Smart Parking

(v. 0.1.0)

Valerio Giannini
valerio.giannini@studio.unibo.it

Davide Di Marco
davide.dimarco5@studio.unibo.it

November 23, 2025

Contents

1 Domain Driven Design **3**
1.1 Goals, Motivation and Work Plan . . . 3

2 Requirements Analisys **4**
2.1 Domain Definition . . . 4
2.2 Subdomains Definitions . . . 4
2.3 Ubiquitous Language . . . 5
2.4 System Requirements . . . 7
2.4.1 Functional Requirements . . . 7
2.5 Use Case . . . 8
2.6 User Story . . . 10
2.7 Event Storming . . . 12
2.8 Quality Attributes Scenarios . . . 13
2.9 Story Telling . . . 14
2.9.1 Customer Storytelling . . . 14
2.9.2 Admin Storytelling . . . 16
2.10 Domain Model . . . 18
2.10.1 Entities and Value Objects . . . 18
2.10.2 Domain events . . . 19
2.11 Bounded Context . . . 20
2.12 Context Map . . . 21

3 Domain Driven Design to Microservices **23**
3.1 System operations . . . 23

3.2 Subdomains to microservices . . . 24
3.2.1 Park Management Service . . . 24
3.2.2 Authentication Service . . . 25
3.2.3 Analytics Service . . . 25
3.2.4 Embedded Services . . . 25
3.2.5 Payment Management . . . 25
3.2.6 Ticket Management . . . 26
3.3 System Operation Identification . . . 27
3.4 API Interface definition and identification . . . 28
3.4.1 Embedded-Service . . . 28
3.4.2 Parking-Service . . . 29
3.4.3 Analytics-Service . . . 30
3.4.4 Payment-Service . . . 30
3.4.5 Ticketing-Service . . . 31
3.4.6 Authentication-Service . . . 32

4 System Architecture and Microservices Architecture 33
4.1 Microservice Architectural Patterns Adopted . . . 34
4.1.1 Decomposition by Subdomain . . . 35
4.1.2 Communication Patterns . . . 36
4.1.3 Resilience and Fault Tolerance . . . 37
4.1.4 Querying Data Patterns . . . 38
4.1.5 Deployment Pattern . . . 39
4.1.6 Observability Patterns . . . 40
4.1.7 Cross-Cutting Concerns . . . 42
4.1.8 Security Pattern . . . 43
4.2 Event-Driven Architecture for Reactive Systems . . . 44
4.2.1 MQTT-Based Ingestion Layer . . . 44
4.3 Internal Event Bus Architecture . . . 45
4.3.1 System Advantages . . . 45

5 Admin Dashboard - Frontend 47
5.1 Dashboard . . . 47
5.2 Analytics . . . 48
5.3 Tickets Viewer . . . 49

6 Test & Validation 50
6.1 JUnit Test & Integration Test . . . 50
6.2 Component Test . . . 51
6.3 End to End (e2e) Test . . . 52
6.4 Embedded Simulator (Sensor-Communication Validation) . . . 54
6.5 CI/CD Pipeline . . . 55

7 Conclusions 57

1 Domain Driven Design

1.1 Goals, Motivation and Work Plan

The idea for this project originated with the intention of designing and implementing an innovative *intelligent parking* system capable of integrating embedded technologies with a microservices architecture. The aim is to manage parking space availability, perform automated license plate recognition, and handle virtual payments, while providing administrators with real-time monitoring capabilities.

We identified this topic as a concrete opportunity to apply the concepts studied during our academic course and at the same time to collaborate with a real company operating in this domain. In particular, we visited the headquarters of ATR in Cesena, the company responsible for managing the city's parking facilities[1]. In ATR we got the possibility to understand the domain and how the parking is managed by the company. In this way we facilitate the understading of the domain having always expert supporting us in defining the requisites, goals and quality attributes.
We have divided the project into multiple steps:

1. **Meeting 1, Requirements Analysis:** During the first on-site meeting with ATR, we gathered domain-specific information about how parking facilities are managed in the city of Cesena. This session allowed us to understand the operational workflow, identify key challenges, and collect the initial set of functional and non-functional requirements for our system.

2. **Domain-Driven Design Phase:** Based on the requirements collected, we applied Domain-Driven Design principles to model the domain, identify subdomains and define bounded contexts, ensuring a clear separation of concerns for the upcoming microservices implementation.

3. **Meeting 2, Validation with ATR:** In the second meeting with ATR, we validated our domain model and architectural choices with domain experts, refining certain processes and aligning our design.

4. **Transition from DDD to Microservices:** We mapped each bounded context into a set of independent microservices, defining their responsibilities, communication protocols, and data storage strategies, ensuring scalability and maintainability.

5. **Implementation Phase:** We proceeded with the development of the microservices, integration of the embedded components (ESP32-CAM for license plate recognition and NFC module for payments), database configuration, API development, and containerization using Docker, followed by testing and deployment.

[1] Sito web di ATR: `https://www.atr.fc.it/`

2 Requirements Analisys

In order to better understand the requirements of the system, we asked Stefano, a domain expert from ATR, to describe to us in detail the current parking management process. Through his storytelling, we were able to reconstruct the real operational flow: from the moment a vehicle approaches the entry gate, to the ticketing or license plate recognition phase, the monitoring of occupied and free spaces, up to the payment and exit of the vehicle. The expert's contribution was fundamental to avoid ambiguities.

2.1 Domain Definition

The core domain of this project is the management of a parking facilities. Including all the aspects needed to manage the parking. Because of the management of the park are quite complex we decide to split the domain in more subdomains as shown in the following section.

2.2 Subdomains Definitions

From the requirements analisys we identified the following subdomains inside the core domain of Automated Parking:

- **Supporting — Embedded**: this subdomain includes the interaction with physical devices installed in the parking area. In particular, it manages the sensors (cameras, barriers and IoT devices) and the embedded logic responsible for vehicle plate detection and real-time communication with the backend.
- **Supporting — Ticketing**: this subdomain handles the process of ticket creation and validation. It covers the association between the detected plate number and the parking session, issuing entry/exit tickets (virtual), and ensuring the consistency of the parking workflow.
- **Supporting — Payment**: this subdomain manages all the aspects related to payment transactions. It includes tariff calculation, payment intent creation, support for payment methods and receipt generation after a successful transaction.
- **Supporting — Parking Management**: this subdomain supervises the overall orchestration of the parking facility. It verifies availability.
- **Generic — Analytics**: this subdomain provides data collection and processing functionalities. It offers occupancy rate, average dwell time, revenue tracking. This allows administrators to monitor system performance and optimize operations.
- **Generic — Authentication**: this subdomain manages user identity and access control. It provides mechanisms for microservice authentication, API toekn, and role-based access control to ensure system security.

2.3 Ubiquitous Language

In this domain, some key terms have specific meanings that are shared across all team members, stakeholders, and technical documentation. This language ensures that communication between developers, domain experts, and stakeholders is coherent and consistent throughout the project lifecycle. The following table serves as a reference and can be progressively updated as the project evolves.

