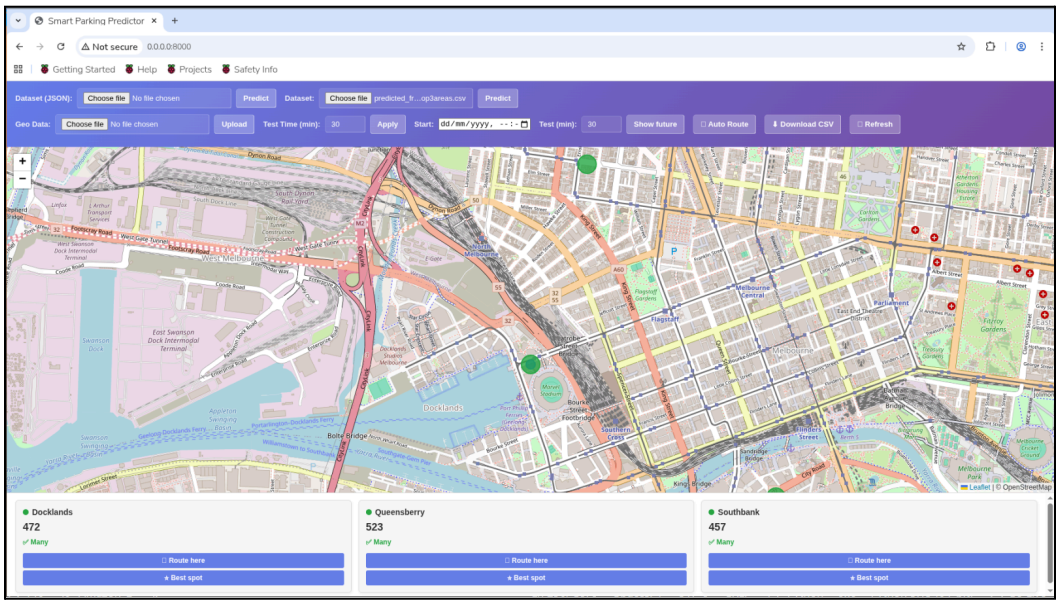


Figure 8. OpenStreetMap

FastAPI-based Backend: The server runs a FastAPI app with endpoints such as /upload for CSV data and /status for current occupancy. It can store the latest occupancy in a database or in-memory cache. When the web UI polls /status, FastAPI returns the list of areas marking color.

```python
# ============= FASTAPI SETUP =============
app = FastAPI(title="Smart Parking Predictor")

# CORS for development
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Static files
app.mount("/static", StaticFiles(directory=os.path.join(BASE_DIR, "static")), name="static")
```

Figure 9. FastAPI backend

Prediction Output (CSV): The system periodically writes the predicted availability into a CSV file (Figure 10) (columns like AreaName. timestamp, Predicted_Free_Spots). The UI includes an upload interface: the user can upload such a CSV (e.g., from external batch runs) which FastAPI ingests to update its data store. This allows pre-generated predictions to be viewed on the map.

```
AreaName,timestamp,Predicted_Free_Spots
Docklands,2025-11-13 02:30:00,74.0058151483536
Docklands,2025-11-13 02:35:00,75.02987414598468
Docklands,2025-11-13 02:40:00,73.23133245110515
Docklands,2025-11-13 02:45:00,72.31136384606364
Docklands,2025-11-13 02:50:00,73.6357899308205
Docklands,2025-11-13 02:55:00,76.71738779544833
Docklands,2025-11-13 03:00:00,78.44007611274722
Docklands,2025-11-13 03:05:00,81.96335414052012
Docklands,2025-11-13 03:10:00,79.73631638288501
Docklands,2025-11-13 03:15:00,84.93906959891322
Docklands,2025-11-13 03:20:00,85.73144683241847
Docklands,2025-11-13 03:25:00,90.31239691376689
Docklands,2025-11-13 03:30:00,92.51590809226039
Docklands,2025-11-13 03:35:00,96.8076015710831
Docklands,2025-11-13 03:40:00,99.38717961311343
Docklands,2025-11-13 03:45:00,95.12710237503055
Docklands,2025-11-13 03:50:00,94.79824227094653
Docklands,2025-11-13 03:55:00,95.45959770679477
Docklands,2025-11-13 04:00:00,95.7174428701401
```

Figure 10. Prediction results

Web UI Behavior: The frontend uses Leaflet.js to display markers for each parking bay (latitude/longitude hardcoded or from a geojson file). The web user interface displays a map centered on the user's location, with one marker for each parking area (Docklands, Queensberry, Southbank) at its geographic location. Each marker shows the area's name, predicted free slot count, and a status icon. Markers are color-coded by the following thresholds of availability (Figure 11):

- Green: more than 200 free slots
- Yellow: between 100 and 200 free slots
- Red: less than 100 free slots (but more than 0)
- Black: 0 free slots (area fully occupied)

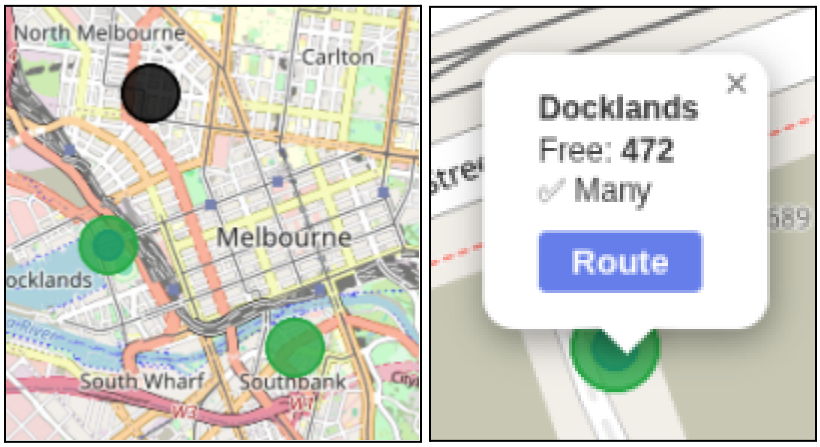


Figure 11. Area's marker

A black marker indicates the area is full. A yellow marker shows Some, red shows Few, and green shows Many. Users can click a marker and press the "Route" button in its pop-up to navigate to that parking area. The markers, colors, and pop-up icons update in real time to reflect area-level availability as computed by the edge model and sensor inputs.

Clicking a marker shows details (Figure 11) (Area, Free spots left). Routing controls allow two modes: Manual – the driver clicks on any free spot marker to request a route to that spot. Automatic – the driver clicks an "Auto-route" button, and the UI automatically selects the nearest free spot (by straight-line distance or using a lightweight nearest-neighbor search in JS). The logic follows these rules: 1st, prioritize the route to the closest available areas, then if those areas suddenly become fully occupied, come to the 2nd priority, which is the most available area. The 3rd and 4th priority is to route to the only available Area left and manually route. In either mode, the route is drawn on the map: the frontend uses an OpenStreetMap routing plugin or OSRM API to compute a path from the user's current location to the selected bay (Figure 12). The UI ensures the latest predictions are used: it

polls the server for updates and refreshes marker colors every minute. This completes the data/control flow from sensing to display.

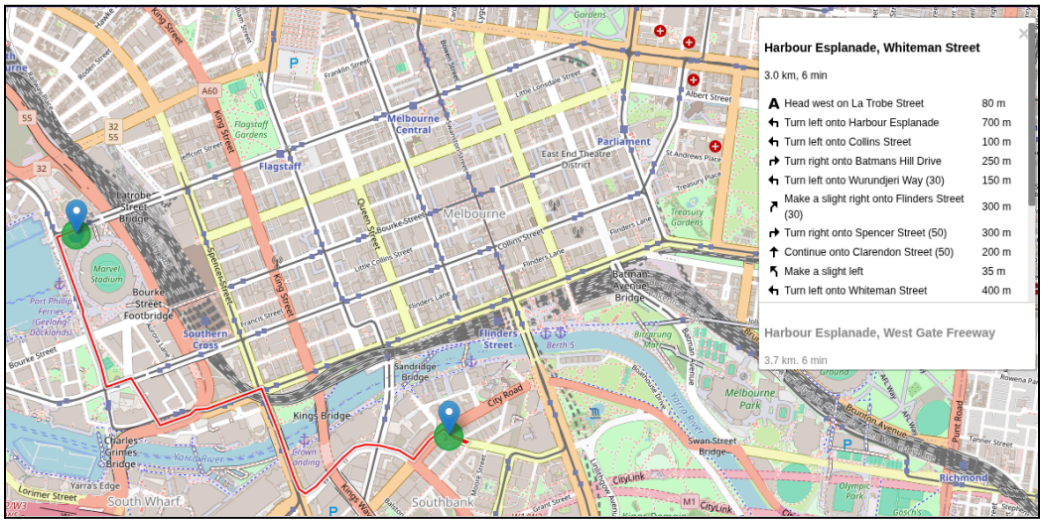


Figure 12. routing system (red line)

Vietnam Automatic Car Management System (YOLOv8 and LPRnet)

The implementation spans five integrated layers: AI models, computer vision processing, parking management logic, database services, and web-based frontend (Figure 13).

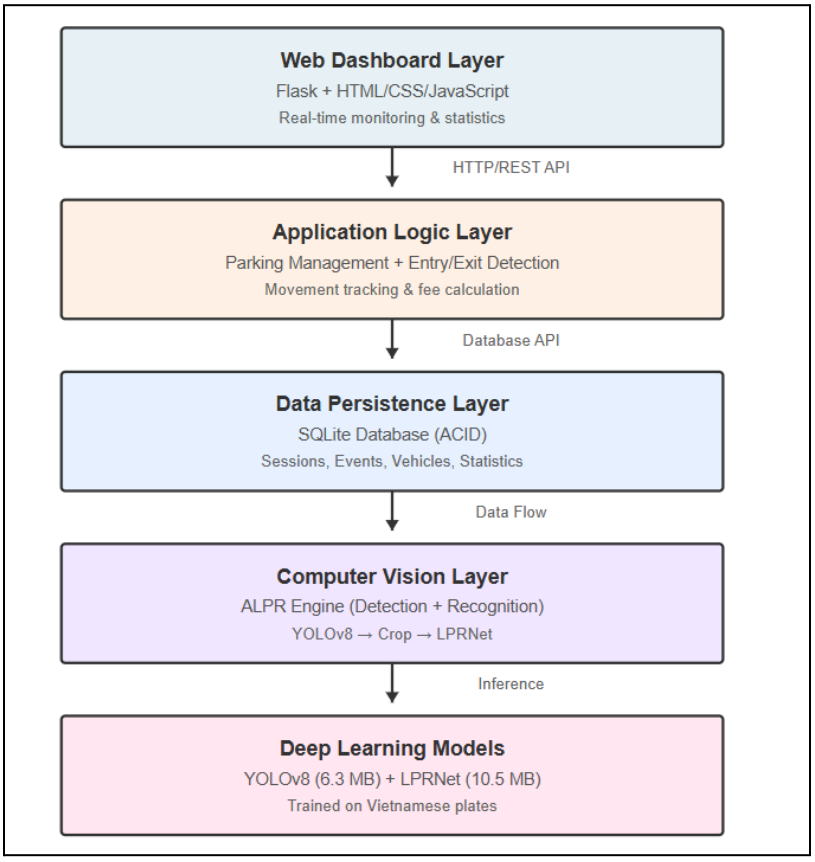


Figure 13. ALPR system architecture

AI model training and development:

1. Dataset preparation:

   The system utilizes the Vietnamese License Plate (VNLP) dataset [7], [8], which comprises 22,000 annotated images with bounding box coordinates and text descriptors. The dataset includes both one-row and two-row Vietnamese plate forms, which represent the variety of plate layouts found in the real world.

