Real-Time Parking Finder for Campuses and Streets

Kunduri Sai Nikhil Reddy **Kunduris1@udayton.edu**

Darshan Jigala Channa Reddy **Jigalachannareddyd1@udayton.edu**

A project submitted in partial fulfillment of the requirements for the course

CPS 622

Software Project Management

Professor

Bayley King, PhD

Spring 2025

Department of Computer Science

University of Dayton

Sai Nikhil Reddy Kunduri
Darshan Jigala Channa Reddy

Project Description :

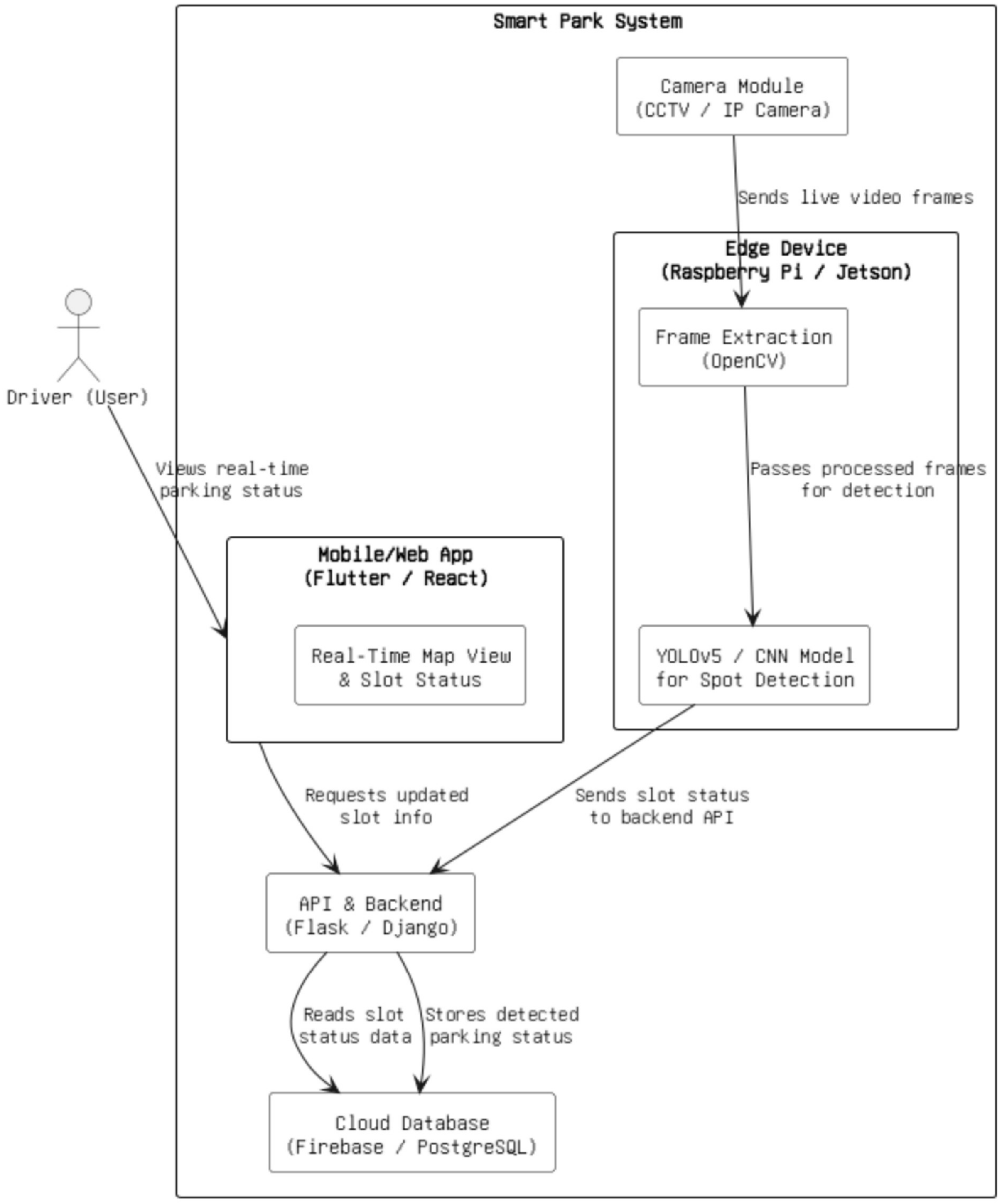
Elevator Pitch :

Smart Park is a real-time, camera-based parking finder system that helps drivers quickly locate available parking spaces in large open areas like university campuses or public streets. With growing traffic and limited parking, drivers often waste time and fuel searching for free spots, increasing congestion and stress.

