Video License Plate Parking System

Dominik Schweigl Daniel Wenger Nikola Adzic

I. Introduction

This project implements a distributed parking system that captures the license plate of incoming and leaving vehicles at the gates of a car park. These detections are used to be able to pay only with one's license plate number at the payment station. With this system it is possible to allow customers to pay without tickets. Instead, customers pay from their smartphones in the web or at payment stations using their license plate number. In addition, the web app allows customers to view car parks with open spaces and book a parking spot in advance.

The dataset we will use is the Car License Plate Detection dataset from Kaggle. It features more than 400 images of car front/ rear views with clearly visible license plates. For a real system one would fine tune a corresponding detection model on the specific surroundings and camera angle of the actual entries to the parking space. We will use a precrafted YOLOv11 license plate detection model for this project.

II. System architecture

Regarding the components and functionality of this project, we will have 2 cameras pointing to the entry and exit respectively at each car park. There will be 2 gates for the entry and exit to the parking space controlled by an edge server. The edge server will process the video stream of the cameras and run the license plate detection algorithm. Once a license plate has been detected for a predefined amount of time in front of the entry, the entry gate will open, and the car will be registered in a local data store to guarantee data persistence. When the edge server detects a vehicle that has already paid at the exit, which is queried from the cloud, the gate will open, and a green signal will appear on the traffic light. If the car has not paid or the amount of time to leave the parking space after payment has expired, the gate will not open, and a red signal will indicate the car to return and pay. Customers can pay their parking fee using the web application or the local payment station using their license plate. A diagram of these components can be seen in Figure 1.

The cloud will receive updates from the edge servers on their current open capacity. This information allows to compute the nearest free parking spots for any destination a customer has. The customer can also book a parking spot in advance for a given car park, which sends an event to the corresponding edge server to reserve a spot for the car with the customers license plate.

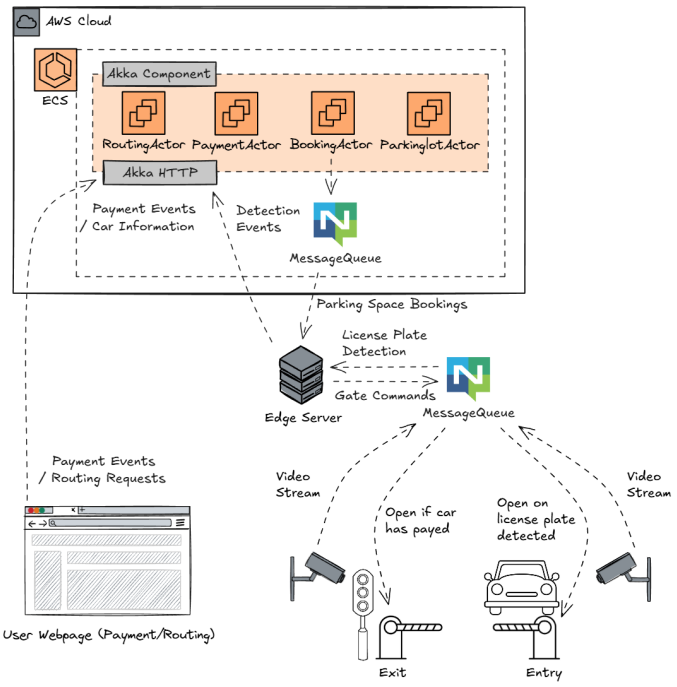

Fig. 1. The architecture diagram for this project. For visual purposes only a single car park system is illustrated.

III. Implementation details

Regarding our Service choices, the Akka Framework is used in the Cloud Layer as it is mandatory for this assignment. Actors will store their state in a DynamoDB table because we want to ensure reliability even if some server with Akka Actors crashes. The communication from the cloud to the edge server is done through a message queue to make the edge server location transparent for the cloud. The edge server computes the detection algorithm on the video stream and manages the local IoT devices as well as parking data to keep network traffic to the cloud low. This way we only send small messages of detection events with current open capacity to the cloud. Communication from the edge to the cloud Akka Backend does not go through the message queue and instead uses Akka's HTTP REST API. This is because some of the calls to the cloud for information retrieval like a GET request for car payment status fit better as synchronous calls rather than asynchronous messages. Therefore, to have a uniform access point for the edge we will use the REST API also for capacity update events. Though we leave the possibility open to also use the message queue for communication from the edge to the cloud backend.

For the edge servers we will use Python to implement the license plate detection as Python is the de facto standard for machine learning related tasks. This means that controls of the gates and traffic lights will also be done via python.

A. IoT Implementation

The camera is simulated by continuously sending images from the license plate detection dataset and a reference empty street scene to the edge server using a NATS message queue. It is made sure that the data sent by the entry and exit cameras is consistent. This means that the exit camera will only show cars leaving that have previously entered. The duration of the cars inside the car parks is sampled randomly.

We plan to simulate the gates and traffic lights with a python program that visualizes their state with a library like Tkinter or similar a library.