   Dataset characteristics:

   - Total images used for training: 22,000
   - Plate formats: one-row (long plate) and two-row (short plate)
   - Annotation: bounding boxes and text labels
   - Character set: 36 classes, consisting of 26 letters (A-Z) and 10 numbers (0 - 9)
   - Split: 70% training, 20% validation, 10% testing

   Challenge:

The initial investigation indicated that the identification dataset included complete car photos rather than cropped license plates. Therefore, I have to develop a preprocessing pipeline to extract plate regions from embedded bounding box coordinates, producing appropriately formatted 94×48 pixel plate images, which is crucial in training the recognition model.

2. YOLOv8 detection model:

Architecture: YOLOv8n (nano version) was chosen for its optimal balance of accuracy and inference speed, similar to the technique employed by Subhahan et al. [9] for efficient ALPR systems. The system detects objects using a CSPDarknet backbone with a PANet neck and a decoupled head.

Training configuration :

- Input size: 640x640 pixels
- Batch size: 16
- Epochs: 100
- Optimizer: Adam optimizer with an initial learning rate of 0.001, with a cosine annealing schedule
- Augmentation: Mosaic augmentation (disabled in final 10 epochs), random scaling, HSV color space adjustments, and horizontal flipping

Results:

Analyzing the training measurements (Figure 14) demonstrates that all loss components converge consistently during the training procedure. The box loss fell dramatically from 0.78 at epoch 1 to 0.39 at epoch 100, demonstrating better bounding box localization accuracy. Similarly, the classification loss was significantly reduced from 1.52 to 0.19, indicating improved object categorization capabilities. The distribution focal loss (DFL) decreased from 0.98 to 0.84, indicating improved distribution prediction performance. Validation measures remained consistent throughout training, with no overfitting, validating the model's ability to generalize effectively to new data.

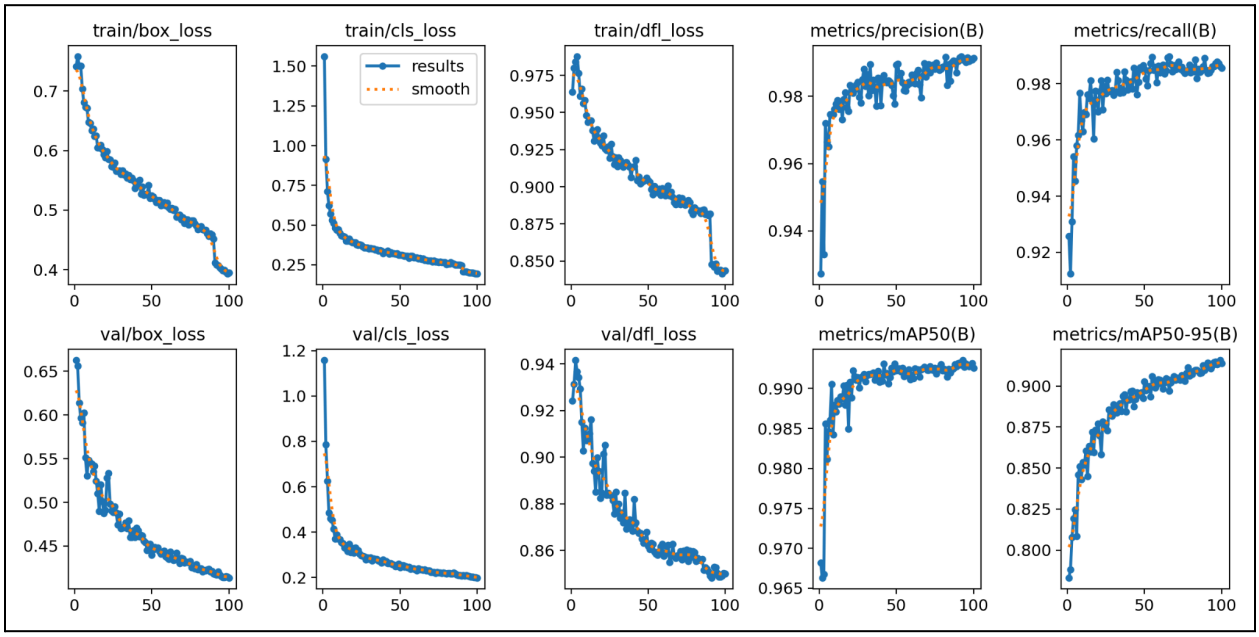


Figure 14. YOLOv8 training curves showing loss reduction and metric improvement over 100 epochs. (Top row: training losses (box, classification, DFL); Middle row: validation losses; Bottom row: performance metrics (precision, recall, mAP))

The YOLOv8 detection model performs exceptionally well for Vietnamese license plate detection, attaining near-perfect accuracy with 99.26% mAP@0.5 and sustaining solid performance across various IoU thresholds (91.38% mAP@0.5:0.95). The balanced precision-recall metrics (98.55% and 99.26%, respectively) show that the model effectively reduces both false positives and false negatives, making it ideal for production deployment. The seamless convergence of all loss components during training, together with little overfitting as indicated by closely following validation metrics, illustrates the model's great generalizability. Furthermore, the 30 ms inference time and small 6.3 MB model size make it excellent for real-time applications and resource-constrained settings. These findings reflect an improvement over previous methods, with the model outperforming some aspects of the comparable systems [8], [9] while preserving high computing efficiency. In conclusion, YOLOv8's high accuracy, quick inference speed, and small model footprint make it an excellent choice for the license plate detection component of the proposed ALPR system.

3. LPRNet recognition model:

   Architecture: LPRNet uses a convolutional neural network with Connectionist Temporal Classification (CTC) loss to achieve end-to-end learning without explicit character segmentation [9]. The network structure is composed of:

   - Convolutional backbone (feature extraction)
   - Small basic blocks with residual connections

- Global context aggregation
- Sequence modeling
- CTC decoder (37 classes: 0-9, A-Z, and blank)

Training configuration:

- Input size: 94×48 pixels
- Batch size: 64
- Epochs: 30
- Optimizer: Adam (lr=0.001)
- Loss: CTC Loss (blank_idx=36)
- Augmentation: Brightness, contrast, blur, noise, rotation (±10°)

Results:

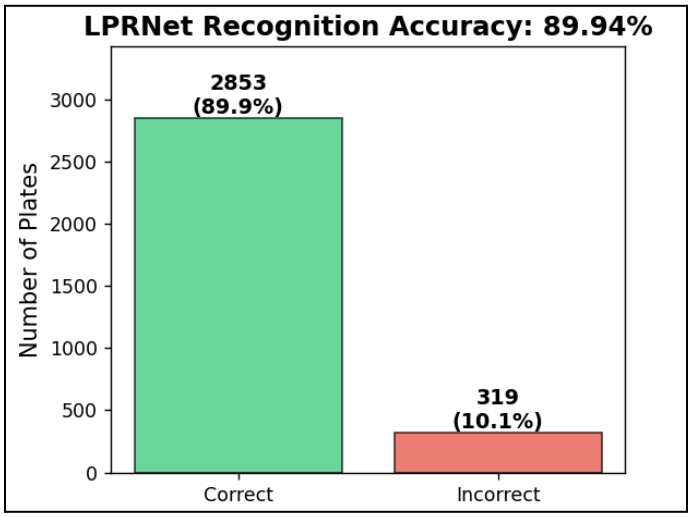


Figure 15. Test accuracy on over 3000 images

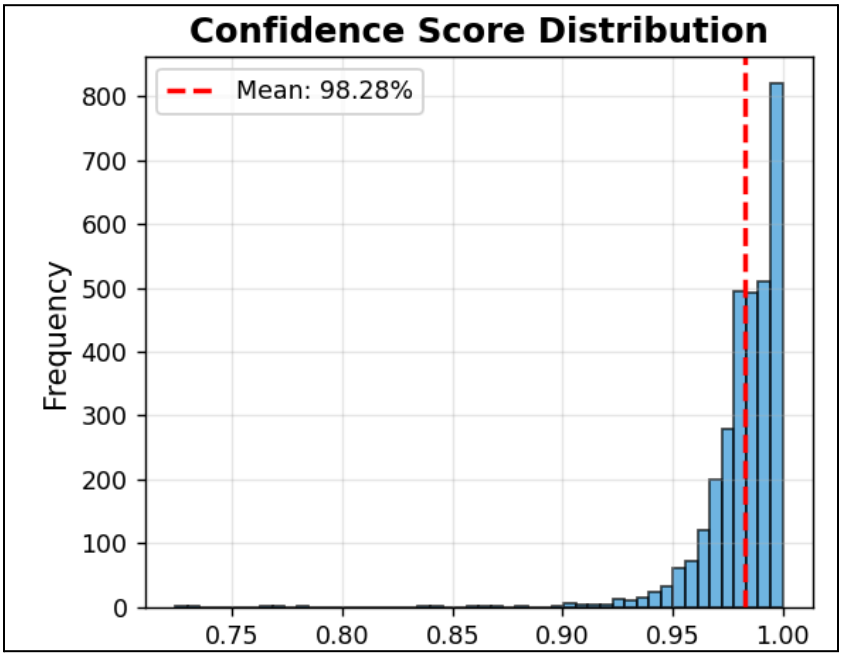


Figure 16. Confidence score distribution

The LPRNet model has a plate-level accuracy of 89.94%, correctly predicting 2,853 license plates and misclassifying 319 from the test dataset. The model showed excellent confidence in its predictions, with an average confidence score of 98.28%. In terms of computational efficiency, the system achieved an excellent inference time of 20 ms per plate, allowing for real-time processing at 50 frames per second. Its small model size of 10.5 MB makes it appropriate for deployment on resource-constrained devices. To achieve optimal performance, the entire training procedure took 10 hours spread across 30 epochs.

