Video License Plate Parking System

Dominik Schweigl Daniel Wenger Nikola Adzic

I. Introduction

Efficient parking management is an increasingly important challenge in urban environments, where traditional ticket-based parking systems often lead to congestion at entry and exit points, operational overhead, and poor user experience due to lost tickets and limited real-time visibility into parking availability.

This project presents a distributed, ticketless parking system based on automatic license plate recognition. Cameras installed at parking lot entry and exit points capture vehicle license plates, which serve as the identifier for access control and payment. This enables vehicles to enter and exit without physical tickets or manual interaction. Users interact with the system through a single web application, accessible both on smartphones and on on-site payment stations, which allows them to pay, reserve parking spaces, view parking space availability, and discover the nearest parking facilities based on their location.

This work focuses on the system architecture, communication patterns, and deployment considerations of the distributed parking solution. Physical devices such as cameras, gates, and traffic lights are simulated, and license plate recognition is implemented using a YOLO license plate detection model combined with the EasyOCR optical character recognition model without evaluation of computer vision performance.

The remainder of this paper is structured as follows. Section II describes the overall system architecture and its main components. Section III details the implementation of the edge, cloud, and web layers. Section IV evaluates the system under different load scenarios. Section V discusses the limitations of our system, and Section VI summarizes the paper and outlines future work.

II. System Architecture

The proposed parking system adopts a distributed edge–cloud architecture, organizing responsibilities across four distinct layers. The IoT layer handles sensing and actuation, the edge layer manages real-time processing and actuator control, the cloud layer provides centralized services for information management, booking, and payments, and the presentation layer enables user interaction through a web-based interface. This layered approach ensures low-latency parking actuator control at parking facilities while supporting centralized payment processing and a global view of parking availability. Figure 1 illustrates the system components and their interactions. For simplicity, only a single parking lot is shown, though each facility deploys its own identical edge configuration.

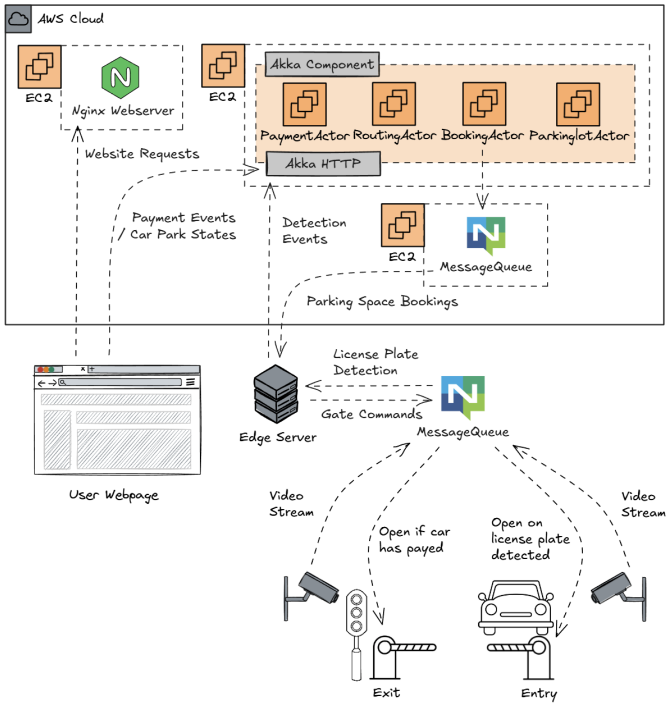

Fig. 1: The architecture diagram for this project. For clarity, only one parking lot edge is shown. Each parking lot has its own edge server, gate, camera, and message queue.

A. Edge Layer

Each parking lot is equipped with an edge server that is directly connected to the physical infrastructure of the facility. This includes two cameras positioned at the entry and exit gates, barrier gates, and traffic lights. The cameras continuously stream video data to the edge server via a local NATS message queue, which decouples video producers from consumers.

The edge server performs real-time license plate detection on incoming video frames using a YOLO license plate detection model combined with the EasyOCR optical character recognition model. When a vehicle is detected at the entry gate, the edge server registers the vehicle locally and issues a command via the local NATS message queue to open the entry barrier actuator. Local persistence using an SQLite database is maintained to ensure data robustness against temporary edge server failures.

