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. With this system, customers are able 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. In addition, the web application supports location-aware nearest parking facility discovery.

The dataset we 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. 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.

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

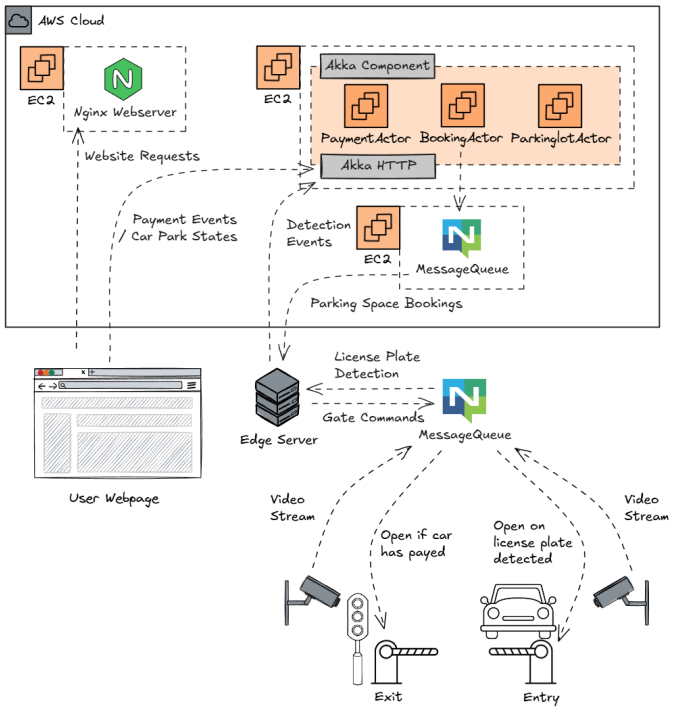

Fig. 1. The architecture diagram for this project. For visual purposes only a single car park system is illustrated.

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 that allows users to interact with the parking system. Through this application, users can see all registered parking lots, check their current availability, reserve a parking space in advance by entering their license plate number and pay. The interface was built with Tailwind CSS to provide a responsive layout that works well on both mobile devices and desktop browsers.

To help users find a suitable parking lot, the application can access the current location of the client device using the browser's geolocation feature. Each parking lot is provided by the backend together with its geographical coordinates. Based on this information, the application calculates the distance between the user and each parking lot and sorts the results so that nearby parking options are shown first. A routing button is also provided, which opens a navigation view in an external map service for the selected parking lot.

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.

Each ParkingLot actor additionally stores static metadata describing the physical location of the car park in the form of latitude and longitude. These coordinates are provided during parking-lot registration and are returned as part of the lot status via the HTTP API. This allows clients to perform distance calculations and routing without requiring external geocoding services.

For the web application, we implemented a web application using React that allows users to interact with the parking system. Through this application, users can see all registered parking lots, check their current availability, reserve a parking space in advance by entering their license plate number and pay. We host the web application on a separate EC2 instance from the Akka backend using Nginx as a web server.

IV. Limitations

While the implemented system covers the core functionality of a video license plate parking system, there are several limitations to consider.

One major limitation is the missing authentication and authorization mechanisms. Currently, there is no mechanism to verify the identity of parking lot servers when they register with the cloud backend or when they send occupancy updates. This could lead to unauthorized entities manipulating parking lot data. Implementing secure authentication, such as mutual TLS or API keys, would be necessary for a production system.

Another big limitation is the simplistic hosting setup. The Akka backend is hosted on a single EC2 instance without any load balancing or auto-scaling. This setup may not handle high traffic loads or provide sufficient fault tolerance. In a production environment, deploying the backend using container orchestration platforms like Amazon ECS along with load balancers and auto-scaling, would improve reliability and scalability.

The deployment should be better automated. Currently the setup of the cloud is done using terraform, but the web application bundle and the Akka backend Jar file need to be manually created and pushed to an S3 bucket. A more automated CI/CD pipeline using AWS CodePipeline or similar services would streamline deployments and reduce manual effort and lower the risk of human error.

V. Evaluation

A. Test Case 1: Cloud Backend Load under Realistic Edge Traffic

This test case evaluates the scalability limits of the cloud backend under a realistic workload generated by multiple independent edge servers. In a real deployment, geographically distributed parking facilities continuously send occupancy updates and sporadically trigger payment-related requests. The goal of this experiment is to determine up to which load level the cloud backend operates reliably and to identify the saturation point at which latency and error rates increase rapidly.

Test Environment: The cloud backend was deployed on a single Amazon EC2 t3.micro instance. This instance type provides one virtual CPU and 1 GB of RAM and relies on a burst-based CPU credit model. No load balancing or horizontal scaling was used in order to observe the inherent limits of a single-instance Akka-based backend.

The backend exposes a REST-based HTTP API implemented using Akka HTTP and Akka Typed actors. All parking-lot state management, payment processing and booking coordination are handled within the actor system.