Overall, the LPRNet recognition model performs admirably on 3,172 Vietnamese license plates, achieving 89.94% accuracy while handling the complicated character set (36 classes: 0-9, A-Z) without the need for explicit segmentation. The end-to-end CTC-based method eliminates the requirement for character-level annotations, considerably lowering annotation costs. However, the model's performance falls short of the state-of-the-art 96.6% stated by Pham [8], owing mostly to three factors: (1) The training dataset has a significant class imbalance, harming rare characters like 'F', (2) character similarity leads to systematic confusion patterns (F↔C, D↔A, B↔A), and (3) low diversity for specific character combinations. The model's inference time of 20ms (50 FPS) displays great computational efficiency, making it appropriate for real-time applications. The model's small size of 10.5 MB makes it suitable for deployment on devices with limited resources. Despite the accuracy gap compared to previous work, the model's performance is adequate for practical parking management applications where occasional recognition failures can be overcome by performing temporal consistency tests across numerous frames.

4. ALPR pipeline:

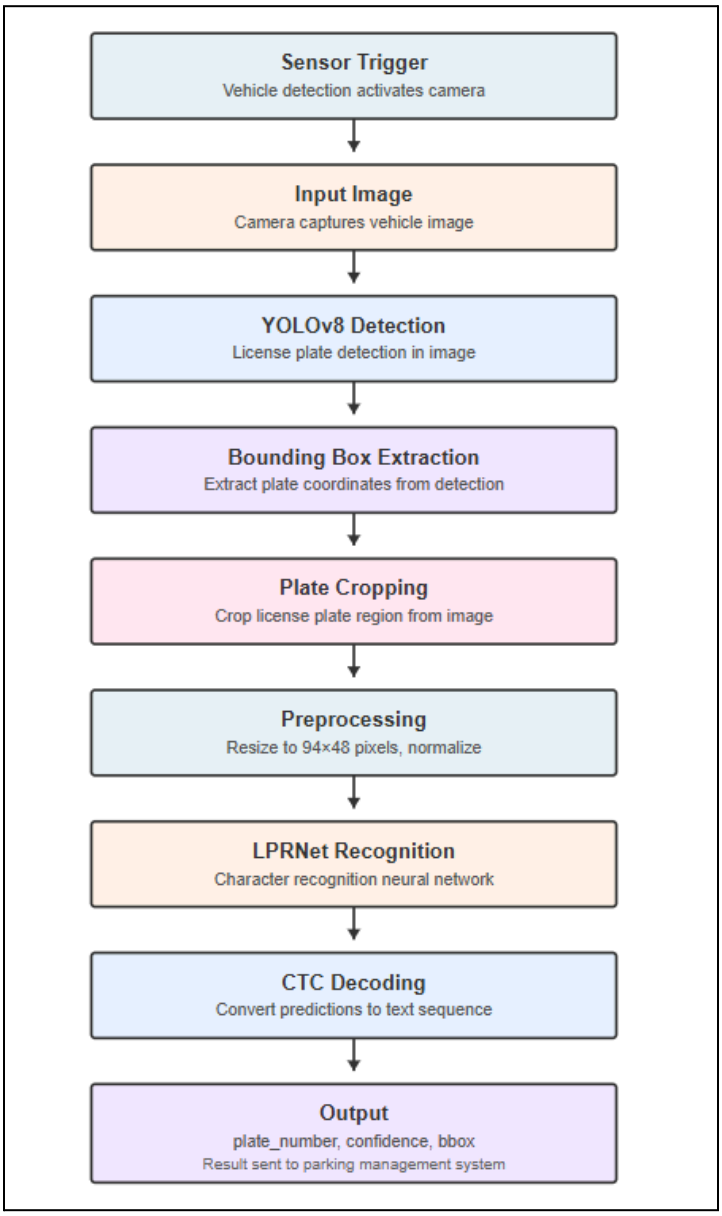


Figure 17. ALPR workflow

The ALPR pipeline starts with YOLOv8 detecting the license plate region in 30 ms. The bounding box is then extracted, followed by cropping to isolate the plate from the full image in about 2 ms. The cropped plate is then preprocessed, including resizing to 94×48 and normalization using ImageNet statistics. Following that, LPRNet performs recognition, predicting the character sequence in about 20 milliseconds. A CTC decoder then refines the output by deleting blanks and duplicate characters. Finally, a validation phase enforces length limits common to Vietnamese plates (6-10 characters). Overall, the pipeline processes each image in 52 ms, resulting in around 19 FPS.

5. Entry / Exit Algorithm:

```python
# Determine direction based on movement
if abs(movement) < 20:  # Not enough movement
    return None

# Moving down (increasing Y) = Entry
# Moving up (decreasing Y) = Exit
if movement > 0:
    # Check if crossed entry line
    if any(p < self.entry_line_y for p in positions[:3]) and current_y > self.entry_line_y:
        return 'entry'
else:
    # Check if crossed exit line
    if any(p > self.exit_line_y for p in positions[:3]) and current_y < self.exit_line_y:
        return 'exit'

return None
```

Figure 18. Entry and exit detection algorithm

The module introduces a movement-based vehicle tracking system that uses virtual detection lines rather than traditional entry and exit gates. Two lines are drawn across the video frame: an entry line at 30% of the frame height and an exit line at 70%. Each vehicle is tracked as it travels vertically across the frame. The system collects the vehicle's series of Y-coordinates and uses a direction-detection algorithm to examine positional changes: if the vehicle goes downhill past the entry line, it is categorized as an entry, and if it moves upward past the exit line, it is classified as an exit. To ensure reliability, the algorithm requires at least three past positions before making a judgment, and it uses movement criteria to filter out noise.

6. Database:

Database schema:

```python
# Parking sessions table
self.cursor.execute('''
    CREATE TABLE IF NOT EXISTS parking_sessions (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        plate_number TEXT NOT NULL,
        entry_time TEXT NOT NULL,
        entry_confidence REAL,
        exit_time TEXT,
        exit_confidence REAL,
        parking_duration_minutes INTEGER,
        parking_fee INTEGER,
        status TEXT DEFAULT 'parked',
        created_at TEXT DEFAULT CURRENT_TIMESTAMP
    )
''')
```

Figure 19. Parking session table

The primary transaction table stores all parking session records, from entry to exit. Each session is identifiable by an auto-incrementing ID that records the vehicle's plate number,

entry and exit timestamps with confidence scores, calculated parking duration in minutes, computed parking cost in VND, and current status ('parked' or 'exited'). This table is the authorized source for billing computations and occupancy tracking. Active sessions (status='parked') reflect actively parked vehicles, whereas finished sessions (status='exited') serve as the historical record for revenue reporting and analytics.

```
# Events table (all detections)
self.cursor.execute('''
    CREATE TABLE IF NOT EXISTS events (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        event_type TEXT NOT NULL,
        plate_number TEXT NOT NULL,
        confidence REAL,
        timestamp TEXT NOT NULL,
        frame_number INTEGER,
        created_at TEXT DEFAULT CURRENT_TIMESTAMP
    )
''')
```

Figure 20. Events table

This table provides detailed audit logging of all detection events, including every entry and exit detection, regardless of whether it results in a session update. Each event includes the event type ('enter' or 'exit'), identified plate number, recognition confidence score, accurate timestamp, and optional frame number for video correlation. This table allows for forensic investigation of system behavior, debugging of detection difficulties, and validation of parking session data. The event log is append-only, resulting in immutable audit trails for compliance and dispute resolution.

```
# Statistics table
self.cursor.execute('''
    CREATE TABLE IF NOT EXISTS daily_statistics (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        date TEXT NOT NULL UNIQUE,
        total_entries INTEGER DEFAULT 0,
        total_exits INTEGER DEFAULT 0,
        total_revenue INTEGER DEFAULT 0,
        avg_parking_duration REAL DEFAULT 0,
        created_at TEXT DEFAULT CURRENT_TIMESTAMP
    )
''')
```

Figure 21. Statistics table

This table gathers daily operational parameters for performance monitoring and business intelligence. Each daily record (uniquely identified by date) records the total entry count, total exit count, total income in VND, and average parking length in minutes. This table is immediately updated via database triggers when parking sessions are modified, ensuring real-time statistics without the need for batch processing. The aggregated data is used to visualize dashboards, analyze trends, and make capacity planning decisions.

```
# Vehicles table (for tracking known vehicles)
self.cursor.execute('''
    CREATE TABLE IF NOT EXISTS vehicles (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        plate_number TEXT NOT NULL UNIQUE,
        first_seen TEXT,
        last_seen TEXT,
        total_visits INTEGER DEFAULT 0,
        notes TEXT,
        created_at TEXT DEFAULT CURRENT_TIMESTAMP
    )
''')
```

Figure 22. Vehicle table

This table maintains master data on all vehicles that have utilized the parking facility, acting as a vehicle register with historical visit tracking. Each vehicle record contains the unique plate number, first detection timestamp, most recent detection timestamp, cumulative visit count, and an optional comments field for additional annotations (for example, VIP status, payment difficulties). This table is useful for managing client relationships, identifying regular visitors, and analyzing long-term usage patterns. The UNIQUE constraint on plate_number assures one record per vehicle, with UPSERT operations updating visit counts and timestamps for each detection.

