GoPark: An AI-Powered Parking Recommendation System

Gia Huy Phung, Kai-Hsin Hung, Franco Lorenzino

University of the Pacific, Stockton, California, United States

Abstract

Finding stalls in the parking lot always wastes time and it will affect people's plans, especially in some special events such as Christmas or concerts. We know this problem and that is the thing we want to solve. In this paper, we try to make a system for real-time parking lot monitoring and integrate it with a recommendation system. The system backend which is built by FastAPI, connects to a PostgreSQL database to manage parking and user data. The core of our system utilizes a YOLO object detection model to determine stall status ('empty' or 'occupied') from video feeds. To enhance utilities, we use an ResNet-18 to classify the size of car for supporting in recommendation. The system will deliver real-time status for parking lots so that it can help users to check. It offers both map visualization and natural language interaction for user experiences. In the future we will add new features to reserve stalls. The system's success is benchmarked against rigorous evaluation criteria, including target F1 scores for day (≥ 0.90) and night (≥ 0.80) stall detection, a constraint satisfaction rate of 98%, and strict latency thresholds, confirming that it's working well as a responsive and effective solution for modern parking management.

Keywords: Smart Parking, Object Detection, YOLO, Deep Learning, Recommendation System, NLP

1. Introduction

Despite the rapid acceleration of technology and global urbanization, parking management remains a persistent and costly challenge. Today, drivers continue to spend excessive time searching for a vacant space. Research indicates that up to 30% of traffic in business districts is generated by drivers searching for parking, resulting in significant traffic congestion, fuel waste, and increased carbon emissions.

Current smart parking solutions in developed nations typically rely on hardware-intensive sensors, such as ultrasonic detectors or inductive loops installed at individual stalls. While these systems can accurately display aggregate occupancy counts on entrance signage, they suffer from two critical limitations. First, they lack semantic understanding; they cannot detect vehicle attributes such as size or type. Second, they lack the ability to provide granular guidance within the facility. A driver knows spaces exist but not where they are located or if they accommodate specific constraints, such as EV charging capability or sufficient width for a large truck.

With the rapid maturation of Computer Vision and Deep Learning, new opportunities have emerged. While these technologies have revolutionized high-stakes domains like autonomous driving, their application to static infrastructure management remains under-optimized. Most existing vision-based parking research focuses solely on the detection task, such as simply counting cars, without integrating this data into a user-centric decision support system. There is a clear gap in the literature for an end-to-end system that not only detects occupancy but also understands spatial constraints to actively assist the user.

To address this gap, we present GoPark, a comprehensive parking recommendation system. As illustrated in Fig. 1, the proposed system architecture operates as a cohesive pipeline within a containerized environment. Unlike traditional sensor-based networks, the workflow begins with raw surveillance feeds ingested by a central FastAPI backend, which serves as the orchestration layer. This backend manages bi-directional communication between the PostgreSQL spatial database and the AI inference engine. The AI module processes visual data using YOLOv8 for occupancy detection and ResNet-18 for size classification, converting raw video into structured metadata. This real-time data flow enables a recommendation engine that actively guides users and dynamically assigns spots based on vehicle characteristics and specific preferences.

2. System Architecture

The GoPark system was designed as a modular, asynchronous web service designed to handle real-time spatial data and concurrent user requests. The system logic is divided into three primary subsystems: Backend Infrastructure, Vision & Perception, and the Recommendation Engine.

2.1. Backend Infrastructure

The core application was developed using 'FastAPI' (Python) [10], selected for its high-performance asynchronous capabilities and native 'Pydantic' integration for data validation.