At the exit gate, the edge server again detects the vehicle's license plate and queries the cloud backend to verify the payment status associated with the plate. If the vehicle has

paid and the allowed exit window has not expired, the edge server opens the exit gate and signals a green traffic light. Otherwise, the gate remains closed and a red signal instructs the driver to return and complete payment. In addition, the edge server sends occupancy updates to the cloud every time a vehicle enters or exits the parking lot.

B. Cloud Layer

The cloud backend is hosted on Amazon EC2 and implemented using the Akka framework. It serves as the central coordination point for all parking facilities. The cloud maintains global state such as parking lot registrations, current occupancy levels, payment status, and advances bookings to the respective edge servers. Each parking lot periodically reports its current occupancy to the cloud, allowing the backend to maintain an up-to-date view of available parking spaces across all facilities.

The Communication between edge servers and the cloud backend uses a combination of synchronous and asynchronous mechanisms. REST-based HTTP endpoints exposed via Akka HTTP are used for operations from the edge to the cloud since many operations need immediate responses, such as payment verification at exit gates. Asynchronous messaging via a NATS message queue hosted on it's own EC2 instance is used for event-driven interactions, such as propagating booking requests from the cloud to the appropriate edge server.

C. Web and User Interaction Layer

User interaction with the system is provided through a web application implemented using React and hosted on a separate EC2 instance using Nginx as a web server. The same web application is accessible both from users' personal smartphones and from on-site payment stations within parking facilities. Through this interface, users can view available parking lots, reserve parking spaces in advance using their license plate number, complete payments, and discover nearby parking facilities based on their current location.

III. Implementation details

This section presents the implementation of the system across its various layers, including the IoT components, edge server, cloud backend, and web interface. For each layer, the chosen design and implementation decisions are explained and justified.

A. IoT Implementation

The IoT components of the parking lot system are implemented in Python and deployed using Docker containers with `docker-compose`, ensuring that the system is independent of the underlying execution infrastructure. Both the camera simulations and actuator interfaces run inside these containers.

a) Camera: The entry and exit points of the parking lot are simulated by two camera streams, which alternately transmit images from a car license plate detection dataset and a reference empty street scene. 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. We chose a subset of the 50 most clearly visible license plates for our simulation.

Images are published to an edge server via a NATS message queue via the camera simulation python script to two separate message queue subjects. It is made sure that the data sent by the entry and exit camera streams 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.

Cars enter the lot at a steady pace, with one car added every two display cycles of 3 seconds (`FRAME_DELAY`), yielding an average entry rate of

$$R_{\text{entry}} = \frac{1}{2 \cdot \text{FRAME\_DELAY}} \approx 0.166 \text{ cars/second.}$$

Each car has a probability of leaving the lot in any given frame of

$$P_{\text{exit}} = \frac{1}{4 \cdot \text{FRAME\_DELAY}} \approx 0.0833 \ (8.33\%),$$

corresponding to an average exit rate of

$$R_{\text{exit}} = \frac{P_{\text{exit}}}{\text{FRAME\_DELAY}} \approx 0.028 \text{ cars/second.}$$

This setup results in a parking lot that gradually fills over time, where entries outpacing exits.

b) Parking Lot Gates and Traffic Light Interface: The parking lot gates and traffic lights are simulated via a web-based interface implemented using `FastAPI`. The framework was selected due to its support for rapid and intuitive development. In addition, FastAPI provides native support for asynchronous communication and WebSockets, which makes it well suited for real-time IoT applications.

The interface provides a live view of the current state of all actuators for each parking lot individually. It receives continuous state updates from the edge server through the local NATS message broker, ensuring that changes in actuator states, such as barrier positions or traffic light signals, are reflected on the web dashboard. Furthermore, the interface displays the camera streams from the entry and exit points, which allows to visually monitor the camera streams and correlate actuator behavior with observed traffic conditions.

B. Edge Implementation

The edge server is responsible for all real-time processing at the parking lot gates. For each incoming camera frame, it executes a YOLOv11-based license plate detection pipeline followed by optical character recognition (OCR) to extract the license plate text. All detections are processed locally at the edge without requiring immediate interaction with the cloud. This design keeps network traffic to the cloud low and ensures low-latency actuator control and allows the system to remain operational during temporary cloud disconnections, though with some limitations at the exit gate due to synchronous payment verification at the cloud. The edge server is implemented in Python, as the employed machine learning models and

inference pipelines are exclusively available through Python-based frameworks.