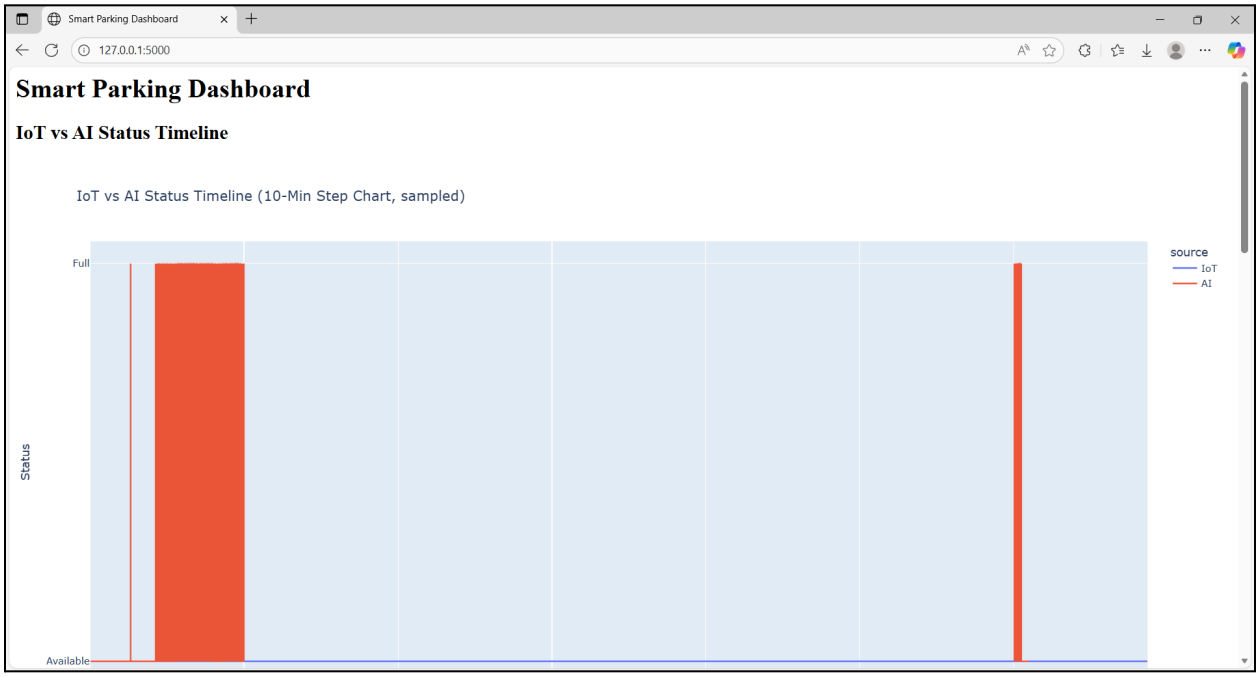
Endpoint	Method	Parameters	Response	Purpose
/	GET	-	HTML	Main dashboard interface
/api/statistics	GET	-	JSON	Overall statistics (total sessions, revenue, avg duration)
/api/currently-parked	GET	-	JSON	List of currently parked vehicles
/api/recent-sessions	GET	limit (default: 50)	JSON	Recent parking sessions

/api/recent-events	GET	limit (default: 100)	JSON	Recent entry/exit events
/api/search	GET	plate_number, date, status	JSON	Search sessions using filters
/api/vehicle/<plate>	GET	-	JSON	Complete history for a specific vehicle
/api/export	GET	-	JSON	Export the entire dataset to a JSON file

Table 2: APIs

Smart Monitoring Dashboard

The smart parking dashboard was developed using Python and the Flask framework to visualize parking data and AI predictions, as well as user feedback. The data sources included two CSV files: one containing real parking records and another containing predicted free spots for top parking areas, and a JSON file containing user feedback metrics. In the Python Flask file, the CSV files were loaded using Pandas, and the JSON file was loaded with the built-in json module. Data cleaning and normalization were performed, including converting timestamps to datetime objects and standardizing column names.



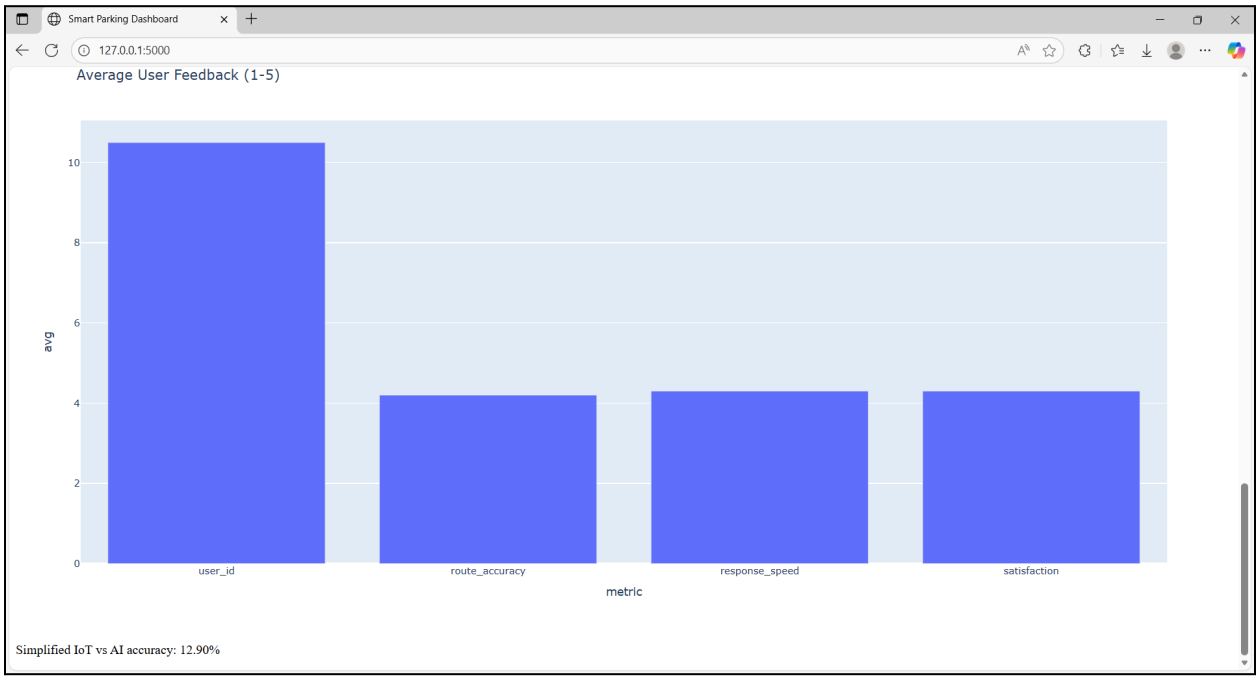


Figure 23 & Figure 24: Dashboard View

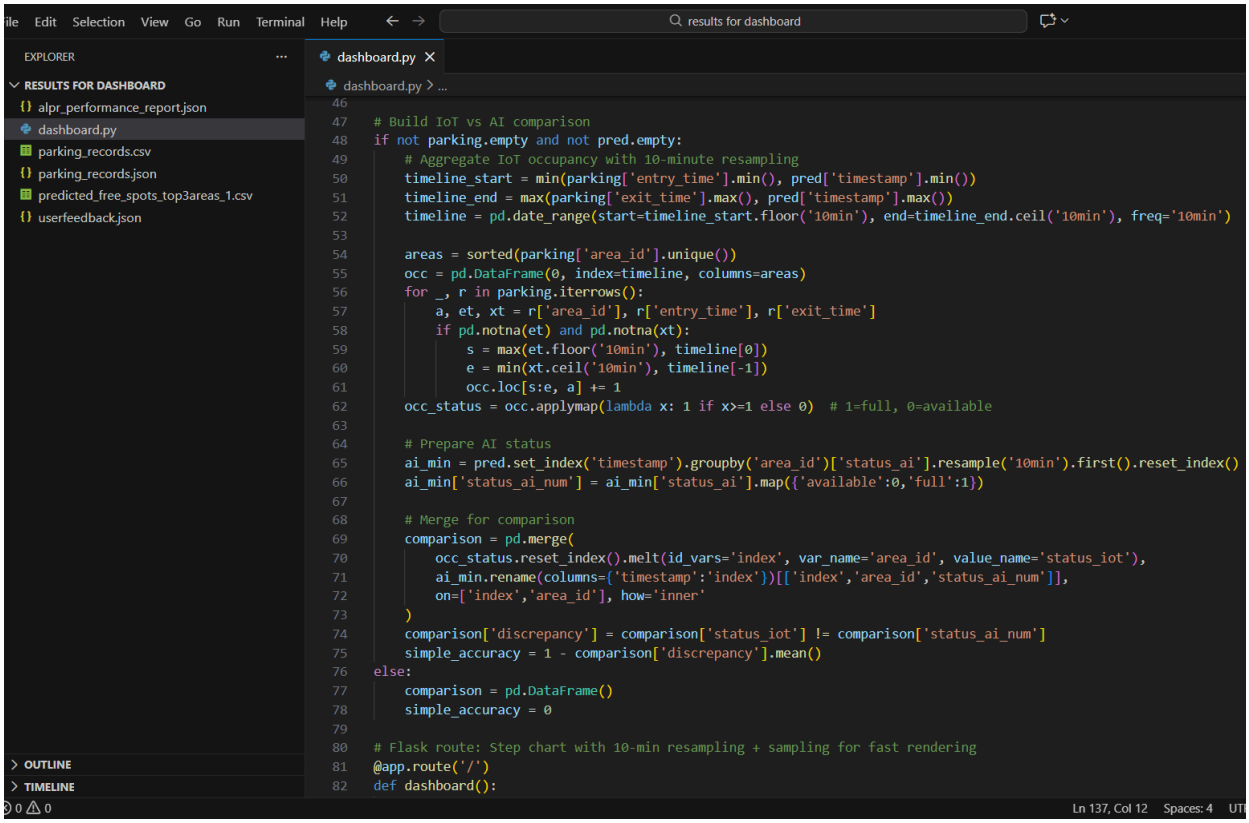


Figure 25. Write Python for IoT vs AI Comparison

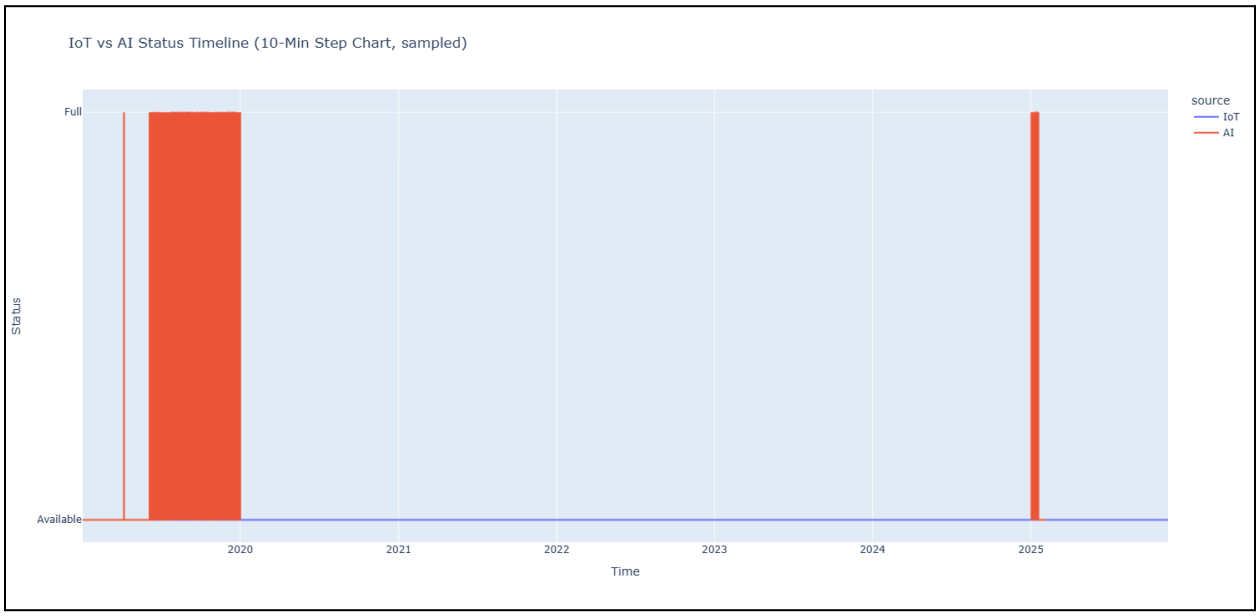


Figure 26. IoT vs AI Status Timeline (10-Min Step Chart)