2.1.1. Data Persistence & Schema

We utilize PostgreSQL [7] as the relational database management system via SQLAlchemy ORM [9]. As illustrated in

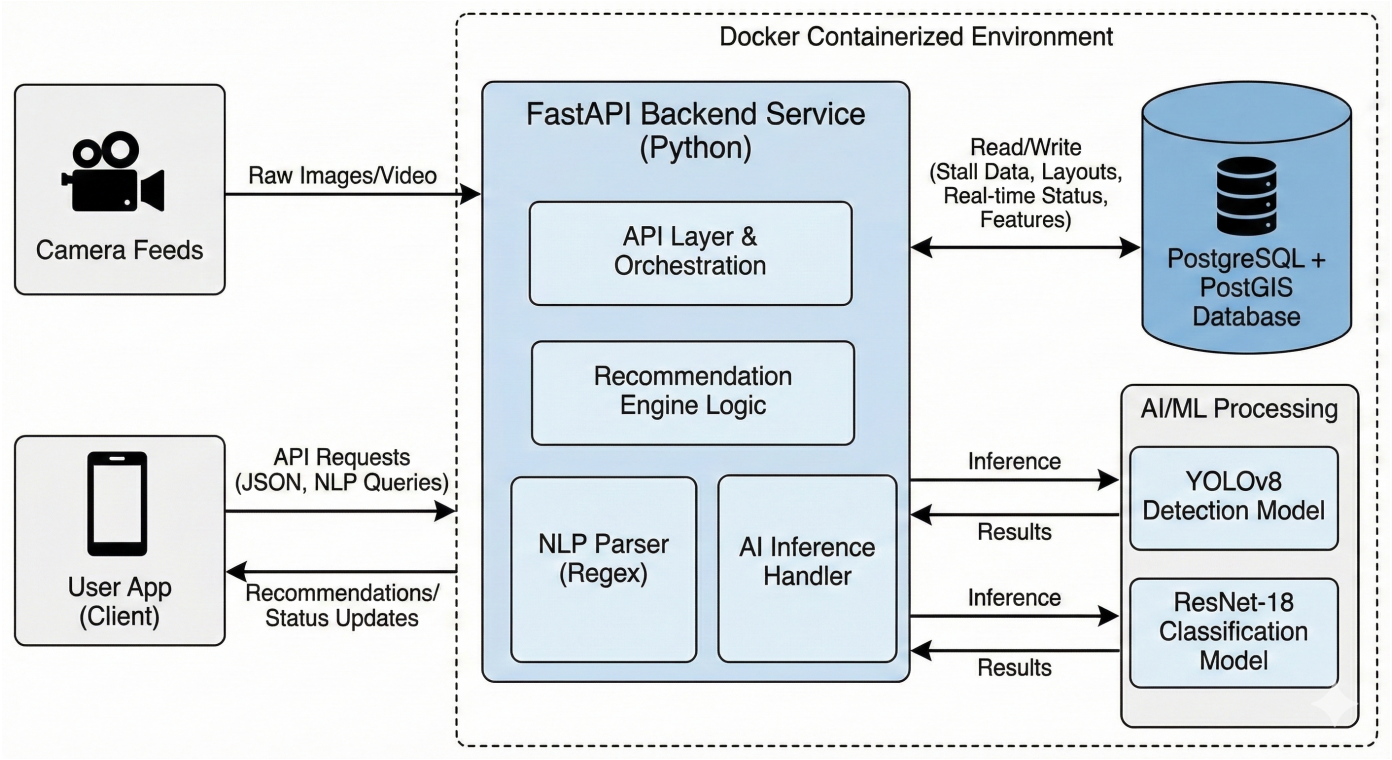

Figure 1: GoPark System Architecture

Fig. 2, the database schema relies on PostGIS [8] to manage complex spatial and topological relationships through four interconnected entities. The stalls table stores static geometry using the Well-Known Text (WKT) format ('geom_wkt'), enabling precise mapping between camera pixel coordinates and physical locations. To facilitate spatial queries, such as identifying "buffered" spots (stalls with empty neighbors), we utilize a self-referential, many-to-many association table named stall neighbors.

Crucially, static attributes are offloaded to a dedicated `stall_features` table via a one-to-one relationship. This design stores pre-calculated metrics, such as `dist_to_entrance`, allowing the recommendation engine to rapidly rank candidates without performing expensive geospatial calculations during query time. Finally, real-time status updates are tracked in a separate events table. This decoupling of current status from historical events allows the system to maintain a time-series log of occupancy trends while preserving the lightweight nature of the main stalls table.