Our solution uses computer vision and real-time data processing to identify vacant spots using camera feeds and display this information on a user-friendly mobile/web interface. Unlike current systems that rely on costly sensors or only cover indoor garages, Smart Park is designed to be scalable, low-cost, and accurate, even in large outdoor environments.

High Level Architectural Diagram :

This Diagram shows the high-level architectural diagram for real time parking finding system for campuses and streets.


Smart Park System
Camera Module (CCTV / IP Camera)
Sends live video frames
Edge Device (Raspberry Pi / Jetson)
Frame Extraction (OpenCV)
Passes processed frames for detection
YOLOv5 / CNN Model for Spot Detection
Driver (User)
Views real-time parking status
Mobile/Web App (Flutter / React)
Real-Time Map View & Slot Status
Requests updated slot info
Sends slot status to backend API
API & Backend (Flask / Django)
Reads slot status data
Stores detected parking status
Cloud Database (Firebase / PostgreSQL)

Detailed Description for High Level Architecture Diagram:

The Real time Parking finder system is a modular and scalable software solution designed to solve the real-world problem of finding available parking spaces in real time across large outdoor environments such as university campuses, city streets, and hospitals. Unlike traditional parking systems that rely on embedded ground sensors—which are expensive, hard to maintain, and unsuitable for outdoor lots—Smart Park leverages modern tools such as camera-based input, computer vision, edge computing, and cloud platforms to provide a low-cost and efficient alternative. The system is carefully engineered using multiple integrated components, each serving a distinct technical function, while remaining interoperable and easy to deploy.

1. Camera Feed (Input Layer)

CCTV or IP cameras are installed to cover the parking areas. These cameras are configured to continuously capture real-time video footage of the designated zones, typically streaming at 15–30 FPS. The cameras are chosen based on weatherproofing, resolution (minimum 1080p), and infrared capability for low-light performance. The video feed is streamed over a local network to the **edge device** instead of the cloud, reducing network congestion, lowering latency, and ensuring real-time performance.

2. Edge Processing (Edge Layer)

Video feeds are processed locally using low-power, cost-effective edge computing devices like **Raspberry Pi 4** or **NVIDIA Jetson Nano**. These devices are equipped with a lightweight, custom-trained version of **YOLOv5** (You Only Look Once) or a Convolutional Neural Network (CNN), capable of detecting vehicles in each frame. The YOLO model is chosen for its speed (real-time inference) and accuracy in object detection. Once the vehicle positions are detected in a frame, the system maps these against pre-defined bounding boxes that represent parking slots, determining whether each slot is occupied or empty. This decision, along with a timestamp, is then passed to the next module.

3. Computer Vision Module (Processing Layer)

This module combines **OpenCV** for real-time video processing and pre-processing with **PyTorch** or **TensorFlow** for executing the deep learning model. The parking lot's layout is hardcoded into the system as a set of parking slot coordinates. Each frame is analyzed and compared with these coordinates to determine slot-level status. The model labels each slot with binary status—occupied or vacant—and packages the data into a structured JSON payload, which includes the slot ID, status, confidence score, timestamp, and location ID. This payload is passed to the backend API.

4. Backend and API Gateway (Middleware Layer)

The backend component is a **RESTful API**, built using **Flask** or **Django**, which acts as the interface between the vision module and the frontend/mobile application. It handles both **inbound POST requests** from the vision module and **outbound GET requests** from user applications. All communication is carried in lightweight JSON objects. The API layer also includes basic validation and access control. Documentation for the API is maintained using tools like **Swagger** or **Postman** for testing and validation.

5. Real-Time Database (Data Storage Layer)

The processed slot status data is stored in a **cloud-based database**, such as **Firebase Realtime Database** (for live push-based updates) or **PostgreSQL** (if relational queries and analytics are needed). The database stores:

- Slot IDs and their current occupancy status
- Parking lot metadata (location, number of spots, timestamps)

- Historical logs (for usage analytics)

The data schema is lightweight and indexed for fast read/write performance. The database enables real-time syncing with the frontend interface and scalability for multiple lots or zones.