The edge deployment follows the same container-based approach as the IoT layer. Both the edge server and the local NATS message broker are executed within Docker containers, and together with the IoT components they are deployed as a single edge application using `docker-compose`. This setup enables consistent and reproducible deployment across different parking facilities, simplifies configuration management, and allows the complete edge stack—including processing logic and messaging infrastructure—to be started, stopped, and updated as a single unit.

To maintain local state, the edge server uses a lightweight SQLite database. SQLite was selected for its low operational overhead, zero-configuration deployment, and suitability for resource-constrained edge environments. As an embedded database, it runs within the application process and requires no separate database service, providing sufficient performance for handling several hundred concurrent parking sessions per lot while keeping the system simple.

When a vehicle appears at the entry point, the edge server checks whether an active session for the same license plate already exists and only creates a new record if the vehicle is not already inside the parking lot. This prevents duplicated entries caused by repeated detections of the same vehicle while it remains in front of the entry camera. For exit events, the edge queries the cloud service to verify the payment status, retrieves the corresponding local session from SQLite, and completes it by recording the exit timestamp.

Communication between the camera streams, actuator controllers, and the edge server is handled using NATS as the message broker. NATS was selected for its lightweight architecture, low-latency message delivery, and minimal operational overhead, making it well suited for deployment on resource-constrained edge servers. In contrast to heavier brokers such as Apache Kafka or RabbitMQ, which emphasize message persistence and complex broker management, NATS is optimized for real-time, transient messaging patterns while the throughput and provides ample throughput to reliably handle the modest message rates in the low tens generated by the camera streams and actuator commands.

Using NATS creates a clean decoupling between data producers (camera streams) and consumers (edge processing and actuator controllers). The camera streams publish data using NATS core publish–subscribe semantics, where messages are delivered on a best-effort basis only to currently active subscribers and are not persistently stored. This behavior aligns with the nature of camera frames, which are only relevant at the time of generation and have no value if processed later. By avoiding message persistence, the system reduces memory and storage overhead while ensuring timely delivery of live camera data. Actuator commands and state updates similarly benefit from this low-latency, ephemeral communication model, enabling responsive and reliable gate control at the edge.

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 ActorSystem and exposes a synchronous HTTP REST API to edge servers and the web application. This is a standard protocol, which makes it easy to provide uniform access for edge servers and web clients alike. Internally, the system is entirely message-driven, which provides clear separation of concerns and concurrency.

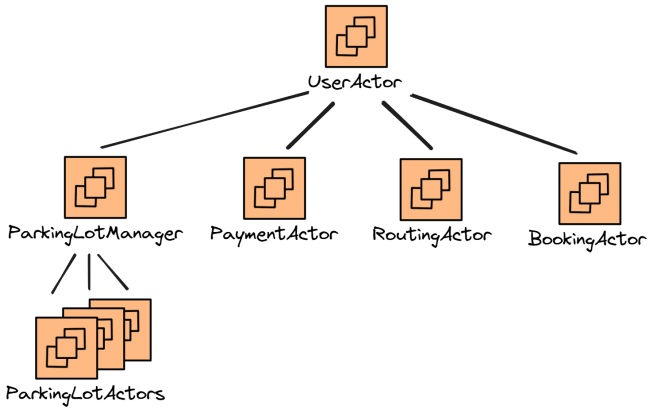

Fig. 2: Structure of the application-level Akka actor hierarchy in the cloud backend. The user actor is shown as the root of the application-level actors, Akka system and infrastructure actors are omitted for clarity.

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. Parking lot edge occupancy updates are handled as fire-and-forget messages to minimize latency, while status queries are served using Akka's ask pattern.

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, which reduces system load.

Bookings are mediated by a 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 to a parking-lot specific NATS subject, which the corresponding edge server subscribes to. This approach decouples the cloud from the physical deployment of edge servers, allowing bookings to be routed while the edge network locations transparent to the cloud. When a booking is cancelled, the actor similarly pub-

lishes an action "cancel" message. By using an asynchronous message queue, the system remains resilient to temporary connectivity issues. This approach intentionally keeps booking state lightweight in the cloud and delegates the effective reservation and occupancy enforcement to the edge, where the real-world constraints such as maximum occupancy and gate control are managed. The edge server is therefore responsible for enforcing bookings locally and rejecting incoming vehicles when all available parking spaces are already reserved.