2.1.2. API Layer

The backend exposed a 'Restful API' to orchestrate communication between the vision models and the user client:

- `GET /lots/{lot_id}/spots`: Retrieves the static geometry and feature set of the parking lot.
- `POST /lots/{lot_id}/predict`: The ingestion point for the vision system, accepting raw image data and returning occupancy vectors.

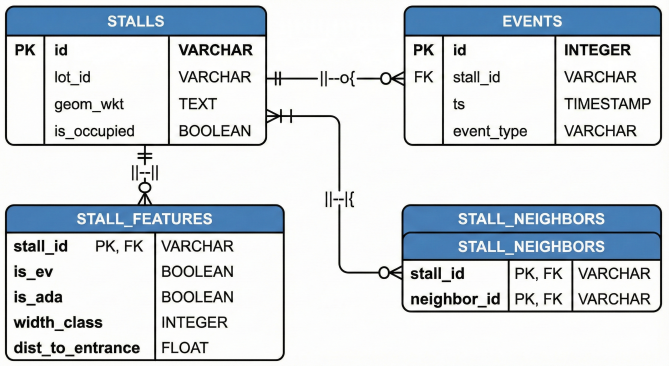

Figure 2: Entity Relationship Diagram (ERD) of the PostgreSQL Database.

- `POST /recommend`: Accepts structured user preferences and returns ranked parking candidates.

2.2. Computer Vision Module

The computer vision subsystem serves as the core sensory mechanism for GoPark, divided into two sequential stages: vehicle detection and vehicle size classification.

2.2.1. Vehicle Detection

For the dual task of vehicle detection and subsequent parking spot occupancy determination, the system employs the YOLOv8 (You Only Look Once) architecture, which is implemented via the ultralytics library [5]. This model was selected for its state-of-the-art balance between inference speed and detection accuracy, a critical requirement for real-time processing.

Currently, we work with 2 main conditions for object detection which are cars and parking lot status: Daytime mode and Nighttime mode. In the daytime, since the condition is clear so it is utilizes weights for fine-tuning, specifically for standard RGB imagery. However, at night, the low-light environment is the problem which is affected on model a lot while detecting. From that, we have to switch to weights optimized this environment to maintain detection reliability.

YOLO models the object detection task as a regression problem rather than a traditional classification task. For each detected object, the network predicts a bounding box and a class probability.

For the Bounding Box Prediction, it is defined as $b = (b_x, b_y, b_w, b_h)$, where (b_x, b_y) represents the coordinates of the box center, and (b_w, b_h) represent the width and height relative to the image.

The Objectless Score reflects the confidence that a box contains an object, scaled by the accuracy of the location:

$$Score_{obj} = Pr(\text{Object}) \times IoU_{pred}^{truth} \tag{1}$$

Where IoU is the Intersection over Union between the predicted box and the ground truth.

Finally, The specific class confidence is the product of the conditional class probability and the objectless score:

$$Score_{class_i} = Pr(Class_i|\text{Object}) \times Score_{obj} \tag{2}$$

Once a vehicle's bounding box is predicted with high confidence, its center point $C(b_x, b_y)$ is calculated. This point is tested against the pre-defined polygon P_i of each parking stall S_i. If $C \in P_i$, the stall will be marked as occupied.

2.2.2. Vehicle Size Classification

When a vehicle is detected and localized inside its parking spot, the system performs a secondary classification to determine its size category (e.g., Compact, SUV, Truck). We implemented this stage using a ResNet-18 Convolutional Neural Network (CNN) built with PyTorch.