Term	**Definition**	**Bounded Context**
Parking Slot	A physical parking space delimited by lines identified by a unique ID within the parking facility.	ParkManagement
Occupancy	The current status of a parking space, which can be *Free* or *Occupied*.	ParkManagement
Session	The lifecycle of a vehicle's stay in the parking lot, starting at entry and ending at exit, including timestamps and calculated tariff and fee.	ParkManagement
Tariff	The set of rules used to compute the parking fee, based on time of stay or other pricing policies.	Payment
Admin or Administrator	The administrator of the parking facilies.	Embedded
Parking Fee	The monetary amount applied according to the tariff rules.	Payment
License Plate / Plate	The alphanumeric identifier of a vehicle, detected manually or via Plate Detection.	Embedded / Analytics / Ticketing
Virtual Ticket	The ticket dispensed every time a license plate is detected that contains all needed information for park management.	Payment / Ticketing
Payment	A payment request linked to a parking session, which can be *PENDING*, *PAID*.	Payment
Entrance Detection Event	An event emitted when the Automatic License Plate Recognition system detects and processes a plate at entry.	Embedded / Analytics
Exit Detection Event	An event emitted when the Automatic License Plate Recognition system detects and processes a plate at exit.	Embedded / Analytics
NFC Tap	A payment interaction performed by tapping a contactless card or device at the cashout.	Payment / Embedded
Admin Action	A manual intervention performed by an administrator to modify the state of a session, payment or barrier problem. Usually to resolve an error or exceptional case.	ParkManagement / Payment
Automatic Checkout	The physical Automatic Checkout placed at the pedestrians Entrance used for the payment.	Embedded / ParkManagement / Payment
Barrier	The physical barrier at the entry of the parking facilities.	Embedded / ParkManagement / Payment
Help Button	The physical button present at the entry and exit of the parking to be pushed by the user in case of emergency or help.	Embedded / ParkManagement / Payment

Coil	The inductive loop embedded in the ground at the entry and exit of the parking. It detects the presence of a vehicle and triggers the license plate recognition and barrier management process.	Embedded

Table 1: Ubiquitous Language for the Smart Parking Domain with Bounded Context mapping

2.4 System Requirements

In this section are explained the system requirements extracted from a first interview of the expert.

2.4.1 Functional Requirements

- The system shall detect vehicle license plates at both entry and exit points using camera detection.
- The System must retrieve a Virtual Ticket every time a license plate is detected at entry.
- The system shall calculate the parking fee based on time of stay using the tariff configuration.
- The system shall enable users to pay using NFC payment.
- The system shall open the entry and exit barriers after successful identification (at entry) or payment (at exit).
- The system shall allow administrators to access a web console or dashboard for monitoring, reporting, and configuration of the parking facilities.
- The system shall support real-time updates of parking occupancy status and information about the overall system.
- The system shall allow the admin to override, edit, or correct parking configuration (tariff and fee change) manually by a a web-based administration console.
- The system shall have an help button at entry and exit.

2.5 Use Case

In this section are differentiate the different use cases of the two main user actors in the project: customer and admin.

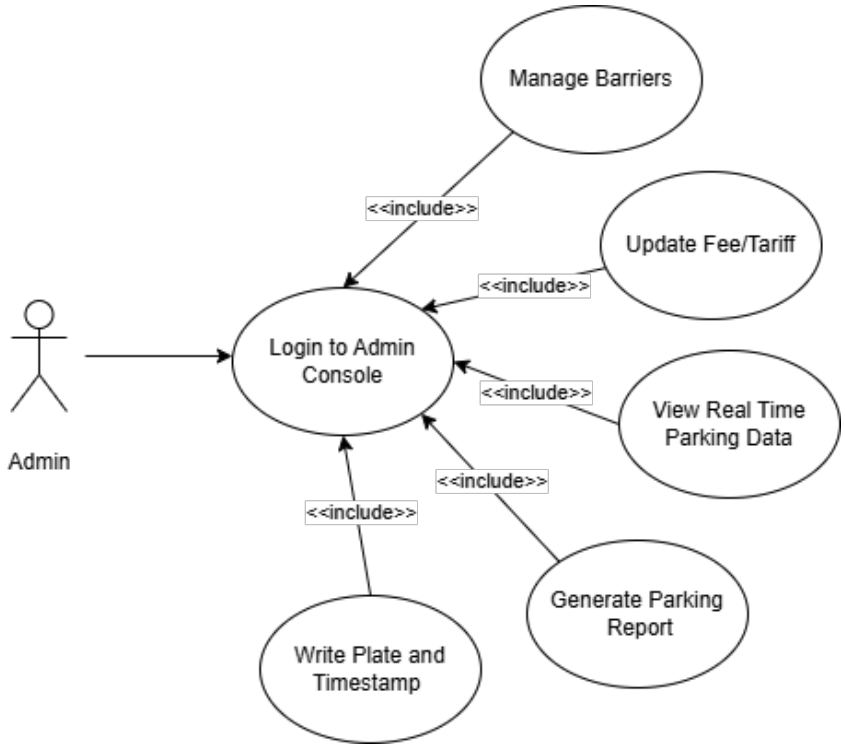

Figure 1: Use Case Diagram of Administrator of the system

Use Case	**Description**
Login to Admin Console	Allows the administrator to access the management interface. Required for all other admin actions.
Manage Barriers	Enables the administrator to open or block physical barriers manually or remotely in case of failures or overrides.
Update Fee/Tariff	Allows the administrator to update pricing rules and tariff logic used to calculate parking fees.
View Real Time Parking Data	Displays the current status of parking slots, usage, and occupancy in real time.
Generate Parking Report	Allows generation of analytical reports including revenue and daily metrics.
Write Plate	The administrator can manually enter a plate in case of Detection failure and when is called by the help button.