For the computation of the closest parking lots based on user location, the cloud backend defines a Routing Actor that calculates the Haversine distance between the user's coordinates and each parking lot's stored geographical location. The actor then sorts the parking lots based on this distance and returns the sorted list to the web client.

D. Web Interface

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. React was chosen for its component-based architecture, which simplifies the development and maintenance of dynamic user interfaces. The interface was styled using Tailwind CSS, whose utility-first approach enables rapid creation of responsive and consistent layouts with minimal custom styling.

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 and the distance to the corresponding parking lot. The frontend application 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.

IV. Evaluation

A. Evaluation Setup

The cloud components of the system were deployed on Amazon Web Services (AWS) using Terraform to ensure reproducibility and consistency across experiments. The setup consists of three EC2 instances hosting the backend application, message broker, and web frontend, respectively. All instances use the `t3.micro` instance type, providing 2 virtual CPUs and 1 GB of RAM, which is sufficient for evaluating the system under moderate load. The instances run Amazon Linux 2023 (x86_64).

The backend service is deployed as a Java-based Akka application running on Amazon Corretto JDK 17, while the NATS message broker is hosted in a Docker container to facilitate lightweight and isolated operation. The web frontend is served via Nginx as a static application. Network access is controlled using a dedicated security group that exposes only the required ports for HTTP communication, backend APIs, NATS messaging, and SSH access.

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

This test examines the cloud backend's scalability under a realistic workload from multiple independent edge servers. In deployment, distributed parking facilities continuously report occupancy and occasionally send payment requests. The experiment identifies the load at which the backend remains reliable and the point where latency and errors rise sharply.

Workload Model: To generate this load, a custom asynchronous Python-based generator was used to simulate the behavior of multiple edge servers. Each of the 10 servers periodically sends occupancy updates at rates of 10-70 updates per second per edge following a Poisson process, reflecting the stochastic nature of vehicle arrivals. Separately, payment sessions are issued steadily at 6 sessions per second, with each session performing multiple dependent HTTP requests (enter, check, pay, exit) to model realistic user interactions. This setup ensures that the resulting measurements accurately reflect the backend's performance under operational conditions.

Experimental Setup: The experiments used a fixed set of ten simulated edge servers. Offered load was increased incrementally by raising the occupancy update rate per server, while all other parameters remained constant. Each load level was measured for 90 seconds following a 10-second warm-up period. During these measurements, we recorded total request throughput, success rate and timeout occurrences, and the 95th percentile of request latency.

The results reveal three operating regimes (Figures 3-5). At low to moderate loads (up to 300 requests/s), the system scales nearly linearly: throughput matches the target, success rate stays at 100%, and p95 latency remains below 200 ms. Near 400 requests/s, the backend saturates: tail latency spikes, throughput plateaus around 420 requests/s, and success rate drops due to timeouts. Beyond this, the system enters sustained overload: effective throughput collapses despite higher target request rates, p95 latency stays high, and success rate deteriorates sharply to 50%.

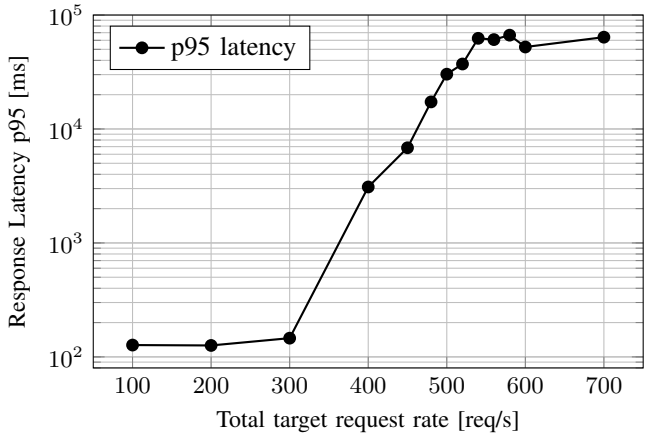

Fig. 3: Cloud backend p95 latency as a function of the total target request rate of all 10 edge servers. Latency remains low and stable up to approximately 300 requests/s. Beyond 400 requests/s, tail latency increases sharply.