Workload Model: To generate load, a custom asynchronous Python-based load generator was used. Instead of issuing synthetic request bursts, the tool simulates the behaviour of multiple independent edge servers:

- Each simulated edge server periodically sends occupancy updates to the cloud backend.
- Request arrivals follow a Poisson process, reflecting the stochastic nature of real vehicle movements.
- In addition to occupancy updates, a low but constant rate of payment sessions is generated. Each payment session consists of multiple dependent HTTP requests (enter, check, pay, exit), modelling realistic user interaction rather than isolated requests.
- A global limit on the number of in-flight HTTP requests is enforced to prevent the client from unrealistically overwhelming the server.

This composition ensures that the observed behaviour reflects realistic operational load rather than a purely synthetic stress test.

Experimental Setup: The number of simulated edge servers was fixed at ten. The offered load was increased step-wise by raising the occupancy update rate per edge server, while keeping all other parameters constant. Each load level was

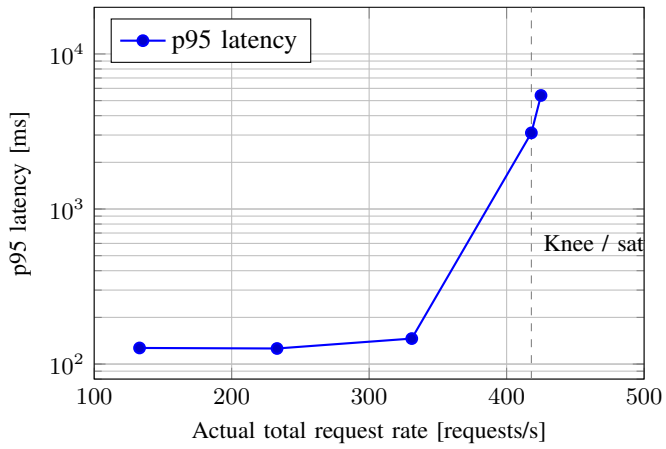

Fig. 2. Cloud backend latency under increasing edge-generated load. The system shows stable and low latency up to approximately 330 requests/s. At around 420 requests/s, tail latency increases sharply, indicating the onset of saturation. Further load leads to severe overload and queueing collapse.

measured for 90 seconds following a warm-up phase of 10 seconds.

The following metrics were recorded:

- Achieved total request throughput (requests per second)
- Success rate and number of timeouts
- End-to-end request latency, with a focus on the 95th percentile

The results show three distinct operating regions. For low to moderate load levels, the backend scales linearly and maintains a stable tail latency below 200 ms. At approximately 420 requests/s, the system reaches its saturation point, where latency increases by more than an order of magnitude and timeouts begin to occur. Beyond this point, additional load reduces the effective throughput and leads to severe degradation in both latency and success rate, which is characteristic of overload in queue-based systems.

B. Test Case 2: Web Interface Load Test (Nginx + React)

To evaluate the performance of the web interface, we first benchmarked the standalone web server that serves the compiled React frontend. This test was intentionally isolated from the rest of the system components, meaning that neither the Akka cloud backend nor any edge servers were involved. In this setup, the measured latency and throughput mainly reflect the capability of the Nginx server to deliver static frontend assets under concurrent access.

We used the `wrk` benchmarking tool with 4 threads and 100 concurrent connections over a duration of 60 seconds:

```
wrk -t4 -c100 -d60s http://3.87.5.60
```

During the test run, the server handled a total of 54,162 requests, resulting in an average throughput of 901.89 requests per second. The measured average response latency was 111 ms, while the maximum observed latency remained below 450 ms. No failed requests or timeouts were recorded, indicating stable behaviour under this load. Overall, these results suggest that the web server is not a bottleneck for moderate concurrency and can reliably serve a high number of simultaneous users with low latency.

High-load attempt. We additionally attempted a more aggressive load configuration to simulate significantly higher concurrency. Under this extreme setup, the system behaviour became unstable and requests started to time out, which indicates that the current deployment is not configured to sustain such high load without additional scaling or tuning. Since this load level exceeds the expected operating range of our prototype deployment, we primarily use the moderate load test above as the main reference point for the evaluation.

C. Test Case 3: NATS Publish Load Test for Edge Messaging

In addition to the HTTP-based cloud interface and the web frontend, we evaluated the NATS messaging layer used at the edge of the system. NATS is responsible for transporting continuous camera streams and control messages between IoT devices and edge servers and is therefore critical for real-time operation.

In this test case, we measured how the NATS server handles sustained publish load without request–reply semantics. A dedicated load test client continuously published messages to the entry and exit camera subjects of a single parking lot. Each message had a payload size of approximately 150 KB, which corresponds to compressed image frames used by the simulated cameras. Messages were sent at a rate of 5 messages per second per stream over a duration of 60 seconds.

During the experiment, a total of 299 messages were published for each stream, resulting in nearly 600 published messages overall. No send errors, timeouts or missing subscribers were observed during the test. All messages were accepted by the NATS server, indicating stable operation under continuous load.

Since the camera streams are implemented as fire-and-forget publishers, no round-trip latency was measured in this test. Instead, the focus was on message throughput and delivery reliability. The results show that the NATS server can reliably handle sustained streaming traffic at the edge without message loss or instability.

Overall, this experiment confirms that NATS is well suited for continuous data streaming between IoT devices and edge servers in the proposed system. While this test does not provide latency measurements, it demonstrates that the messaging layer remains stable under realistic camera load and complements the HTTP-based load tests performed on the cloud backend.

VI. 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.