Table 2: Admin Use Case Descriptions

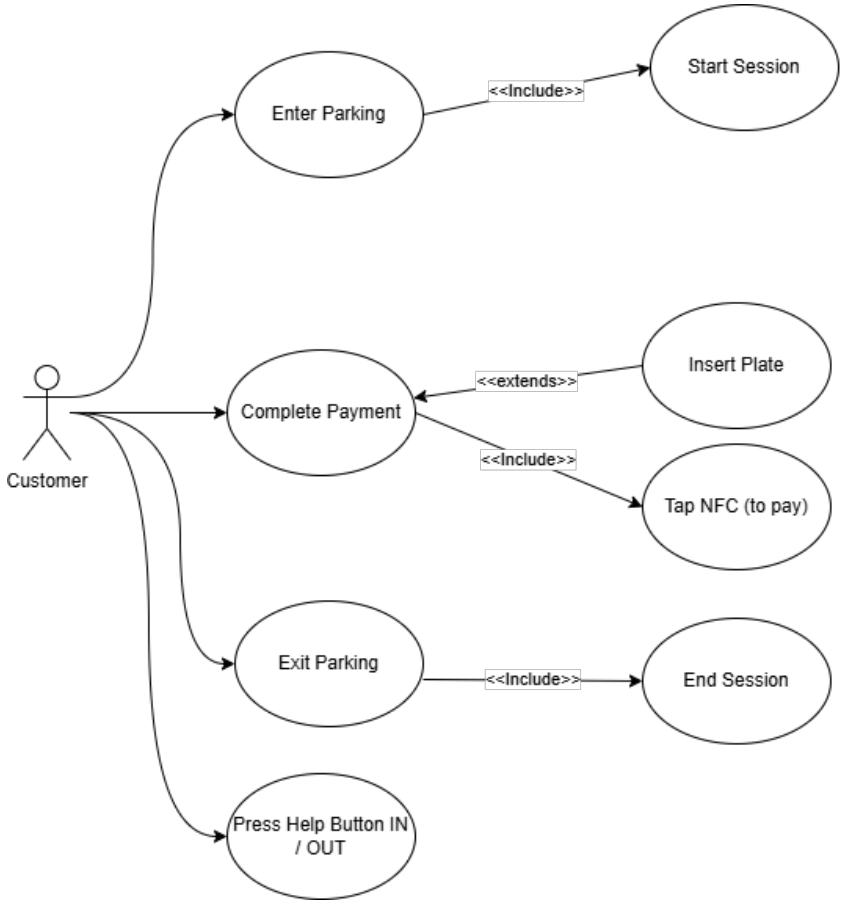

Figure 2: Use Case Diagram of customer

Use Case	Description
Enter Parking	The user starts the parking process being recognized via Entry Camera. This triggers the opening of the entry barrier and creates a session.
Start Session	A session is automatically initiated by the system when a plate is detected.
Complete Payment	The user completes payment via NFC at a payment station writing his license plate.
Tap NFC (to pay)	The user pays by tapping a contactless card or device at the payment terminal.
Exit Parking	The user exits the parking area after successful payment. The barrier opens after the system verifies the session and payment.
End Session	The system ends the parking session after payment and exit.
Press Help Button IN-/OUT	The user can request assistance by pressing a help button placed at entry or exit terminals.

Table 3: Customer Use Case Descriptions

2.6 User Story

This user stories can be grouped by the Given term: it is important to specify what of the stories describe a situation that involve one of the Actors.

User Story	**Condition**	**Description**
US1	**Given:**	An administrator logged into the system
	When:	They modify tariff plans
	Then:	The changes are saved and applied
US2	**Given:**	An emergency situation
	When:	The admin needs to intervene
	Then:	The system allows barrier control
US4	**Given:**	A vehicle approaching the entrance
	When:	It triggers the induction coil
	Then:	The camera activates
US5	**Given:**	A vehicle at the entrance
	When:	The camera is activated
	Then:	The system reads the license plate
US10	**Given:**	A vehicle ready to exit
	When:	Payment verification is required
	Then:	The system confirms payment status
US12	**Given:**	A customer at the payment terminal
	When:	They enter their license plate
	Then:	The system processes the payment
US16	**Given:**	Parking data collected over time
	When:	Analytics are requested
	Then:	The system calculates the daily summary/report

Table 4: User Stories by Actors

2.7 Event Storming

In this section we show the event storming done used for initially track the event. In Figure 1 are shown:

- Events (Orange): Are all the relevant situations that changes the system Status.
- Actions (Blue): All the action triggered by the Actors
- Actors (Yellow): Customer or Car and Admin
- Critical Poit (Pink): All the situation that could fail.
- Automatic Action (Purple): all the action triggered and made by the system

The event storming is about 2 phases: At the beginning the team work to find all the events, actors, actions, critical point and automatic action; then the focus was on create a flow of events that connect all of them. The scheme was made in 2 level, at the top we find the perfect flow, the process without errors, while below the line there are the alternative path followed when the intervention of the Admin are needed.

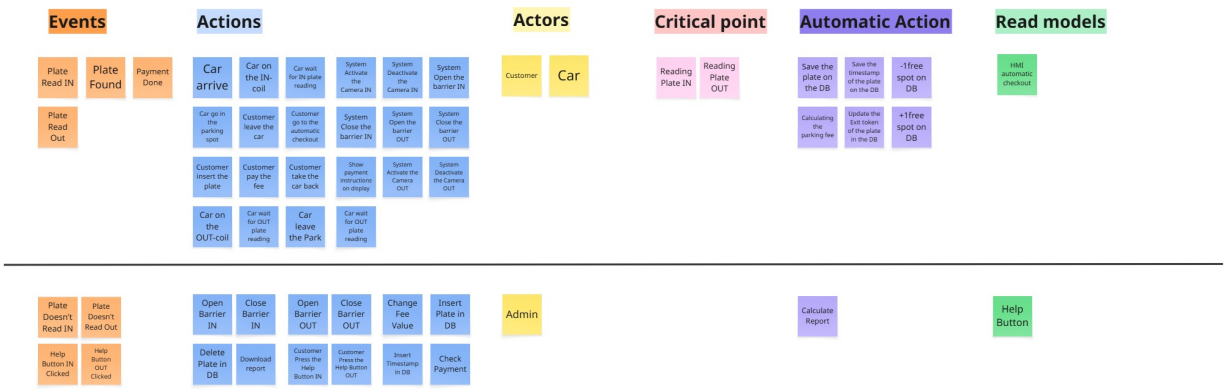

Figure 3: Sticky Notes Whiteboard for event Storming

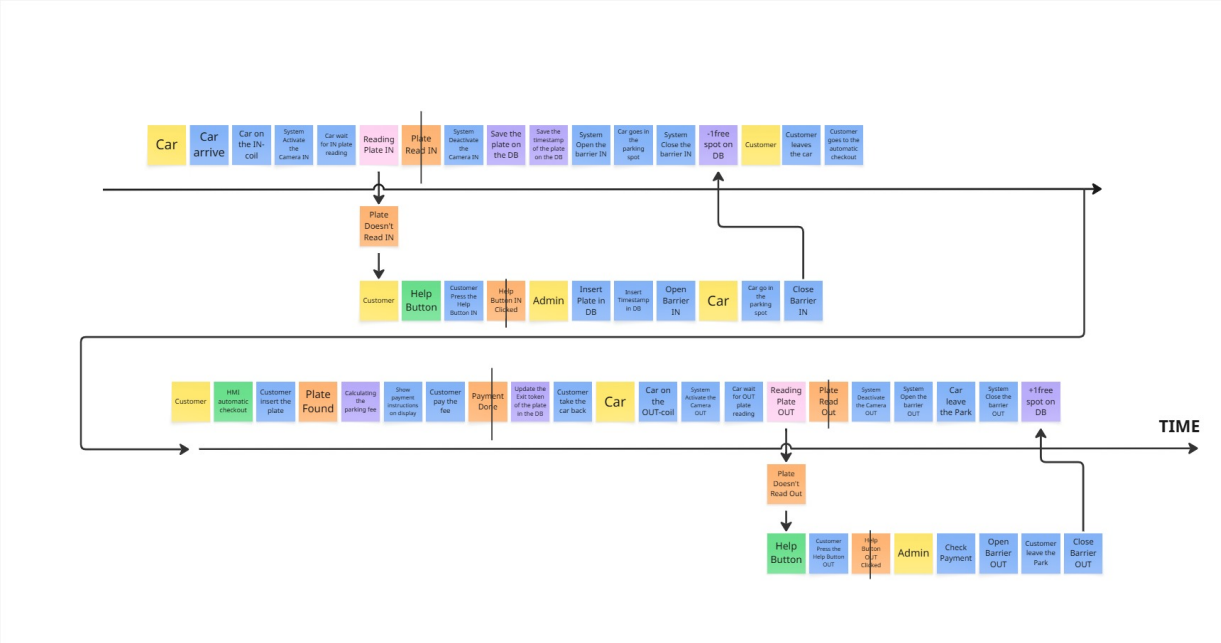

Figure 4: Event Storming Flow

2.8 Quality Attributes Scenarios

Explanation in Smart Parking context:

- **Extensibility:** If the property wants to introduce more functionalities, a new microservice can be deployed without touching other microservices.
- **Scalability (functional):** If in the future we add a Reservation Service (users reserve a parking spot via app), this is integrated by adding a new microservice without breaking existing ones.
- **Modifiability:** Updating the logic or architecture of the system doesn't impact the rest of the system in production environment because of isolated deployment.
- **Testability:** The embedded service can be tested with simulated input. The system can be tested with automated testing.
- **Interoperability:** the system is capable of interacting with other system.
- **Observability:** An admin can see logs of every car entry, metrics of available parking space and trace of parking session from the entry to exit.
- **Security:** Access to the Admin Dashboard requires authentication and all communication between components.

Quality Attribute	Source	Stimulus	Artifact	Environment	Response	Response Measure
Extensibility	Developer	Introduce a new microservice for a new capability	Microservices architecture	Development → Production	The new service is added without impacting existing ones	Zero downtime; no modification to other services' DB schema; integration completed in $\leq$ 1 day
Scalability (functional)	Product Owner	Need to add a new service	Microservices ecosystem	Deployment in production	New microservice is routed via Gateway and registered	No downtime; no changes to existing code; onboarding time $\leq$ 8h
Modifiability	Developer	Update tariff rules or change recognition model	Tariff service, Plate Detection service	Runtime in production	Update is deployed without disrupting other services	Zero downtime; all requests continue to be served; update rolled out via CI/CD in $\leq$ 30min
Testability	Tester	Execute unit/integration tests on Parking Management service	Microservices testable directly in real environment	Development/test environment	Each service is testable independently	Unit tests coverage $\geq$ 80%
Interoperability	External system	Integration with external provider	Embedded microservice, Gateway	Runtime in production	System integrates with external API securely	Integration time $\leq$ 2h
Observability	Operator/Admin	Need to monitor system status	Logging or Monitoring stack	Runtime (normal ops)	Metrics, and tracing available	Logs searchable in $<$ 5 click; alerts generated within 30s; trace coverage $\geq$ 50%
Security	Admin	Attempt to access without authorization	API Gateway, Auth service	Runtime (external access)	Unauthorized access to API	100% of unauthorized requests blocked.

Table 5: Quality Attribute Scenarios for Smart Parking System

2.9 Story Telling

In this section, we asked domain experts to describe what each actor does when interacting with the system. We invited them to tell us a story that is coherent with the flow of actions typically performed by an administrator to carry out their tasks, as well as those performed by a customer.

2.9.1 Customer Storytelling

At the entrance to the parking lot, a customer drives the car over a coil that activates the camera for plate detection. The camera reads the car's license plate, the system records the timestamp and write it in the DB with the plate just read; Then opens the barrier, allowing the customer to enter.

If there are any problem, the customer can always press the Help button, which calls the admin directly.

Before leaving, the customer must go to an automatic checkout to pay the parking fee. The system asks the customer to enter their license plate number. It then calculates the fee based on the entrance timestamp.

After the payment is complete, the customer can leave the parking lot using the same method they used to enter (plate recognition). There is also a Help button at the exit, so the administrator can manage any problems that may occur.

The scheme represents all the actions that can be performed in every parking session

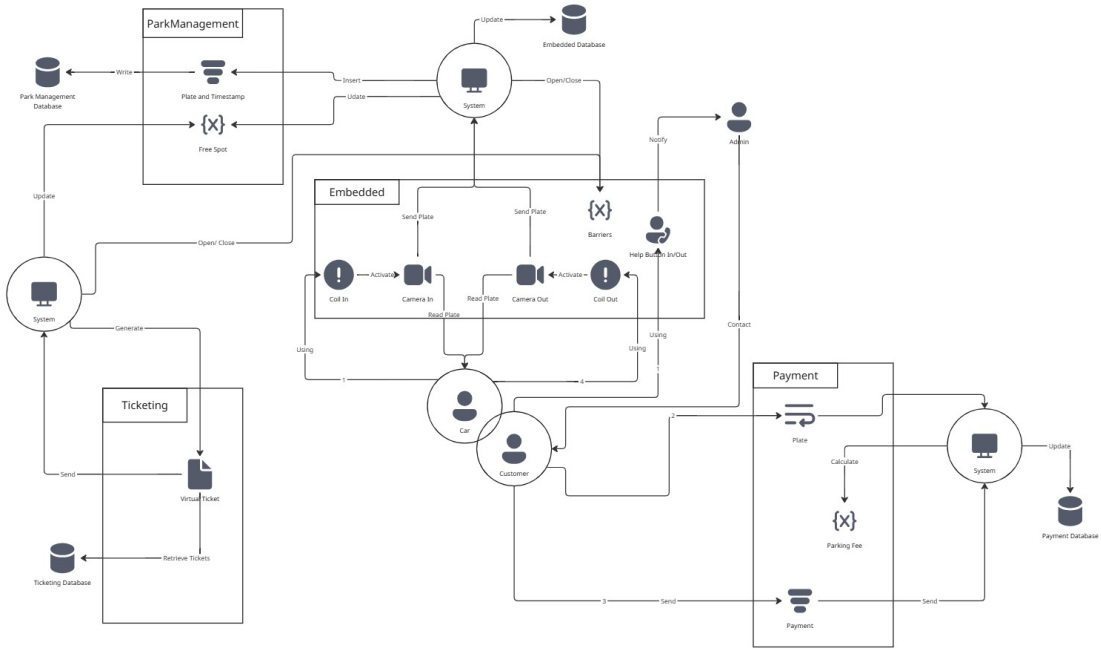

Figure 5: Customer Storytelling

(for every customer). The numbers on the lines indicate the priority of the action (arrows with the same number need to be read as an OR, only one of them can be performed), and the customers can not jump to higher numbers unless They have done the previous actions.