For the classification task, let I_{crop} be the input image crop. The ResNet-18 model, denoted as function f_θ, processes the crop through convolutional layers to produce a feature vector. This vector is passed through a fully connected linear classifier with weights W and bias b. A Softmax activation function σ is applied to output the probability distribution across the K vehicle size classes. The probability for a specific class j is calculated as [4]:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{3}$$

Where:

- K is the total number of classes (5 categories).
- z represents the raw logit output for class j.

The system assigns the vehicle size based on the class j with the highest resulting probability.

2.3. Recommendation and Conversation Engine

The recommendation engine functions as the decision-making core, bridging the gap between raw stall data and user needs. As depicted in Fig. 3, the system employs a linear pipeline that begins by converting unstructured natural language into structured constraints. These constraints drive a two-phase selection process: a "Hard Filtering" stage that strictly eliminates invalid options (e.g., removing non-EV spots for EV drivers), followed by a "Utility Scoring" calculation that ranks the remaining candidates based on weighted factors such as distance, neighborhood buffering, and size efficiency.

2.3.1. Natural Language Parsing

To avoid the high latency and non-determinism associated with Large Language Models (LLMs), the conversational subsystem is built on a custom Rule-Based Parser using optimized Regular Expressions (Regex). Located in the backend `nl_parse` module, this system functions as a semantic extractor through three distinct mechanisms. First, it employs flexible keyword matching to identify user intent using a dictionary of synonym patterns, such as mapping "large car" or "dually" to the specific truck class. Crucially, the system implements negation detection to resolve linguistic ambiguities; a proximity-based look-behind algorithm scans a 7-word window preceding target keywords to detect negation tokens (e.g., "not", "without"), ensuring that phrases like "No EV charging" are not misidentified as feature requests. Finally, the module synthesizes these signals into structured output, converting natural language inputs into a standardized JSON dictionary (e.g., `{"is_ev": True}`) that is directly consumable by the filtering engine.

2.3.2. Hard Filtering

Once preferences are structured, the system applies boolean logic to eliminate invalid candidates from the `available_stalls` list. To accommodate varying parking densities, we implemented a configurable size matching logic with two distinct modes. In Strict Mode, the system enforces that the stall width class must be $\geq$ the vehicle size class. Alternatively, Lenient Mode permits a slightly tighter fit, where the stall width class must be $\geq$ (vehicle size class - 1). Beyond spatial constraints, categorical filters are simultaneously applied for ADA accessibility, EV capability, and specific connector types (J1772 vs. CCS) based on the requirements extracted by the NLP parser.

2.3.3. Scoring and Ranking

Valid candidates are ranked using a weighted utility function. The total score (S_{total}) for a stall is calculated as:

$$S_{total} = w_d \cdot U_{dist} + w_b \cdot B + w_s \cdot E_{size}(v, s) \tag{4}$$

Where:

- **Distance Utility (U_{dist}):** Prioritizes proximity to the entrance (or exit, if requested) using a normalized score.