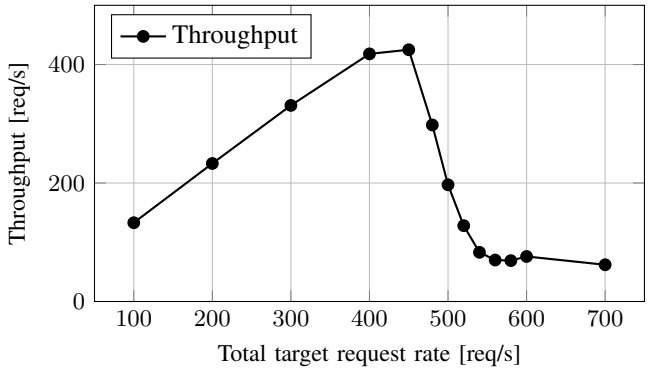

Fig. 4: Achieved backend throughput versus total target request rate. Throughput scales nearly linearly with load up to approximately 400 requests/s. Beyond this point, achieved throughput decreases sharply despite increasing offered load.

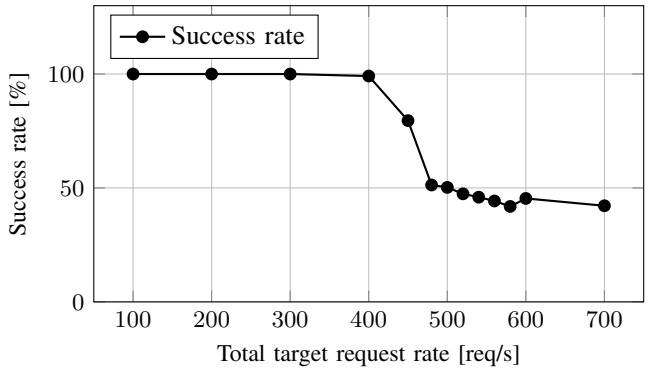

Fig. 5: Request success rate versus target request rate. The system maintains near-perfect reliability up to approximately 400 requests/s. Beyond this point, success rate diminishes to approximately 50% as timeouts increase.

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

The goal of this test was to evaluate the performance of the web interface, specifically the ability of the Nginx server to serve a compiled React frontend under concurrent access. 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 the first test shown in Figure 6, the web server was subjected to a step-wise load test using `wrk2`. A fixed number of threads and connections was maintained throughout the experiment, while the target request rate (`-R`) was increased incrementally. The test targeted the root endpoint (`/`), thereby isolating the server's ability to serve static frontend content under increasing per-user request rates.

Figure 6 shows the achieved throughput as a function of the configured target rate, with the dashed line representing ideal linear scaling. At lower request rates, the server closely tracks the offered load, indicating stable operation. As the target rate increases, throughput increasingly deviates from linear behavior and becomes unstable, with a pronounced drop observed beyond approximately 30 requests per second. This behavior indicates server saturation and the loss of efficient per-user request handling above this threshold.

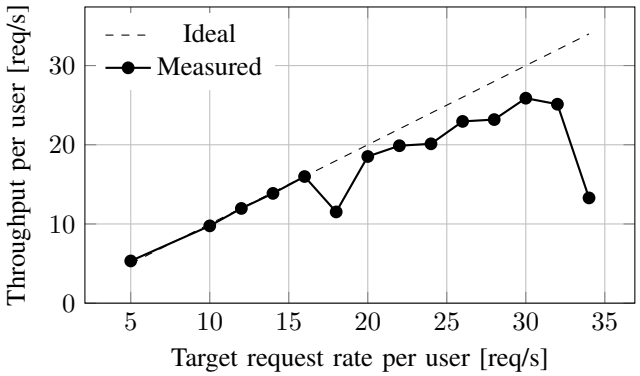

Fig. 6: Step-wise `wrk2` load test of the Nginx web interface serving a compiled React frontend. The figure compares the configured per-user request rate with the achieved throughput for the root endpoint (`/`). The dashed line indicates ideal linear scaling.