2.9.2 Admin Storytelling

At the beginning of the day, the administrator logs into the Admin Console through a web interface. After authentication, they can access a dashboard that shows the current status of the parking facilites, the number of free spots and a set of operations He can perform. During operations, the administrator may decide to update the tariff configuration. This configuration is performed through a dedicated section in the console and immediately propagated to the microservice responsible for pricing. If anomalies are detected such as vehicles remaining too long, incorrect payments, or plate dection errors the administrator can manually moify the session actions. In these cases, they may choose to override a parking session by manually insert the plate number so that the customers can start their parking session. These actions are logged and linked to the admin ID for traceability. At the end of the day or upon request, the administrator can generate detailed parking reports, including revenue and ticket dispensed. The administrator is also responsible for managing the physical barriers remotely in case of system malfunctions, and for monitoring the connectivity status of embedded devices like ESP32-CAM and NFC readers.

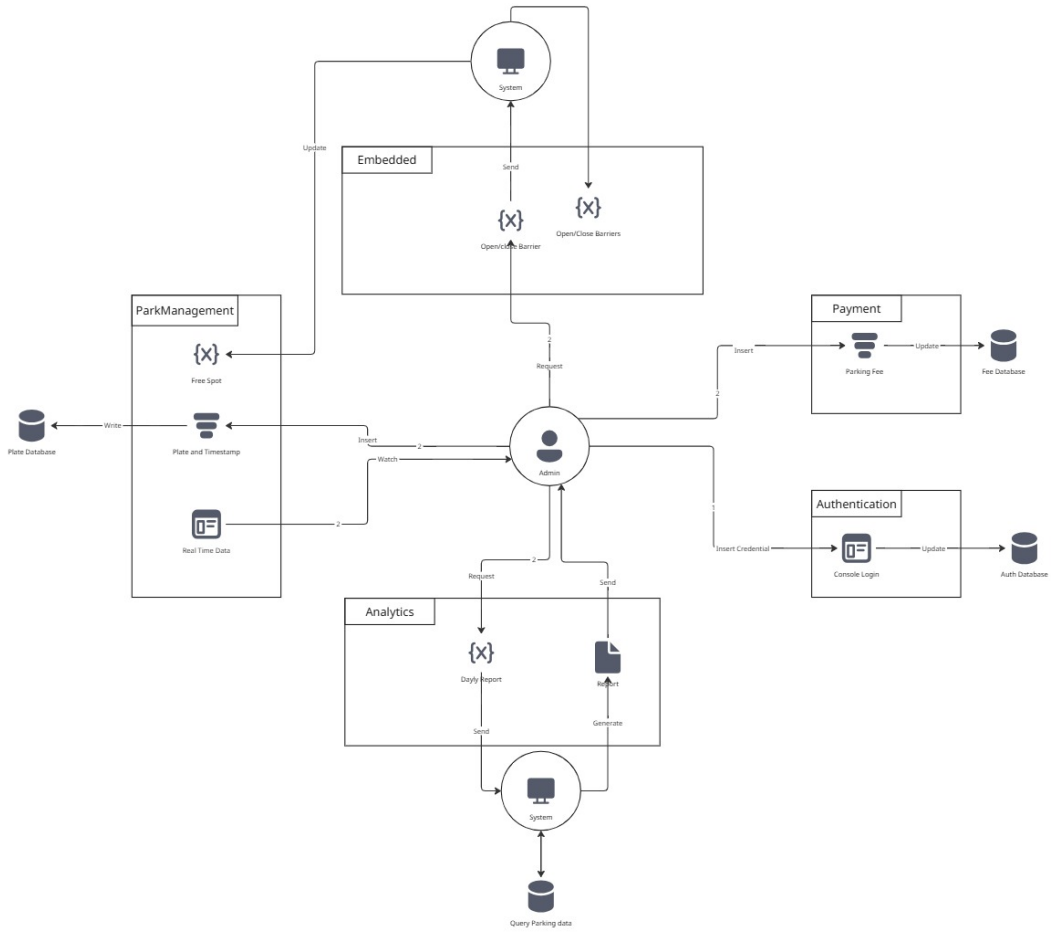

Figure 6: Admin Storytelling

The scheme represents all the actions that can be performed in every parking session (by an Admin). The numbers on the lines indicate the priority of the action (arrows with the same number need to be read as an OR, only one of them can be performed), and the Admin can not jump to higher numbers unless They have done the previous actions.

2.10 Domain Model

In this section we show the model of the system composed by the main entities and the main relation between them.

2.10.1 Entities and Value Objects

Entity	**Attributes**	**Role**
Car	id, plate	Represents the customer arriving at the parking lot whose license plate is read.
Customer	none	Represents the customer arriving at the parking lot who perform the operation of payment and press help button.
Admin	id, userName, password	Represents the user who has access to the software and can perform actions such as creating virtual tickets (to allow Car entry), generating reports (for the period of interest), communicating with Customer, and managing the entire embedded system.
VirtualTicket	id, licensePlate, entranceTimestamp, exitTimestamp, totalFee	Represents the profile associated with Car with data related to the current parking.
HelpButton	deviceId	System to connect Customer and Admin in case of problems. Present at the entrance and the exit.
Camera	deviceId	System for reading Car's license plate. Present at the entrance to read the license plate to be inserted in VirtualTicket, and at the exit for the verification required to open the barriers.
Report	id, date, Virtual Ticket Created, Virtual Ticket Completed, revenue,	Represents the report for a specific day with all revenue related to the parking lot.

Table 6: Principal Entities description

- Car: Interact with the Camera that can read the licensePlate
- Customer: He can press the HelpButton to interact with the Admin
- Customer: He can pay the fee.
- Admin: He can talk to the Customer
- Admin: He can generate a Report
- Admin: He can Generate a VirtualTicket