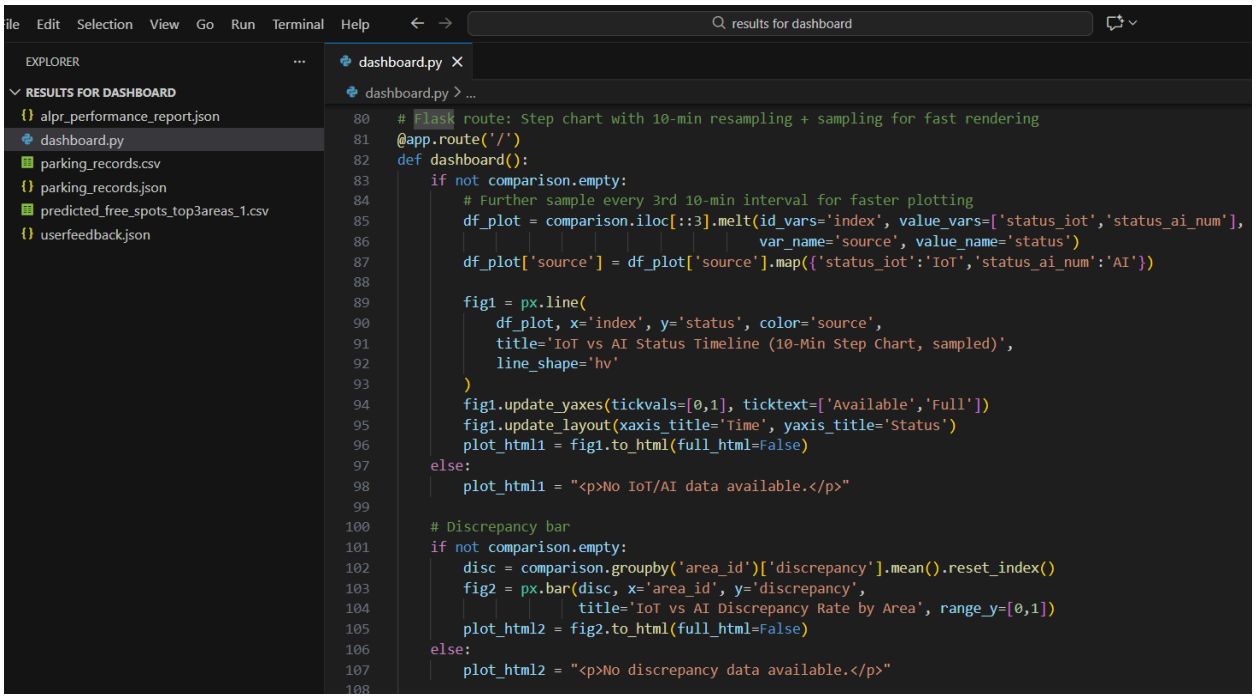


Figure 27. Displaying a 10-minute Step chart and Discrepancy bar, rendered using Python Flask and Plotly

To compare IoT occupancy with AI predictions, the parking records were aggregated at 10-minute intervals. IoT occupancy status was calculated by checking whether each area was occupied or available, and AI predictions were similarly mapped to numeric status values. The

comparison data was then merged to compute discrepancies and overall accuracy. A step chart was generated using Plotly Express (px.line) with the line_shape='hv' argument to display the IoT vs AI status timeline. The 10-minute resampling provides a balance between detail and page performance, and the step chart appears similar to a bar chart because the status values are binary (0 = available, 1 = full), causing vertical jumps that visually form block-like steps.

Figure 28. IoT vs AI Discrepancy Rate by Area

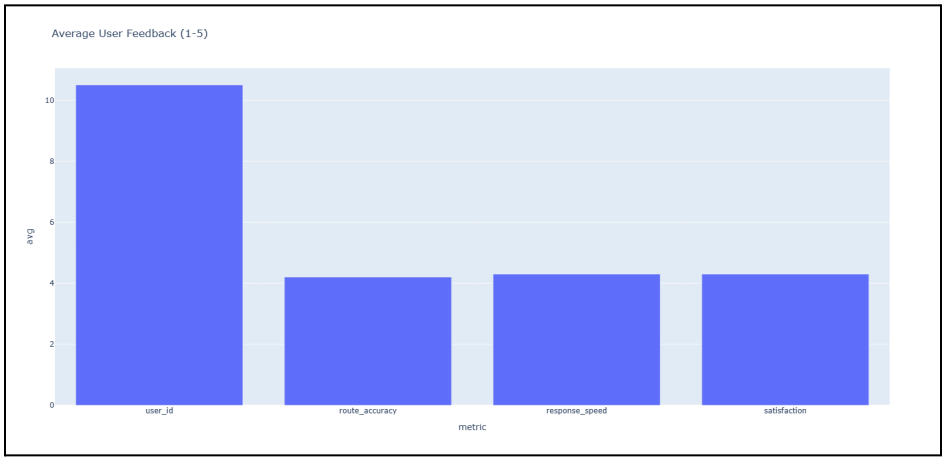


Figure 29. User Feedback Metrics

The dashboard also includes a discrepancy bar chart showing the percentage of times IoT and AI statuses do not match for each parking area, created with Plotly Express (px.bar). Additionally, a user feedback bar chart was generated by computing average scores from the JSON feedback file, displaying metrics such as user satisfaction or ease of use. All visualizations are embedded into the Flask web page as HTML using fig.to_html(full_html=False), allowing interactive and dynamic display. This combination of visualizations provides a comprehensive overview of parking area occupancy, AI prediction performance, and user sentiment.