6. Mobile/Web Application (User Interface Layer)

The user-facing component of the system is a **mobile application (built using Flutter)** and a **web interface (built using ReactJS)**. Both apps allow drivers to view a live parking map where each slot is visually represented and color-coded:

- **Green**: vacant
- **Red**: occupied
- **Gray**: unknown or under maintenance

Users can search by lot, filter by availability, and receive instant updates as the database changes. The app connects to the backend using periodic polling or **WebSocket** (for real-time). It is designed to work across all devices and browsers, with responsive design and accessibility considerations.

7. Deployment Infrastructure (DevOps Layer)

All backend components (API, vision module) are **containerized using Docker**, enabling consistent deployment across development and production environments. These containers can be hosted on cloud platforms like **Google Cloud Platform (GCP)** or **AWS EC2**, depending on the institution's infrastructure. For university projects, an on-premises server can also host the services. CI/CD tools can be optionally added for automatic updates. **Monitoring** tools like Grafana or Google Cloud Monitoring can track service health, performance, and uptime.

8. Technical Tools and Resources

- **Computer Vision & ML**: OpenCV, YOLOv5, PyTorch/TensorFlow

- **Backend/API**: Flask or Django
- **Database**: Firebase Realtime DB or PostgreSQL
- **Frontend**: Flutter (mobile), ReactJS (web)
- **Deployment**: Docker, AWS EC2 or GCP
- **Documentation & Testing**: GitHub, Swagger, Postman
- **Diagramming**: Draw.io (for architectural visualization)

Connection to Deliverables

This system architecture directly supports the deliverables defined under each task:

- **Task 1**: Camera module and video integration with test logs and frame outputs.
- **Task 2**: Object detection using YOLO, with model training files, result videos.
- **Task 3**: Flask/Django API endpoints with Swagger/Postman documentation.
- **Task 4**: Real-time sync with Firebase/PostgreSQL schemas.
- **Task 5**: Fully interactive mobile/web UI with demo screenshots and feedback reports.

Main Technical Tasks:

Task	Task Name	Task Description
1.	Camera Feed Integration	Capture live video from IP/CCTV cameras and format for processing.
2.	Parking Slot Detection	Apply object detection (YOLO) to video frames to identify vacant/occupied spots.
3.	API and Backend Logic	Create REST APIs to store and retrieve slot data from the database.
4.	Real-Time Database Sync	Sync detection results with a cloud DB for consistent and fast access.

5.	User Interface Design	Develop a responsive interface showing live parking availability.

List of Deliverables:

Task	Task Name	Deliverables
1.	Camera Feed Integration	Camera interface script, Test frame output, Technical report.
2.	Parking Slot Detection	YOLO training notebook, Model files, Result video, Code review notes.
3.	API and Backend Logic	API endpoints (Postman), Swagger docs, Working packaged backend.
4.	Real-Time Database Sync	Firebase schema, Sync test logs, Slot status sync demo, Final system integration demo.
5.	User Interface Design	UI screenshots, Live demo map, Presentation with stakeholders, Final report, Working packaged software product.

Risk Analysis Plan:

Defined Risk Levels:

- **High**: Causes major failure or inaccuracy in core functionality (e.g., parking detection not working)
- **Medium**: Causes delays or reduces app responsiveness (e.g., API delays)
- **Low**: Cosmetic or user-interface bugs (e.g., UI layout on small screens)

Risk Table:

Risk	Description	Initial Risk	Mitigation Strategy	Final Risk
1.	Camera footage is unclear or blocked.	High	Use weatherproof IR cameras; maintain clean lenses.	Medium
2.	Object detection model fails in low lighting.	High	Train with varied lighting data and use night mode.	Medium
3.	API bottlenecks delay updates.	Medium	Use caching (Redis), async queues.	Low
4.	User interface fails on mobile devices.	Medium	Use responsive design and test across devices.	Low
5.	Parking lot layout changes after deployment.	Low	Add admin panel to update slot coordinates dynamically.	Low

Sai Nikhil Reddy Kunduri
Darshan Jigala Channa Reddy