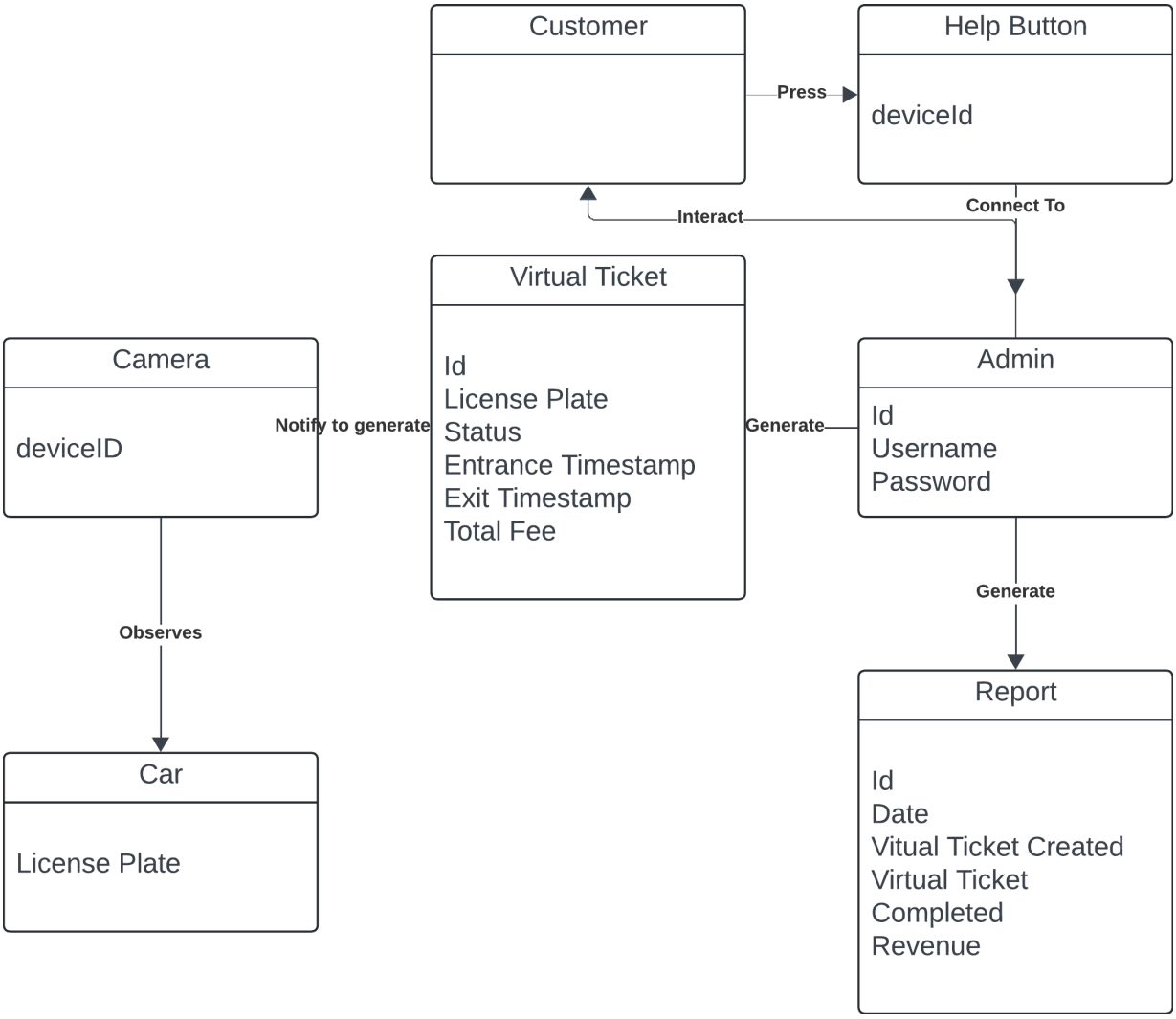

Figure 7: Entities Diagram with relations

ValueObject	**Role**
licensePlate	The id associated to a car. Needed to create a virtual ticket from the Camera or the Admin throuhg the HelpButton.
timestamp	The data with the date and the hour of a specific event.

Table 7: Value Objects and their Roles

2.10.2 Domain events

- **Plate Read IN**: when the Car (entrance) aproaches the Camera which has been able to read the license plate.

- **Plate Doesn't Read IN**: when the Car (entrance) aproaches the Camera which has NOT been able to read the license plate.
- **Plate Found**: when the Customer insert the license plate at the AutomaticCheckout and the system found the data associated to that license plate.
- **Payment Done**: when the Customer complete the payment process at the AutomaticCheckout.
- **Plate Read OUT**: when the Car (exit) aproaches the Camera which has been able to read the license plate and the system confirm the payment.
- **Plate Doesn't Read OUT**: when the Car (exit) aproaches the Camera which has NOT been able to read the license plate or the system do NOT confirm the payment.
- **HelpButtonINClicked**: when the Customer (entrance) press the HelpButton
- **HelpButtonOUTClicked**: when the Customer (exit) press the HelpButton

2.11 Bounded Context

From the analisys of the Domain Model, in this project we have identified different bounded context. We have define core context and support context and translate them into service following domain driven design suggestion. We identify the following:

- **Embedded(support)**: covers the logic and services related to physical devices and sensors (e.g., ESP32, cameras, barriers, coil sensors) that interact with the parking infrastructure and support integration with external system.
- **ParkManagement(core)**: responsible for managing parking spaces and providing administrators with a real-time overview of availability and system status information.
- **Analytics(support)**: collects and processes data on occupancy, and parking usage in order to generate statistics and reports that support service improvement.
- **Payment(core)**: handles payment flows and ensures validation, secure processing, and proper recording of transactions.
- **Ticketing(core)**: manages the issuance and handling of tickets (virtual), linking them to vehicles and users for parking entry and exit.
- **Authentication(support)**: manage administrator console authentication.