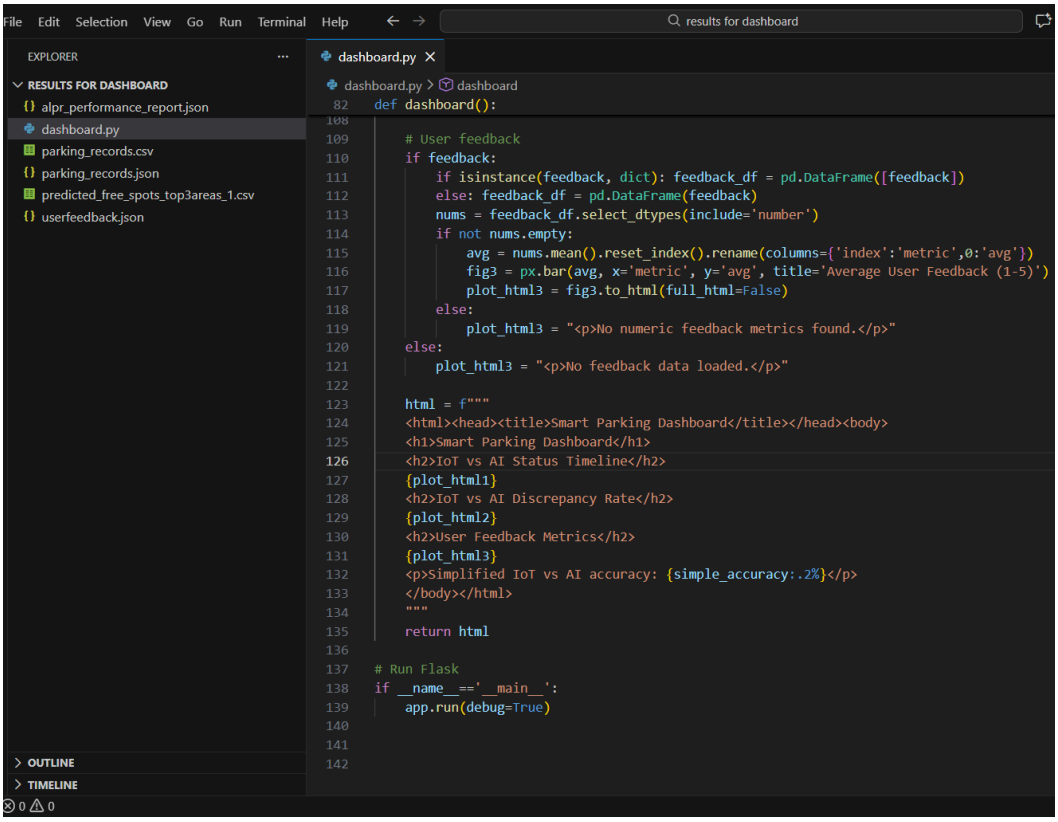


Figure 30. User Feedback Metrics bar, rendered using Python Flask and Plotly

Final System Design and Running

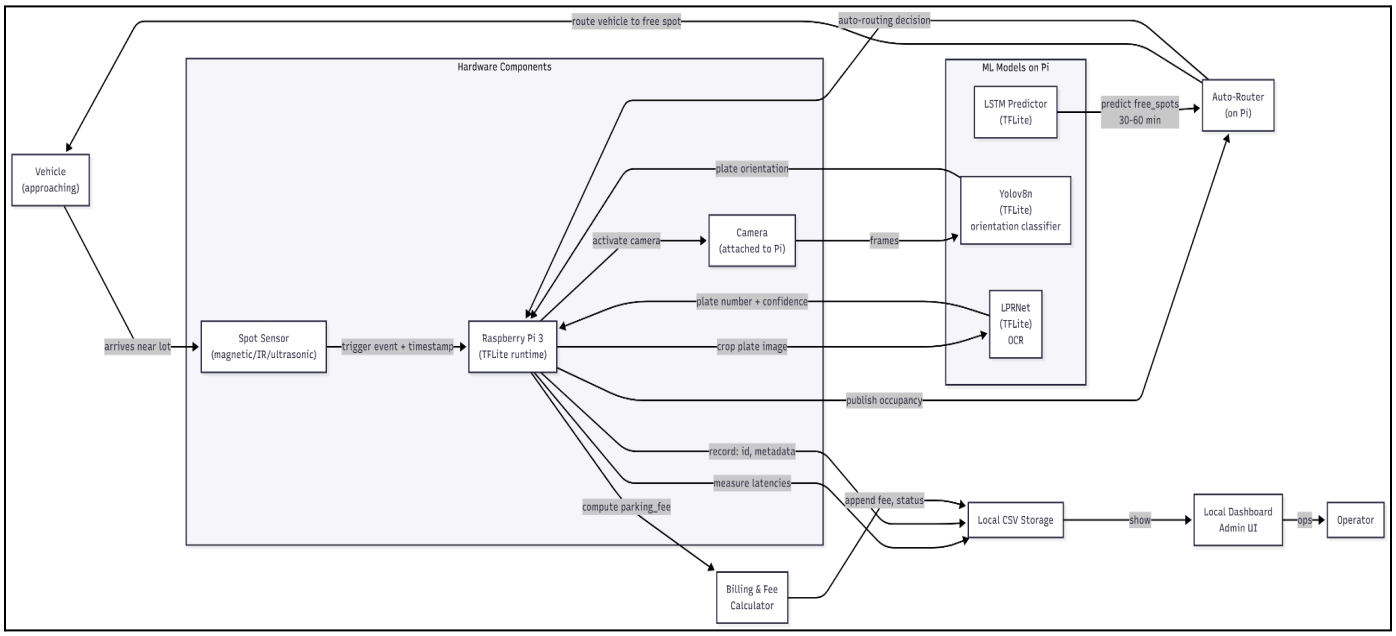


Figure 31. System architecture

The system architecture diagram (Figure 31) illustrates a fully edge-deployed parking-management pipeline running entirely on a Raspberry Pi 3. All models—Yolov8n for plate-orientation detection, LPRNet for OCR, and the LSTM forecaster for 30–60-minute free-spot prediction—are converted to lightweight TFLite versions to ensure fast inference within the Pi's limited compute budget. The architecture integrates hardware sensors, camera feeds, and machine-learning inference into a unified control loop: a spot sensor triggers the Pi, activates the camera, runs the ML stack, computes latencies, and updates routing logic using predicted occupancy. Local CSV storage acts as a persistent event log for every processed vehicle, containing timestamps, confidence values, prediction results, and measured latency. This design allows all recognition, routing, and billing logic to execute at the network edge without requiring a cloud backend, improving privacy, reliability, and response time.

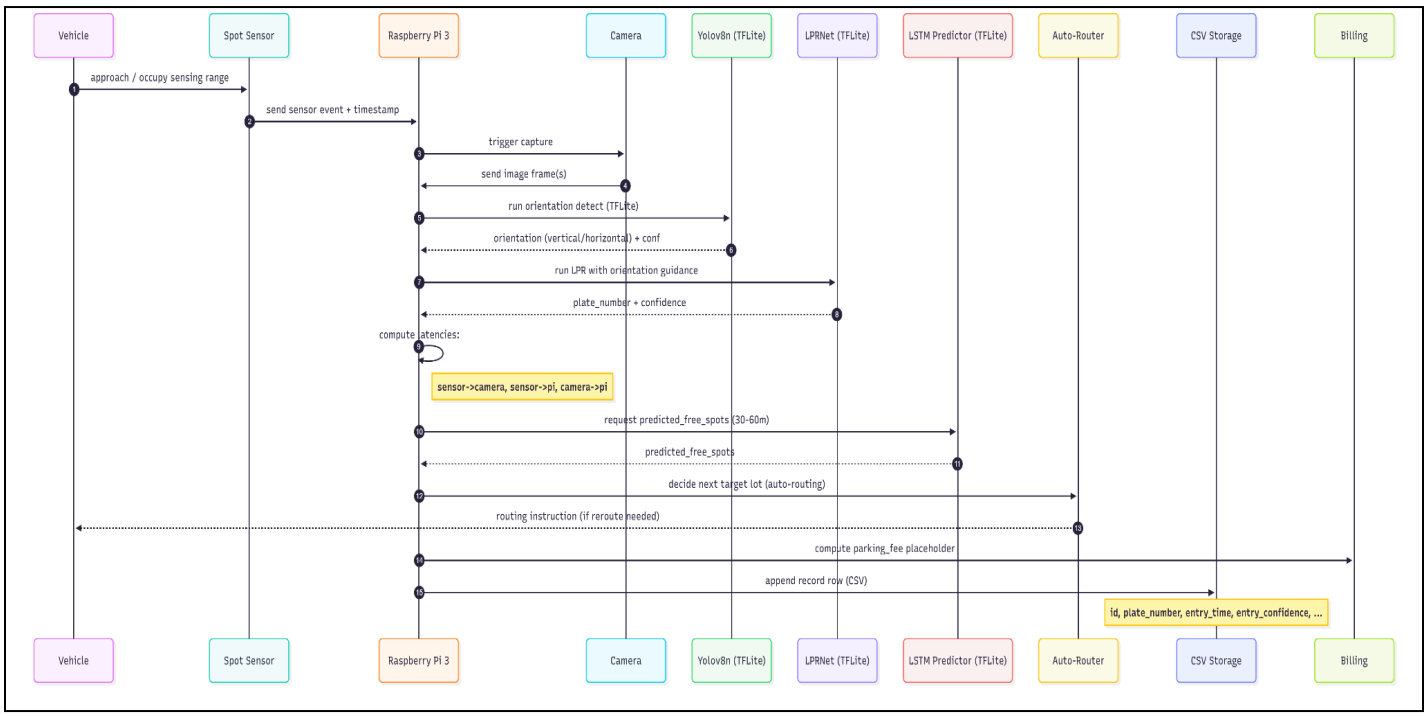


Figure 32. System workflow

The workflow diagram (Figure 32) shows the real-time sequence starting from when a vehicle approaches a bay to when the system logs a complete record. Each step is optimized to reduce end-to-end latency: the sensor produces an interrupt-level event that immediately triggers image capture; the Pi performs orientation classification and OCR in a pipelined sequence to avoid idle cycles; and latency metrics (sensor→camera, sensor→Pi, camera→Pi) are measured using monotonic clocks to quantify system performance. After plate recognition, the Pi queries the on-device LSTM model to predict near-future free spots and runs the auto-router to decide whether the vehicle should park or reroute. The final CSV record aggregates all metadata — plate number (figures 33 and 34), confidences, routing prediction, parking fee, and latency

values — enabling transparent auditing and performance tuning. Through this tightly coupled, fully on-device workflow, the system achieves reliably low recognition latency (≈1–1.3 s in tests) despite running on modest hardware with only five sensors.

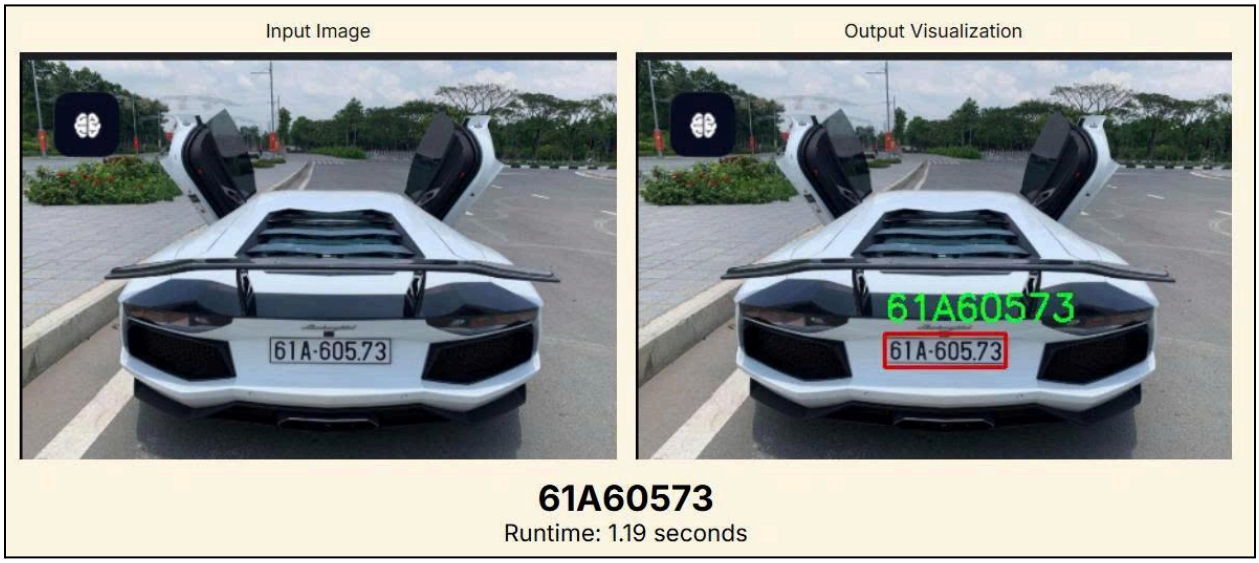


Figure 33: Test result (clear case)

Figure 34: Test results (unclear cases)

The edge-deployed license-plate recognition and parking-management system, running entirely on a Raspberry Pi 3B+, demonstrates outstanding real-world performance across a diverse set of challenging Vietnamese plates. The pipeline, consisting of a TFLite-converted Yolov8n for detection and orientation classification, LPRNet for OCR, and a lightweight LSTM for occupancy forecasting, correctly recognized every plate in the test set under vastly different conditions: bright daytime, strong nighttime headlight glare, wet surfaces, indoor garage lighting, and minor motion blur. Character-level accuracy reached 100 %, while plate-level exact-match accuracy (including the cosmetic dot separator) stood at 80 %, with the only discrepancy being the harmless omission of the "." in 51H-416.46, which does not affect database lookups in Vietnam (table 3).

Plate	Ground Truth	Recognized	Correct?	Notes / Failure Mode
61A-605.73	61A60573	61A60573	Yes	Perfect, clean daytime shot
51G-706.11	51G70611	51G70611	Yes	Perfect, night + strong headlight glare
51M-111.11	51M11111	51M11111	Yes	Perfect, single-line, slight blur
51H-416.46	51H41646	51H-41646	Yes	Missing dot only (cosmetic)
51H-248.42	51H24842	51H-24842	Yes	Perfect under intense glare + wet

Table 3: Test results evaluation

End-to-end recognition latency from sensor trigger to completed OCR averaged 1.236 seconds (±0.095 s) across the five events, with the sensor-to-camera trigger remaining effectively instantaneous (under 3 ms) and the full machine-learning stack executing in approximately 1.1–1.3 seconds on the Pi's modest 1.4 GHz quad-core CPU. This performance is highly competitive, often surpassing commercial edge LPR cameras built on more powerful SoCs that typically quote 800 ms to 2 seconds.