To evaluate the server's performance under a common web server request pattern—a growing number of concurrent users, each generating a small number of requests—a second test was conducted using `hey`. This test measures latency and throughput as a function of the total target request rate while maintaining a constant rate of 5 requests per second per concurrent user. The test gradually increases the total request rate by adding more concurrent users, with each step lasting 60 seconds and a 30-second pause between steps to allow server recovery. The system employed 6 cores and 50,000 open ports to handle concurrent connections.

Figures 7 and 8 present the results. Latency percentiles (p10, p50, p95) remain low and stable up to roughly 4000 requests/s, indicating efficient handling of the load. Beyond this threshold, all percentiles rise sharply, signaling saturation and eventual overload. Achieved throughput initially scales nearly linearly with the offered load but drops abruptly past 4000 requests/s, eventually collapsing near 5000 requests/s as the server becomes severely overloaded and the majority of requests timeout.

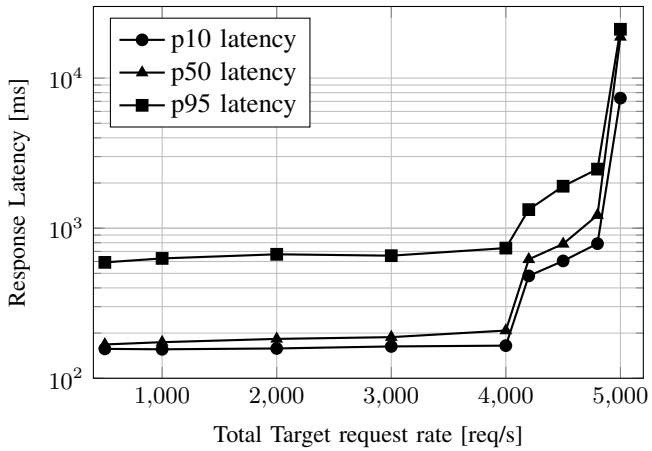

Fig. 7: Nginx server response latency across different percentiles as a function of offered load. Latency remains low and stable up to roughly 4000 requests/s. Above this threshold, latency rises sharply for all percentiles.

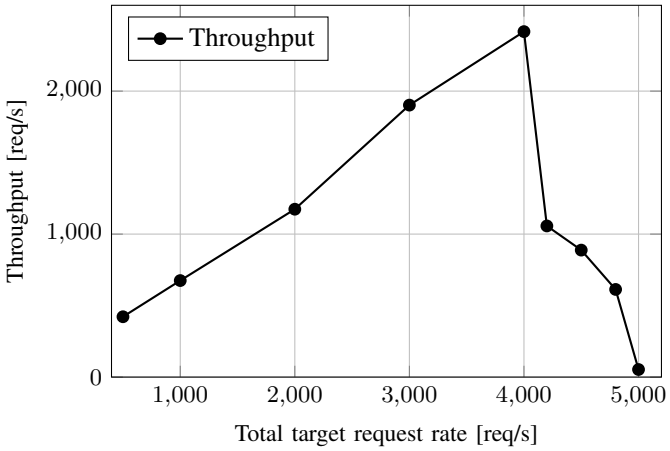

Fig. 8: Measured nginx throughput as a function of offered load. Throughput scales nearly linearly with offered load up to 4000 requests/s. Beyond this point, throughput drops sharply.

V. Limitations

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

a) Payment event latency optimization: To reduce latency at the exit gate, payment events could be propagated to the edge server proactively rather than queried on demand. In the current design, the edge server performs a synchronous request to the cloud backend when a vehicle approaches the exit, introducing avoidable latency. By pushing payment status updates to the edge immediately after a transaction is completed, the edge system could locally cache this information and eliminate the need for synchronous cloud queries at exit time. This approach would result in faster gate responses and improved robustness during transient cloud disconnections.