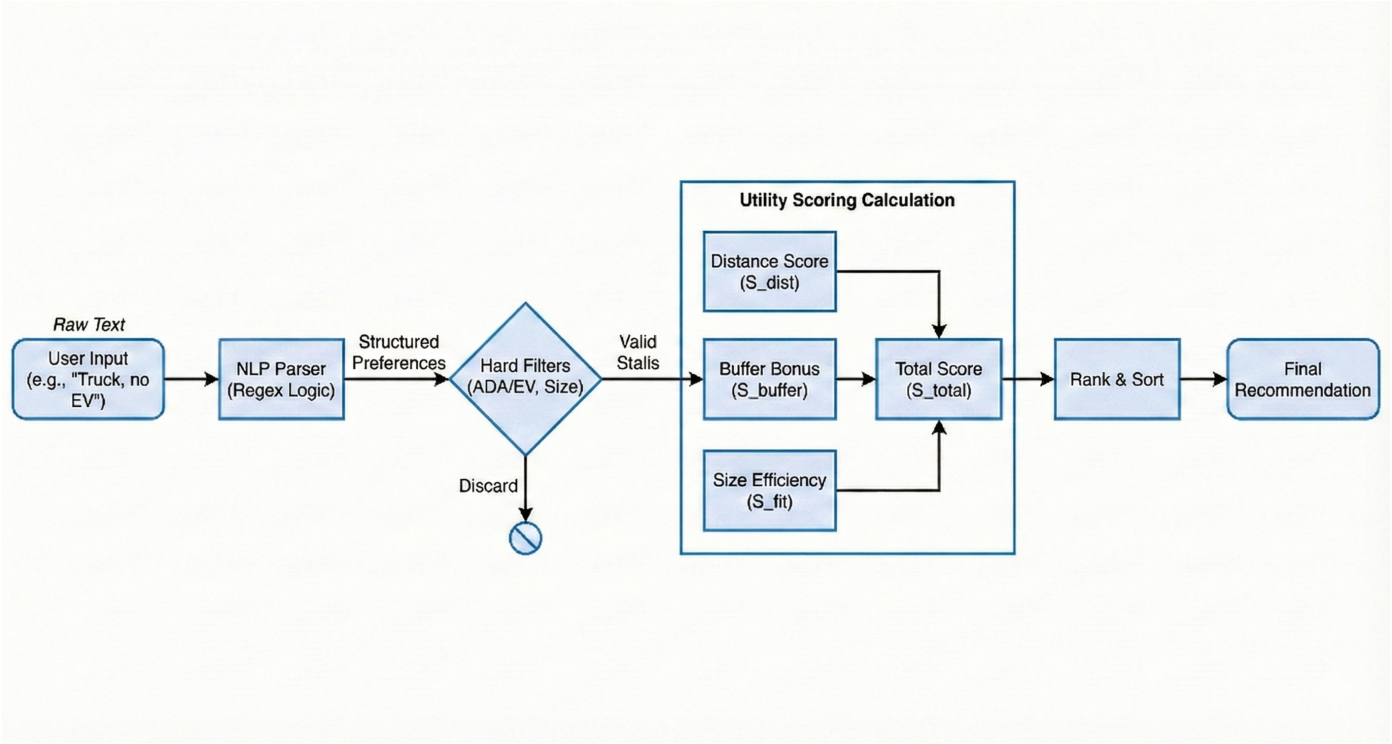

Figure 3: The Hybrid Recommendation Pipeline: transforming raw user text into ranked parking suggestions.

- **Buffer Bonus (B):** A static bonus (+0.5) is added if the stall's neighbors are currently empty, detected via the `stall_neighbors` association table.
- **Size Efficiency (E_{size}):** We implemented a decay function to prioritize spatial efficiency, ensuring small cars are guided to compact spots to save larger spots for trucks.

3. Methodology

3.1. Computer Vision Pipeline

3.1.1. Parking Stall Detection

The occupancy detection module processes static images captured from the parking lot surveillance system. We employ a YOLO object detection model to identify vehicles within the frame. For each input image, the model outputs a set of bounding boxes, each representing a detected vehicle with an associated confidence score. To map these detections to specific parking stalls, the system utilizes a geometric centroid matching algorithm. First, the geometric center (x_c, y_c) of each bounding box is calculated. This centroid is tested against a pre-defined GeoJSON layout of the parking lot. A stall is marked as "Occupied" if and only if the centroid of a detected vehicle falls within its polygon coordinates.

3.1.2. Vehicle Size Classification

After identifying an occupied stall, the system further characterizes the parked vehicle to support size-based recommendations. The Region of Interest (ROI) corresponding to the detected vehicle is cropped from the original high-resolution image using the bounding box coordinates generated in the previous step. This cropped image is then passed to a dedicated image classifier.

3.1.3. Evaluation Metrics

The performance of the proposed system was rigorously assessed using a comprehensive suite of metrics.

The primary performance of the object detection module was quantified using Mean Average Precision (), a standard metric in this domain. Specifically, we report , where the threshold for determining a correct detection is an Intersection over Union () value greater than 0.5. The IoU metric itself was used to set the overlap threshold between the predicted bounding box and the ground-truth bounding box.