For the web interface, we implemented a web application using React. This allows us to simulate a real-world scenario where customers use their smartphones or a kiosk to interact with the system. The user interface was designed with Tailwind CSS to ensure it is easy to use and responsive on different screen sizes. Through this web app, users can check for available parking spots in real-time, reserve a space in advance, and pay their parking fees by simply entering their license plate number, which triggers the necessary validation checks in our backend.

B. Edge Implementation

The edge server performs all real-time processing required at the car park gates. Images from the simulated entry and exit cameras are published via NATS, and the edge server subscribes to these streams. For each received frame, the server runs a YOLOv11-based license plate detector followed by OCR to extract the plate text. The resulting detection is then processed locally without requiring immediate cloud interaction.

To maintain local state, the edge server uses a lightweight SQLite database. When a vehicle appears at the entry, the edge checks whether an active session for the same plate already exists and only creates a new record if the vehicle is not already inside. This prevents duplicated entries caused by multiple detections of the same car while it remains in front of the camera. For exit events, the edge looks up the corresponding session and completes it by adding an exit timestamp. In a full deployment this step would also include a payment verification request to the cloud; in the current implementation this behaviour is simulated.

After updating the database, the edge server publishes a control message via NATS to trigger the barrier device. This creates a complete workflow in which camera images flow through NATS to the edge, are processed and stored locally, and finally result in barrier actuation through NATS again. By keeping this logic at the edge, the system remains responsive even under cloud latency or temporary disconnection, while the cloud only needs to receive high-level events such as occupancy updates or payment checks.

C. Cloud Implementation

In the cloud layer we rely on Akka Typed as the central orchestration mechanism for all parking-lot related functionality. An embedded Akka HTTP server runs within the same typed ActorSystem and exposes a synchronous API to edge servers. Internally, the system is entirely message-driven, which gives us clear separation of concerns, robust concurrency, and predictable error handling.

At the core sits a typed ParkingLotManager actor that maintains the registry of parking lots and supervises one ParkingLot actor per lot. The manager accepts commands to register and deregister lots, to update occupancy, and to fetch either the status of a specific lot or the complete set of registered lots. Each ParkingLot actor keeps the local state for a single lot, namely current occupancy and maximum capacity, and derives the number of available spaces from those. Occupancy updates are handled as fire-and-forget messages to minimize latency, while status queries are served synchronously using Akka's ask pattern with typed timeouts to provide deterministic responses.

Payments are handled by a dedicated typed Payment actor that tracks a parking session per license plate. When a car is recorded as having entered, the actor creates or updates the session with the entry time. A payment command marks the session as paid and computes the price based on the parking duration using a simple policy with half-hour increments. At exit, the actor can be asked synchronously to verify whether the car has paid and to return the current price; once the car leaves, the session is deleted to ensure that only currently parked cars are stored.

Bookings are mediated by a typed Booking actor that acts as the cloud-side gateway to the edge servers. When a booking is created through the API, the actor publishes a JSON message containing an action field set to "book" and the license plate to a parking-lot specific NATS subject, which the corresponding edge server subscribes to. When a booking is cancelled, the actor similarly publishes an action "cancel" message. This approach intentionally keeps booking state lightweight in the cloud and delegates the effective reservation and occupancy adjustments to the edge, where the real-world constraints and device control live.

The HTTP API mirrors these responsibilities in a small set of endpoints. Parking-lot management includes an endpoint to register lots, one to deregister, a status endpoint to retrieve current occupancy and capacity, and an endpoint to list all registered lots. Occupancy updates are accepted via a simple POST call and relayed to the appropriate ParkingLot actor without awaiting a reply. The payment flow provides endpoints to record entry, to mark a license plate as paid and return the computed price, to check payment status at the exit, and to delete the session after successful departure. Bookings are exposed through a POST endpoint to create a booking and a DELETE endpoint to cancel it, both of which result in the Booking actor publishing the appropriate NATS messages to the lot-specific subject.

Communication between cloud and edge is deliberately mixed to match the semantics of each interaction. Edge-to-cloud interactions that benefit from synchronous responses—such as retrieving payment status or listing registered lots—use the HTTP API and the ask pattern under the hood. High-frequency, one-way updates—such as occupancy changes—use fire-and-forget messages for minimal overhead. Cloud-to-edge interactions for bookings use NATS publishing because reservations are asynchronous events best handled locally at the edge, and messaging scales naturally across many geographically distributed edge servers while keeping the cloud simple.

Using REST for edge-to-cloud requests provides a uniform access point for edge servers and web servers and is a natural fit for synchronous queries, while NATS for cloud-to-edge booking events decouples producers and consumers and keeps the edge authoritative for local capacity management. Finally, placing state where it belongs—lot occupancy inside ParkingLot actors, payment sessions inside the Payment actor, and booking mediation inside the Booking actor—minimizes side effects and keeps responsibilities clear.

IV. Evaluation

Stress your application to prove the correctness of your implementation, be aware of its main limitations. Explain first the experiments done (e.g., vary the number of input events), then introduce and discuss the results obtained.

V. Conclusions and future work

Summarize your solution described in this report, as well as honestly mention the current limitations and the areas that could be explored in future work.