This limitation is further emphasized by the fact that the current payment system is a mock implementation without integration with real payment gateways. In a production environment, secure payment providers such as Stripe or PayPal would be required to process actual financial transactions, inevitably introducing additional latency during payment verification. By proactively propagating payment status updates to the edge server, this added delay could be mitigated.

b) Security and authentication: Another major limitation of the current system is the absence of authentication and authorization mechanisms. At present, the cloud backend does not verify the identity of parking lot servers during registration or when receiving occupancy updates, which could allow unauthorized entities to inject or manipulate parking lot data. A production–ready deployment would therefore require the introduction of secure authentication mechanisms, such as mutual TLS or API key based access control, to ensure data integrity and system security.

c) Cloud Scalability and Availability: A further limitation of the current system is the simplicity of the cloud deployment. The cloud components—the Nginx web server, the Akka-based backend, and the NATS message broker—are each hosted on a single EC2 instance, without load balancing or automatic scaling. While sufficient for prototyping and evaluation, this architecture may not sustain high traffic volumes and offers limited fault tolerance, as each component constitutes a single point of failure. Reliability and scalability could be significantly improved by deploying the cloud services using container orchestration platforms such as Amazon ECS, in combination with load balancers and auto-scaling mechanisms.

Furthermore, because all cloud-to-edge communication is mediated by the message broker and involves temporary buffering of messages, the system could benefit from brokers such as Apache Kafka, which provide persistent storage, replication, and message replay in the event of consumer or broker failures. For the scale of the current system, however, NATS remains a suitable choice. To prevent loss of critical messages, such as booking updates, when an edge server is temporarily disconnected, NATS JetStream could be employed to enable message persistence and reliable delivery.

d) Booking Race Conditions: The current booking implementation does not prevent race conditions when web users and drivers at the entry gate attempt to access the last available parking space simultaneously. For example, a user may reserve the final space through the web interface at the same time a car arrives at the entry gate without a prior booking. If the car enters and the booking request arrives at the cloud before the edge server updates its occupancy, the booking request is still processed successfully by the cloud and forwarded to the edge, resulting in overbooking. To fully prevent such race conditions, the booking process would require explicit confirmation from the edge server before finalizing a reservation.

e) Deployment Automation: The deployment process could benefit from greater automation. While the cloud infrastructure is provisioned using Terraform, the web application bundle and the Akka backend JAR currently need to be built manually and locally, which are then automatically uploaded during terraform deployment using file provisioners. This manual step introduces additional effort and increases the potential for human error, which can affect reproducibility and slow down iterative development. Implementing a fully automated CI/CD pipeline, for example using, GitHub Actions, or GitLab CI, would streamline the build, test, and deployment processes. Such a pipeline could automatically compile the backend and frontend artifacts, run tests, and deploy them to the cloud infrastructure with minimal human intervention, improved reliability, and reduce deployment time.

f) Distance Calculation: Nearby parking lots are currently identified using the Haversine formula, which computes straight-line distances. For more accurate routing, integration with a mapping service that provides real-world driving distances and directions would be advantageous, though it may introduce additional latency.

g) License Plate Detection Model: In a production system, the detection model would ideally be fine-tuned to the specific environment and camera angles of the parking lot entrances. For this project, we employ a pre-trained YOLOv11 license plate detection model combined with EasyOCR for

optical character recognition. In our scenario, this approach is sufficient to demonstrate the system's functionality, as each car is represented by a single, unchanging image, resulting in consistently stable detection outcomes.

VI. Conclusions and future work

This paper presented a distributed, ticketless parking system based on automatic license plate recognition, combining edge computing, cloud-based coordination, and a web-based user interface. By processing camera streams locally at the edge and delegating global state management, payments, and bookings to a cloud backend, the system achieves low-latency gate control while maintaining a centralized view of parking availability. The actor-based cloud architecture, NATS messaging, and containerized deployment demonstrate a scalable and modular design suitable for coordinating multiple parking facilities. Experimental evaluation shows that the system performs reliably under moderate load, with stable response times for cloud APIs, web access, and edge messaging.

Several limitations remain and point toward future work. Payment verification currently introduces avoidable latency at the exit gate and could be optimized through proactive propagation of payment events to the edge. Security mechanisms such as authentication and authorization are not yet implemented and would be essential in a production deployment. Cloud scalability and fault tolerance are limited by single-instance components and could be improved through load balancing, auto-scaling, and persistent messaging solutions. Additional challenges include booking race conditions, partial deployment automation, simplified distance calculations, and the use of a pre-trained license plate detection model without environment-specific tuning. Addressing these aspects would further improve robustness, scalability, and real-world applicability of the proposed system.