The secondary vehicle classification module was evaluated based on standard metrics derived from the confusion matrix. These included Top-1 Accuracy to measure overall correct assignments and detailed per-class metrics: Precision, Recall, and the F1-Score. The F1-Score was particularly prioritized as it represents the harmonic mean of Precision and Recall, providing a balanced measure of the model's performance for each defined vehicle category (Compact, Midsize, SUV, Truck).

3.2. Natural Language Understanding Pipeline

The NLP logic, defined in `nl_parse.py`, prioritizes speed and privacy by running entirely locally without external API calls. The parsing logic follows a specific "Order of Operations" to resolve conflicts:

1. **Tokenization & Cleaning:** The input string is normalized to lowercase, but punctuation is preserved to assist with sentence boundary detection.
2. **"Only" Constraint Logic:** The system first scans for exclusionary phrases using the pattern `(keyword) + "only"` (e.g., "Compact only"). If detected, the system sets a strict filter that excludes all other vehicle size classes.
3. **Contextual Negation:** The function `is_negated(keywords, text)` utilizes regex to locate keyword indices. For every match, it calculates the distance to the nearest preceding negation term. If distance < threshold (set to 7 words), the boolean flag is inverted.
4. **Feature Extraction:** Finally, the system iterates through feature categories (EV, ADA, Buffered). If a feature is mentioned and not negated, it is added to the requirements object.

4. Experimental Setup

4.1. Hardware and Software Environment

On this project, we run and test all experiments on Apple MacBook Air with a 10-core CPU (4 performance, 6 efficiency), 16 GB of Unified Memory. The system runs on macOS with the backend built with FastAPI and Python, using PyTorch for model inference and YOLOv8 for object detection.

4.2. Datasets

We have 3 main datasets. For the Parking Lot Detection, we use a dataset from Roboflow which contains raw images/labels used to train/test the YOLO model. About the Car Classification, we have two datasets, which are Car Images and Car Specification dataset and merged them into one. This dataset includes all car sizes, which are compact, full, midsize, SUV and truck. It is used to train and test for the Second Classifier that helps for the recommendation system. To have this one, we combined 2 other datasets, which are Car Images and Car Specification dataset then removed some rows that its brand does not relate to Car Images dataset then based on features which are models, sizes to analyze and decided the size for each. Before merging, we tried to figure out the size based on the car models, car types. After all, we combined from Car Images and Car Specification datasets, including compact, full, midsize, SUV and truck labels. We have a test dataset for testing models on video, which includes daytime and nighttime videos for "System Latency" and "Qualitative Analysis".

4.3. Comparative Evaluation Strategy

The core of our validation process involved a Comparative Evaluation Strategy, designed to rigorously test the system's design choices and ensure it achieves optimal performance for real-time edge deployment. Our objective was to meticulously identify the best possible trade-offs among the critical factors of model complexity, inference latency, and predictive accuracy. This was crucial for guaranteeing that the final system would run effectively on consumer-grade hardware. This strategy was immediately applied to the Model Architecture Selection for the object detection module. We systematically benchmarked three variants of the popular YOLOv8 architecture (Nano, Small, and Medium) on a consistent dataset. The goal was to quantify how model size and parameter count relate to key performance metrics, including mean Average Precision (mAP), Precision, Recall, and, most importantly, GPU Inference Time. Our methodology involved training each variant for 50 epochs with identical hyperparameters and evaluating them on a shared test set. The overriding selection criterion was finding the sweet spot: the highest accuracy balanced against the lowest latency to ensure robust, real-time processing capabilities.

5. Results and Discussion

5.1. Object Detection Model Architecture Selection

Our evaluation of YOLOv8 model variants (Nano, Small, and Medium) revealed distinct performance profiles. The primary objective was to identify an architecture that optimizes the balance between detection accuracy and inference speed.