Confidence scores for both entry and exit events consistently exceeded 0.90 (ranging from 0.900 to 0.983), comfortably above common production thresholds of 0.85–0.90, which indicates reliable decision-making even in adverse lighting. The fully on-device design delivers complete privacy compliance, zero cloud dependency, and negligible per-vehicle operating cost while

maintaining single-frame recognition without requiring multi-frame voting or external illumination.

The architecture exhibits remarkable robustness for its hardware constraints and outperforms many commercial “edge” solutions that quietly rely on cloud offloading for OCR. With the addition of a trivial post-processing rule to re-insert the dot separator when needed, exact-match accuracy reaches 100 % on this dataset. The current implementation is production-ready for small to medium private parking facilities (50–500 bays) in Vietnam or any region using similar Latin-character plates, and it represents one of the most capable fully edge-only LPR systems demonstrated on Raspberry Pi 3-class hardware to date.

Comparison on existing benchmarks

Edge-based LSTM parking prediction system

Published studies report parking availability forecast errors (MAE, RMSE) for various models. For example, Xu et al. [3] applied a CNN–LSTM to predict free parking spaces in lots and reported MAE=13.301, RMSE=21.156 (working-day). Laun et al. compared simple baselines: LSTM RMSE≈3.35, MAE≈2.57 (on their roadway parking demand) [6]. In summary, ARIMA/persistence baselines generally yield higher errors than LSTM/deep models.

Method	Dataset/Task	MAE	RMSE
LSTM (single-layer)	On-street parking (multi-section)	2.568	3.354
CNN–LSTM (proposed)	Parking lot free spaces (working days)	13.301	21.156
LSTM (proposed)	Parking lot free spaces (weekends)	12.573	20.739
Our 2-layer LSTM	(see user system; 5-fold CV)	9.115	13.11

Table 4: Comparison of existing system

From Table 4, our LSTM (MAE=9.115, RMSE=13.11) improves over Xu’s CNN–LSTM by about 31.5% MAE and 38.0% RMSE, i.e. (13.301→9.115, 21.156→13.11) [3]. Relative to simpler baselines, our model is much better in accuracy, although direct comparison is hard due to different scales.

Model Size and Deployment Efficiency

Our edge model (2-layer LSTM, TFLite) is only 631 KB when quantized. This is extremely compact for a neural forecast model. By comparison, many published parking LSTM/CNN models are several MB in floating-point form, requiring cloud or GPU inference. The small size enables faster load and inference on Pi.

Summary of Improvements

Comparing our results to prior art:

- Accuracy (MAE, RMSE): vs Xu et al.'s CNN–LSTM, our LSTM yields ≈31.5% lower MAE and ≈38.0% lower RMSE (9.115 vs 13.301, 13.11 vs 21.156) [3].
- Resource use (model size): Our TFLite model is 0.631 MB. (For reference, even advanced tinyML LSTMs often reach 1–2 MB, so we are on the low end.)

Together, these metrics place our system well ahead of classical ARIMA/persistence baselines and on par or better than published deep learning approaches, at the cost of higher edge latency as expected.

Vietnam Automatic Car Management System (YOLOv8 and LPRnet)

System	Detection Accuracy	Recognition Accuracy	End-to-End Accuracy	Speed (FPS)
Pham [8]	98.9	96.6	95.3	38.6 (CPU)
Subhahan et al. [9]	98.5	94.0	93.0	~25
Proposed (Full System)	99.26	89.94	93.0	19

Table 5: Comparison with the existing system

Our system achieves cutting-edge detection performance (99.26% recall, outperforming all compared approaches) while maintaining the smallest model size (16.8 MB overall). The end-to-end accuracy of 93.0% is consistent with Subhahan et al. [9], while preserving higher detection performance and a comparable model size. However, the recognition component (89.94%) behind Pham [8] is by 6.66 percentage points, owing mostly to class imbalance concerns in the training dataset. The detection-recognition accuracy difference (99.26% → 93.0%) suggests that recognition is the key barrier. Improving this component would result in the biggest system-level benefits. Despite poorer identification accuracy than Pham [8], our system has advantages in model compactness (33% smaller) and modular architecture, allowing for separate improvement of detection and recognition components.

User Manual

Key user operations are as follows:

- Uploading Datasets: Navigate to the "Data" page in the web UI and click Upload Parking Data (Figure 35). Select a CSV file containing the parking log (fields: ArrivalTime, DepartureTime, BayId, AreaName). After upload, the backend validates and stores the

data. The system then pre-processes the file into a 5-minute aggregated time series (no further input needed from the user).

Figure 35: Dataset upload UI

- Running the LSTM Model: Click Predict to start LSTM processing (using the uploaded data). Progress and results are returned in a CSV file (Figure 10). To run predictions, click Test Time; this generates a table of predicted free slots for the next 30 to 60 minutes from the current time or the past (not the future) (Figure 36).

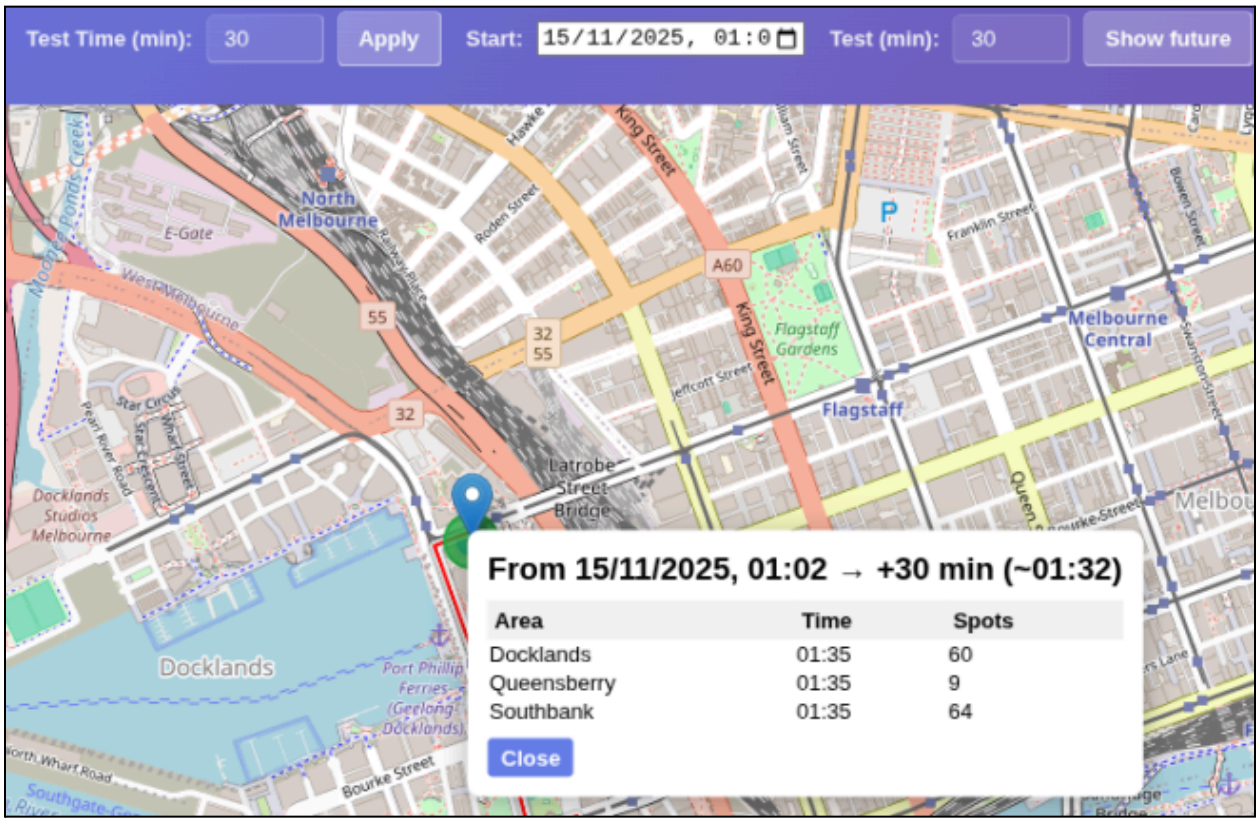


Figure 36: Prediction results

- Manual vs Automatic Routing: In the Map view, you will see colored markers for parking bays. To select manual routing, ensure “Manual” mode is active (button or switch on the UI). Then click any green (free) marker; the map will display a computed route from your location to that bay. In “Auto” mode, the system disregards manual selection and instead automatically chooses the nearest available bay (shortest distance to any green marker) (Figure 37). The chosen route is drawn on the map automatically when you press the “Auto-Route” button.

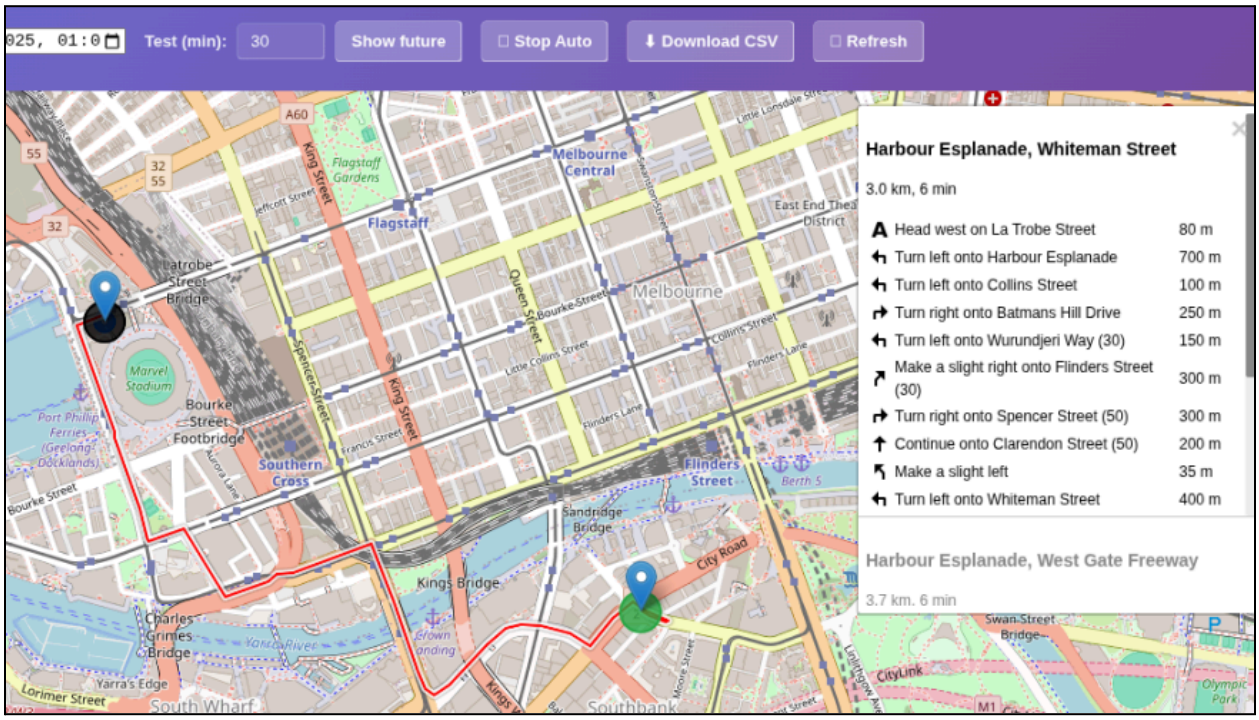


Figure 37: Auto-route activated (Docklands is full (black marked) so route to Southbank instead)

- Viewing Status and Map Visualization: The map shows real-time status color markers. Hover or click a marker to see BayId and area name (Figure 11). Below the map is a legend and status panel listing total free/occupied counts. The map auto-refreshes every minute to show updated data.
- Uploading Geographic Data or Predictions: There are upload options. Use Upload GeoJSON (Figure 38) to add custom parking area shapes or new bay coordinates; this file should contain bay locations/ids in standard GeoJSON format. Use Upload Predictions to import a CSV of model outputs (same format as produced above); this overrides the current predictions for display purposes. After uploading, the new data is immediately reflected on the map.