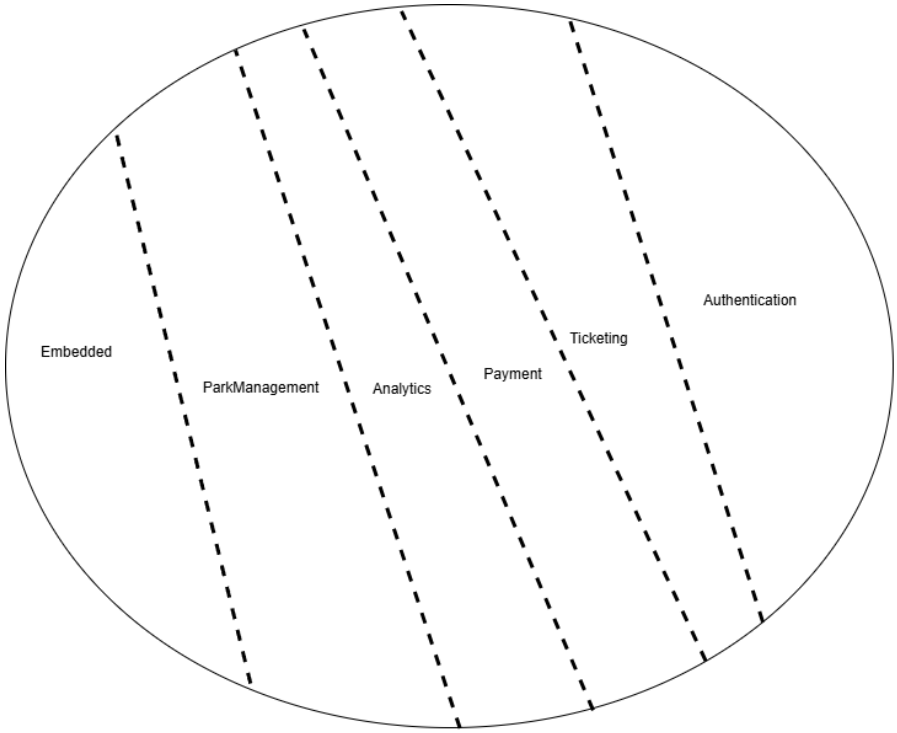

Figure 8: Bounded context

2.12 Context Map

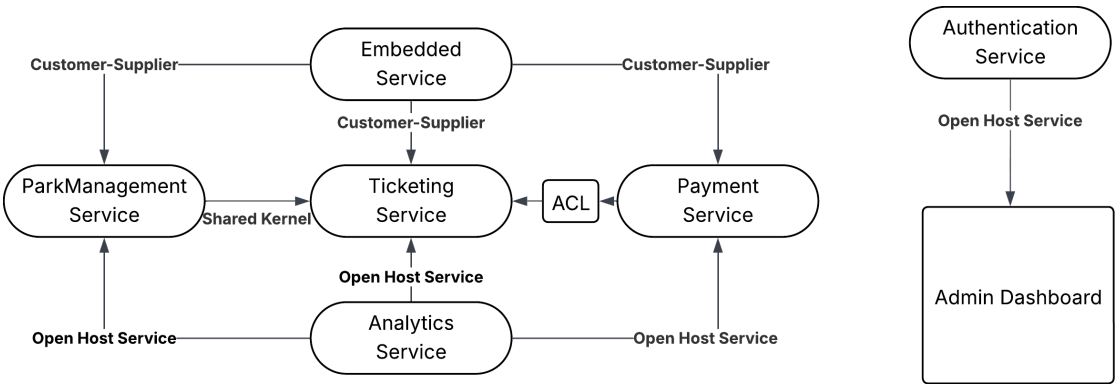

Figure 9: Context Map

The following section explains the relationships between the contexts identified in the Context Map and the applied DDD patterns.

- **Embedded → ParkManagement (Customer-Supplier)**:
  Embedded Service Collects events from physical devices (coil, cameras, NFC,

help button) while **ParkManagement Service** depends on data provided by Embedded to update parking status and control barriers. In this relation, the Embedded Service acts as the **supplier** of events, while ParkManagement is the **customer** consuming them to orchestrate the workflow.

- **Embedded → Ticketing (Customer-Supplier)**:
  Embedded Service sends plate detection events with relative value and the **Ticketing Service** creates or updates virtual tickets based on received information. Like the previous the Ticketing Service (**customer**) relies on Embedded Service (**supplier**) to start the ticket lifecycle.

- **Embedded → Payment (Customer-Supplier)**:
  Embedded Service notifies NFC events to initiate the payment process from the **Payment Service** and updates ticket status. The Embedded Service (**supplier**) provides the physical trigger, while Payment Service (**customer**) completes the business logic.

- **ParkManagement ↔ Ticketing (Shared Kernel)**:
  Both Services share key concepts such as "session" and "virtual ticket". With this pattern, they must maintain consistency regarding parking and ticket states.

- **Payment ↔ Ticketing (Anti-Corruption Layer)**:
  The **Payment Services** translates its payment model to update tickets without contaminating the **Ticketing Service**'s domain model. With this pattern the system prevents payment rules from influencing ticket domain logic.

- **Analytics → ParkManagement / Payment / Ticketing (Open Host Service)**
  The **Analytics Service** consumes data exposed by services through APIs. This aproach avoid modification of the data caused by the Analytics Service but uses it to generate reports.

- **Authentication Service → Admin Dashboard (Open Host Service)**
  Provides login/logout functionality for administrators to access the dashboard and perform operations. The **Authentication Service** exposes APIs for user access

3 Domain Driven Design to Microservices

In following subsections we describe how we created the microservices architecture starting from the definition of the domain and subdomains described in previous sections.

3.1 System operations

In this section are shown the mais systems operations identified in the smart parking system. Operations are derived use cases and represent the main commands that guides the behaviour of the system.

Actor	**Story**	**Command**	**Description**
Car	US4–US5 – Vehicle detection and plate reading	`readLicensePlate()`	Activates the camera at the entrance when the coil detects a vehicle, reads the license plate, and sends it to the backend to start a new parking session.
Customer	US12–US13 – Payment process	`processPayment()`	Processes the user's payment (via NFC) at the automatic checkout, updating the session and marking the payment as completed.
Customer/Car	US10–US11 – Exit authorization	`authorizeExit()`	Verifies payment and authorizes vehicle exit by opening the barrier.
Admin	US1 – Modify tariff plans	`updateTariff()`	Allows the administrator to update and apply new tariff rules to the parking system.
Admin	US2 – Manual barrier control	`openBarrier()`	Lets the administrator manually open or close barriers in case of failure or emergency.
Admin	US16 – Generate reports	`generateReport()`	Produces reports based on parking sessions and payment data.

Table 8: Main system operations derived from user stories and domain model

In the following the main queries of the system:

Query	Description
`getVirtualTicket(plateNumber)`	Retrieves all information about the parking session associated with a specific license plate. It includes entry timestamp and payment status. Used by the *Payment* service and exit checkout to verify session validity.
`getAllTickets()`	Retrieves all the virtual tickets associated with all the sessions not yet *COMPLETED* or *PAID*. Useful for process the reports or as a useful tool for the admin.
`getAllHistoryTickets()`	Retrieves all the virtual tickets associated with all the sessions *COMPLETED* or *PAID*. Useful for process the reports or as a useful tool for the admin.
`getSlots()`	Returns the current number of free parking spots in real time. Used by the *Park Management* service and displayed on external or user interfaces.
`getReport(date)`	Generates an analytical report for a given date, including metrics such as revenue, and occupancy rate. Used by the *Analytics* service.
`getStatus()`	Retrieves the operational status of all the embedded device (ESP32-CAM, NFC reader, barrier actuator). Used by administrators for system monitoring and maintenance.
`getTariff()`	Retrieves the current active tariff configuration, pricing rules.

Table 9: Main system queries identified in the Smart Parking system

3.2 Subdomains to microservices

To correctly apply the decomposition pattern to our domain we associate each subdomain to a microservice, before doing that we define formally what are the responsabilities of each subdomain in following tables.

3.2.1 Park Management Service

Responsabiliy	Description
Modify park status	Show and update the number of free spots.
Monitoring the cars	Triggers the generation of a virtual ticket managing the flow of the ticket's event
Logging	Logs all the events triggered by all the services.

Table 10: Park Management Service responsibilities and description

3.2.2 Authentication Service

Responsabiliy	Description
Login to Admin Console	Allows the administrator to access the management interface. Required for all other admin actions.
Logout to Admin Consol	Allows the administrator to exit the management interface.

Table 11: Authentication Service responsibilities and description

3.2.3 Analytics Service

Responsabiliy	Description
Generate Report	Allows the administrator to generates reports about a specifics date retrieving all the data from the databases.

Table 12: Analytics Service responsibilities and description

3.2.4 Embedded Services

Responsibility	Description
Device Integration	Manages communication with embedded devices such as ESP32-CAM, ESP32 controllers, NFC readers, and barrier actuators through the MQTT protocol.
Event Handling	Receives and publishes MQTT events related to physical interactions (plate detection, NFC tap, coil activation, help button). Each event is translated into REST API calls to the appropriate backend microservices.
License Plate Detection	Handles messages sent by the ESP32-CAM when a vehicle approaches the entry or exit point. Transfers the recognized plate to the backend for ticket generation.
Barrier Control	Publishes barrier open/close commands on specific MQTT topics when a parking session is validated or payment is confirmed.
Device Monitoring	Reports device connectivity, operational status, and errors to the *Parking-Service* for system observability.