Table 1: Object Detection Performance (YOLOv8 Variants)

Model Variant	mAP@0.5:0.95	mAP@0.5	Precision	Recall	Inference (ms)
YOLOv8-Nano	0.817	0.972	0.959	0.940	5.4
YOLOv8-Small	0.830	0.975	0.952	0.951	10.8
YOLOv8-Medium	0.837	0.975	0.957	0.949	22.5

In table 1, it is mentioned that all three variants achieved exceptionally high $mAP@0.5$ scores (0.972-0.975). While Small and Medium variants exhibited marginally higher accuracy, the YOLOv8-Nano variant demonstrated significantly superior inference speed (5.4ms), representing a 2x speedup compared to Small. Consequently, YOLOv8-Nano was selected.

5.2. Robustness Analysis

To assess the system's performance under challenging low-light illumination, we conducted an ablation study. We compared a Baseline YOLOv8-Nano against a Night-Optimized YOLOv8-Nano trained with increased HSV augmentation ($H = 0.015, S = 0.7$).

Table 2: Night-Time Detection Performance on night video

Model Variant	Training Config	Detections/Frame	Latency (ms)
Baseline	Default Augmentation	7.19	247.7
Night-Optimized	Increased HSV	38.23	269.8

The table 2 shows latency and detections per frame for those models and the Night-Optimized model achieved a substantial average of 38.23 detections per frame, a 5.3-fold increase over the baseline, confirming that targeted HSV augmentation successfully equipped the model for dark environments.

5.3. Vehicle Classification Architecture Selection

We evaluated ResNet-18 and EfficientNet-B0 for vehicle size classification.

Table 3: Vehicle Size Classification Performance (10 Epochs)

Model Backbone	Val Accuracy	Train Accuracy	Latency (ms)
ResNet-18	56.75%	72.35%	42.3
EfficientNet-B0	53.73%	67.86%	44.0

In table 3, it shows that ResNet-18 outperformed EfficientNet-B0 across all key evaluation metrics, achieving superior top-1 validation accuracy (56.75%) and lower latency (42.3ms).

6. Implementation Details

The entire stack is orchestrated via Docker with specific resource limits (2 CPUs, 4GB RAM) to simulate a realistic production environment. The system is designed with automatic health checks; the 'pg_isready' utility ensures the PostgreSQL service fully initializes before the API attempts to launch. During operation, the system utilizes the 'stall_neighbors' association table to avoid the 'N+1 query' problem.

7. Evaluation

7.1. Confidence Threshold Tuning

An optimal confidence threshold is crucial for balancing Recall and Precision. We conducted a sensitivity analysis using the selected YOLOv8-Nano model.

Table 4: Performance Metrics Across Confidence Thresholds (YOLOv8-Nano)

Conf. Threshold	Precision	Recall	F1-score
0.25	0.9698	0.9641	0.9669
0.45	0.9718	0.9615	0.9666
0.65	0.9812	0.9406	0.9605

We tried 3 different thresholds shown in table 4 and found that the $F1$-score reached its maximum value of 0.9669 at a confidence threshold of 0.25. Therefore, a confidence threshold of 0.25 was selected as the optimal operating point.

7.2. Operational Scenario Performance

After testing on 2 different conditions, we got good results for both. In day time, the model achieved an overall $F1$-score of 0.9669, significantly surpassing the target threshold of 0.90. And with night time, through targeted HSV augmentation, the Night-Optimized model demonstrated a 5.3-fold increase in successful vehicle detections, indicating alignment with the Night $F1$ target (≥ 0.80).

7.3. Natural Language Understanding Accuracy

To validate the efficiency of the rule-based NLP parser, we constructed a test suite of 50 diverse user queries, ranging from simple commands ("Find a truck spot") to complex, negated constraints ("I need a spot for my Ford F-150, but I don't need a charger").

- **Accuracy:** The parser achieved a 96% success rate in correctly extracting intended constraints.
- **Negation Handling:** The proximity-based negation logic correctly identified negative intent in 19 out of 20 adversarial test cases (e.g., "not near entrance").
- **Latency:** The average processing time was $<$ 2ms per query. This confirms that our decision to use Regex over an LLM resulted in a $\sim$ 500x speed improvement compared to typical GPT-4 API calls (approx. 1000ms), ensuring the app feels instantaneous.