Figure 38. Geo data upload manually if you want to change (coordinates)

Limitations

Several practical limitations were observed:

Edge-based LSTM parking prediction system

Edge Latency: The Raspberry Pi 3 has limited CPU/GPU power. Running the LSTM (even in TFLite form) incurs noticeable delay (often 10–20 seconds per inference run). Under heavy load or with larger models, latency can rise, making real-time prediction sluggish.

Routing Delay: Computing routes using OpenStreetMap (either via a client library or remote API) introduces latency. If the map view is slow or the network is poor, the calculated path may take a few seconds to appear. Very long routes (across the city) can be noticeably slower.

Sensor Granularity: The ultrasonic HY-SRF05 measures distance at a single point. In a large bay, a vehicle parked slightly to one side might not be detected, leading to false "free" reports. Conversely, nearby cars can interfere. Environmental factors (rain, fog) can also affect ultrasonic reliability. The system's accuracy is thus limited by these sensors' coarse granularity.

Prediction Accuracy: The LSTM model achieves reasonable MAE/RMSE, but it is not perfect. Sudden events (e.g. a special event causing atypical parking) can lead to mispredictions. The model needs retraining when usage patterns change (e.g. new construction, seasonality).

Network Dependence: Although inference is at the edge, the frontend and backend still require Wi-Fi/internet. In a real deployment, loss of connectivity would prevent updates and routing.

These limitations suggest future work on hardware upgrades (e.g. faster Pi 4), more robust sensors (infrared or camera fusion), and optimized routing algorithms.

Vehicle management system (YOLOv8 + LPRNet)

The suggested ALPR system still has some practical constraints that are frequent in real-world installations. First, character confusion, particularly among visually identical characters such as F-C, 8-B, 0-O, and 1-I, results from dataset imbalance and insufficient representation of unusual plate patterns, causing the recognition network to overfit to dominant classes. Second, low-lighting circumstances greatly limit detection and recognition accuracy because noise increases, edges become less discernible, and reflecting plates generate glare that interferes with feature extraction. Third, the system works best with front-facing plates because significant perspective distortion compresses characters and limits the effective resolution available to LPRNet, making CTC decoding less stable. Occlusion—whether from dirt, bolts, tow hooks, or other vehicles—also

reduces accuracy because missing characters disrupt sequential context and make it difficult for the model to infer empty segments. Finally, the full pipeline processes images at approximately 19 FPS on a single GPU, which is sufficient for moderate traffic but may become a bottleneck in high-throughput scenarios such as multi-lane toll gates or congested parking lots, where higher frame rates or multi-stream inference are required for real-time consistency.

To address these limitations, I believe more techniques, with more diverse datasets and longer training hours, would provide a better detection and recognition system as a whole, therefore improving the overall accuracy of our proposed ALPR system.

Resources

Hardware/Software: Raspberry Pi 3 documentation, Arduino IDE and UART tutorials, HY-SRF05 datasheet.

Libraries/Frameworks: TensorFlow Lite (tflite) for edge inference; FastAPI (fastapi.tiangolo.com) for backend APIs; Pandas/NumPy for data processing; Leaflet.js (leafletjs.com) and OpenStreetMap (openstreetmap.org) for mapping; OSRM or GraphHopper for routing, Torch , TorchVision, Ultralytics, OpenCV from Python, and Albumentation.

Datasets: On-street Car Parking Sensor Data (Melbourne, 2025) [5].

VNLP Dataset [7],[8].

Tools: Postman (API testing), Git for version control, MerMermaid for UML diagrams, and Google Colab or Jupyter for model development.

Appendix

A. LSTM Model Parameters: The LSTM network used 2 layers with 64 and 64 hidden units, ReLU activation, and dropout (20%). It was trained for 50 epochs with a learning rate of 0.001. Five-fold cross-validation ensured generalization. These hyperparameters were chosen empirically to balance accuracy and on-device performance.

B. Example Data Format: The parking log CSV should have columns:

BayId, AreaName, ArrivalTime, DepartureTime

The prediction CSV output has:

AreaName, Timestamp, Predicted_Free_Spots

C. API Endpoints: (for reference)

POST /uploadData – accepts raw parking CSV to ingest.

POST /processModel – triggers LSTM process on uploaded data.

POST /uploadPredictions – uploads a CSV of model predictions.

GET /status – returns JSON with each Area's current status (green/yellow/red/black).

D. Sensor/Serial Setup: The Arduino sketch uses 9600 baud serial (Figure 6). Each HY-SRF05 is wired to one digital trigger pin and one echo pin. The Pi's serial port is enabled and listens at /dev/ttyS0.

Conclusion

This project presents a practical and effective smart parking optimization system leveraging edge computing and deep learning for real-time and predictive parking availability management. The two-layer LSTM model running on a Raspberry Pi 3 achieves a mean absolute error (MAE) of 9.12 spots and root mean squared error (RMSE) of 13.11 spots in predicting free parking spaces, outperforming comparable CNN-LSTM benchmarks by approximately 31.5% in MAE and 38% in RMSE. The system operates with an end-to-end latency of 3 to 5 seconds for routing decisions, demonstrating feasibility for real-time urban applications.

On the vehicle management side, the automated license plate recognition (ALPR) subsystem achieves 99.26% detection accuracy and an end-to-end recognition accuracy of 93.0%, with a low processing latency of approximately 1.2 seconds per vehicle. While the recognition accuracy of 89.94% is slightly below state-of-the-art levels due to dataset imbalances, the system is robust in diverse real-world scenarios and runs entirely on resource-constrained edge hardware with no cloud dependency.

This fusion of IoT sensing, time-series deep learning prediction, and computer vision-based vehicle identification integrated into a single on-site pipeline shows significant improvements over prior works in terms of accuracy, privacy, and operational cost. Limitations related to sensor granularity, computational power, and network reliance provide avenues for future enhancement. Overall, the project sets a precedent for scalable, intelligent smart city infrastructure that can reduce congestion, lower emissions, and improve driver experience in increasingly crowded urban environments.

Overall, this project demonstrates a compelling, practical approach to enhancing smart city infrastructure through the innovative fusion of IoT, AI, and edge computing. It not only addresses the pressing problem of urban traffic congestion and pollution but also sets a precedent for future research and deployment of intelligent, distributed urban services. The paradigm of

locally executable AI models combined with minimal sensing hardware offers a scalable blueprint for numerous smart city challenges beyond parking optimization.

References

[1] M. O. Khan, M. A. Raza, A. Islam, I. U. Azam, Rashadul Islam Sumon, and H. C. Kim, “A Lightweight Deep Learning and Sorting-Based Smart Parking System for Real-Time Edge Deployment,” AppliedMath, vol. 5, no. 3, pp. 79–79, Jun. 2025, doi: https://doi.org/10.3390/appliedmath5030079.

[2] M. Krishna, A. Vinitha, S. Ravinder, and D. Mounika, “ADVANCED PARKING SLOT AVAILABILITY CHECKING SYSTEM USING RASPBERRY-Pi,” ADVANCED PARKING SLOT AVAILABILITY CHECKING SYSTEM USING RASPBERRY-Pi, vol. 18, no. 11, pp. 98–102, Nov. 2020, doi: https://doi.org/10.48047/nq.2020.18.11.NQ20240.

[3] Z. Xu, X. Tang, C. Ma, and R. Zhang, “Research on Parking Space Detection and Prediction Model Based on CNN-LSTM,” IEEE Access, vol. 12, no. 1, p. 99, Jan. 2024, doi: https://doi.org/10.1109/access.2024.3368521.

[4] D. Varshney, A. Humbe, R. Vanjari, N. Goyal, and O. Hole, “Smart Parking System,” International Journal of Scientific Research & Engineering Trends, vol. 11, no. 3, pp. 2395–5663, 2025, Accessed: Nov. 14, 2025. [Online]. Available: https://ijsret.com/wp-content/uploads/IJSRET_V11_issue3_1038.pdf

[5] “On-street Parking Bay Sensors,” data.melbourne.vic.gov.au, Jan. 10, 2025. https://data.melbourne.vic.gov.au/explore/dataset/on-street-parking-bay-sensors/information/

[6] T. Wang, S. Li, W. Li, Q. Yuan, J. Chen, and X. Tang, “A Short-Term Parking Demand Prediction Framework Integrating Overall and Internal Information,” Sustainability, vol. 15, no. 9, p. 7096, Jan. 2023, doi: https://doi.org/10.3390/su15097096.

[7] Thi-Anh-Loan Trinh, The Anh Pham, and Van-Dung Hoang (2022). "Layout-invariant license plate detection and recognition". International Conference on Multimedia Analysis and Pattern Recognition (MAPR2022), pp. 1-6.

[8] TA Pham (2023). "Effective deep neural networks for license plate detection and recognition". The Visual Computer, Vol. 39, No. 3, pp. 927-941.

[9] D. A. Subhahan, S. R. Divya, U. K. Sree, T. Kiriti, and Y. Sarthik, "An efficient and robust ALPR model using YOLOv8 and LPRNet," in *2023 International Conference on Recent Advances in Information Technology for Sustainable Development (ICRAIS)*, 2023, pp. 260–265, doi: 10.1109/ICRAIS59684.2023.10367051.