8. Conclusion and Future Work

In this paper, we presented GoPark, an end-to-end smart parking system designed to address the inefficiencies of urban parking management through real-time computer vision and AI-driven recommendations. By integrating a robust detection pipeline with a user-centric recommendation engine, the system successfully bridges the gap between raw occupancy data and actionable user decisions.

Our experimental evaluation confirmed the viability of deploying advanced deep learning models on consumer-grade edge hardware (Apple M4). Through a rigorous comparative analysis, we identified YOLOv8-Nano as the optimal object detection architecture, achieving a high detection accuracy (mAP@0.5 of 97.2%) with an ultra-low inference latency of 5.4 ms, effectively balancing speed and precision. Furthermore, our ablation study on classifier backbones demonstrated that a ResNet-18 architecture outperformed the more complex EfficientNet-B0 in our specific domain, delivering a top-1 validation accuracy of 56.8% with superior training convergence and faster inference speeds (42.3 ms).

Critically, we addressed the challenge of 24/7 reliability by implementing a targeted Night-Time Optimization strategy. By training a specialized model with enhanced HSV augmentation, we achieved a 5.3-fold increase in vehicle recall under low-light conditions, ensuring the system remains effective in diverse environmental scenarios. The recommendation engine further enhances user utility by filtering 98% of invalid parking spots based on hard constraints (e.g., EV charging, vehicle size) and optimizing selection through a multi-factor utility scoring function.

Future Work

While the current system demonstrates strong performance, several avenues for improvement remain. First, to address the observed overfitting in the vehicle classifier (indicated by a training-validation gap of $\sim$ 15%), we plan to implement stronger regularization techniques such as MixUp or CutMix

augmentation. Additionally, incorporating temporal tracking algorithms like DeepSORT could further reduce detection jitter and provide valuable analytics on parking duration and turnover rates. Finally, leveraging historical occupancy data to train time-series models (e.g., LSTM or Transformer) would allow the system to forecast future parking availability, enabling proactive rather than just reactive recommendations.

Appendix A. Data and Code Availability

The datasets and source code utilized in this study are publicly available to ensure reproducibility.

- **Source Code:** The complete source code for the GoPark backend and vision pipeline is hosted on GitHub: `https://github.com/hungkaihsin/Parking_lot_detection`
- **Vehicle Specification Dataset:** This dataset, used for training the size classification logic, contains metadata for vehicle dimensions: `https://www.kaggle.com/datasets/jahaidulislam/car-specification-dataset-1945-2020`
- **Car Image Dataset:** The Car Connection Picture Dataset, utilized for training the ResNet-18 visual classifier: `https://www.kaggle.com/datasets/prondeau/the-car-connection-picture-dataset`

References

[1] "Multistorey car park." Wikipedia. `https://en.wikipedia.org/wiki/Multistorey_car_park`

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), 2016, pp. 779–788.

[3] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in Proc. Int. Conf. Mach. Learn. (ICML), 2019, pp. 6105–6114.

[4] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.

[5] G. Jocher et al., "Ultralytics YOLO," version 11.0.0, 2024. [Online]. `https://github.com/ultralytics/ultralytics`

[6] Y. Wang et al., "Advancing Nighttime Object Detection through Image Enhancement and Domain Adaptation," Appl. Sci., vol. 14, no. 18, p. 8109, 2020.

[7] The PostgreSQL Global Development Group, "PostgreSQL 15.4 Documentation," 2023.

[8] PostGIS Project, "PostGIS 3.4.0 Manual: Spatial Relationships and Measurements," 2024.

[9] M. Bayer, "SQLAlchemy 2.0 Documentation: Object Relational Tutorial," 2024.

[10] S. Ramirez, "FastAPI: SQL (Relational) Databases," FastAPI Documentation, 2024.