Table 13: Embedded Service responsibilities and description

3.2.5 Payment Management

Responsibility	Description
Tariff Calculation	Computes the parking fee based on entry/exit timestamps and the active tariff configuration.
Payment Processing	Manages the payment transaction flow, including NFC tap detection (via the Embedded Service) and virtual ticket validation. Updates the session status once the payment is completed.
Payment Verification	Provides APIs for verifying payment status at exit terminals before allowing the barrier to open. Ensures synchronization between user session and transaction records.
Receipt Generation	Save the data of all Virtual tickets with status `PAID` in a specific collection so that all the transaction are persistent.
Transaction Logging	Logs all payment-related events for traceability and auditing, storing relevant metadata with the structure of a Virtual Ticket (timestamp, licensePlate, amount).

Table 14: Payment Service responsibilities and description

3.2.6 Ticket Management

Responsabiliy	Description
Generate Virtual Ticket	Use the license plate as unique Id and generate the ticket with the entrance timestamp of the Car.
Update Ticket	During the payment process, the Virtual Ticket was modified with the payment/exit timestamp and the new status. At the exit park meter.

Table 15: Ticketing Service responsibilities and description

3.3 System Operation Identification

Micro-service	**Operations**	**Collaborators**
Park Management	addCar(), removeCar(), getCar(), getSlots(), getEmbeddedStatus(), openEntryBarrier(), openExitBarrier(), getAllHistory()	• Embedded Service • Ticketing Service
Authentication Service	logIn(); logOut()	• none
Analytics Service	generateReport(), getReport(), getReports()	• Ticketing Service
Embedded Management	openEntryBarrier(), openExitBarrier(), readLicensePlate()	• none
Payment Service	setTariff(), getTariff(), processPayment(), CalculateFee()	• Ticketing Service
Ticketing Service	generateVirtualTicket(), updateVirtualTicket(), getTicket(), getAllTickets()	• none

3.4 API Interface definition and identification

In this section the interface of the system are explained. Because of the system is cyber-physical we created two different interfaces: the first embedded interface communicate with the physical part using MQTT and publish/subscribe pattern used in IoT. The second communicates with the backend modules through REST API.

3.4.1 Embedded-Service

EndPoint	**Type**	**Description**
/embedded-service/open-exit-barrier	POST	Opens the exit-barrier. The service publishes an MQTT command on topic `parking/sensor/cmd` with payload `"command":"openExit"`. Used by the operator console.
/embedded-service/open-entry-barrier	POST	Opens the entry-barrier. The service publishes an MQTT command on topic `parking/sensor/cmd` with payload `"command":"openEntry"`. Used by the operator console or camera at entry or exit.
/embedded-service/status	GET	returns the status of emmbedded devices
/embedded-service/nfc/enable/{plate}	POST	Enable the embedded NFC module to receive payment after the customer insert the licensePlate into the checkout.
/embedded-service/payment-confirmed	POST	Returns the confirmation of the payment and trigger the Virtual ticket update.
/embedded-service/getPendingPayment	GET	Return the Virtual ticket in PENDING status. The one that have to be PAID. (One at a time).
/embedded-servise/deletePendingPayment	GET	Delete the Virtual ticket from the PENDING collection after the payment.

Table 16: Embedded-Service REST API endpoints

3.4.2 Parking-Service

EndPoint	Type	Description
/parking-service/addCar	POST	Update the parking status when a car enter the park. Trigger the creation of a virtual ticket.
/parking-service/removeCar	POST	Update the parking status when a car or a customer exit the park. Remove a car from the park.
/parking-service/getCar/{id}	GET	Retrieve the car's informations from the DB
/parking-service/getSlots	GET	Retrieve the number of free spots from the embedded-service
/parking-service/getEmbeddedStatus	GET	Retrieve the devices's informations about their status from the embedded-service
/parking-service/openExitBarrier	POST	Trigges the command on the embedded-service to opens the exit-barrier.
/parking-service/openEntryBarrier	POST	Trigges the command on the embedded-service to opens the entry-barrier.
/parking-service/getAllHistory	POST	Call the ticketing-service to retrieve all the virtual tickets with status `PAID`.

Table 17: Parking Management-Service REST API endpoints

3.4.3 Analytics-Service

EndPoint	Type	Description
/analytics-service/report/{date}	GET	Search or Generates a file (json) containing a general analytical report in a certain date.
/analytics-service/report/get/{date}	GET	Retrieve an existing report about a certain date.
/analytics-service/reports	GET	Retrieves all the generated reports.

Table 18: Analytics-Service REST API endpoints

3.4.4 Payment-Service

EndPoint	Type	Description
/payment-service/calculateFee	POST	Calculates the parking fee using entry and exit timestamps and the current tariff configuration from the Park Management service.
/payment-service/processPayment/{NFCTag}	POST	Initiates the payment process (NFC tap). Updates the parking session and marks payment as completed.
/payment-service/setTariff	POST	Set the new tariff configuration.
/payment-service/getTariff	GET	Retrieves the active tariff configuration for use in fee calculation and UI display.

Table 19: Payment-Service REST API endpoints

3.4.5 Ticketing-Service

EndPoint	Type	Description
/ticketing-service/createVirtualTicket	POST	Create a virtual ticket for simulation or testing purposes, associated with a car or external system. Saved with *IN_PROGRESS* status.
/ticketing-service/virtualTicket/{id}	GET	Retrieve all information about a specific virtual ticket using its unique ID.
/ticketing-service/virtualTicket/{id}	POST	Update the information of an existing virtual ticket using its ID.
/ticketing-service/getTicketByPlate/{plate}	GET	Retrieve a ticket by the vehicle's license plate instead of its ID. Returns the most recent active ticket associated with that plate.
/ticketing-service/updateStatus/{plate}	POST	Update the status and exit information of a ticket using the vehicle's license plate (e.g., set to *COMPLETED* when the car exits).
/ticketing-service/getAllTickets	GET	Retrieves all the virtual tickets with the status *IN PROGRESS* from the ticket collection.
/ticketing-service/getAllHistoryTickets	GET	Retrieves all the virtual tickets with the status `PAID` or *TERMINATED* from the history collection.
/ticketing-service/addToHistory/{plate}	POST	After the virtual ticket was paid, the sarvice move it in the History collection.
/ticketing-service/verifyPaymentByPlate/{plate}	GET	Verify that the status of a specific virtual ticket is *PAID*.

Table 20: Updated Ticketing-Service REST API endpoints

3.4.6 Authentication-Service

EndPoint	Type	Description
/auth-service/login/username	GET	Verify the presence of the User on the DB to give the permit to access the system
/auth-service/login	POST	Login the user (administrator) in dashboard console.
/auth-service/logout	POST	Logout the user (administrator) from dashboard console.

Table 21: Authentication-Service REST API endpoints

4 System Architecture and Microservices Architecture

The Smart Parking system is designed as a distributed microservice ecosystem that combines backend services, an embedded IoT subsystem, a web-based frontend, and DevOps infrastructure components. All microservices run inside a shared Docker network, discovered dynamically through Eureka and exposed externally via a single entry point: the API Gateway. This design guarantees independent evolution, scalability, and maintainability of each service.

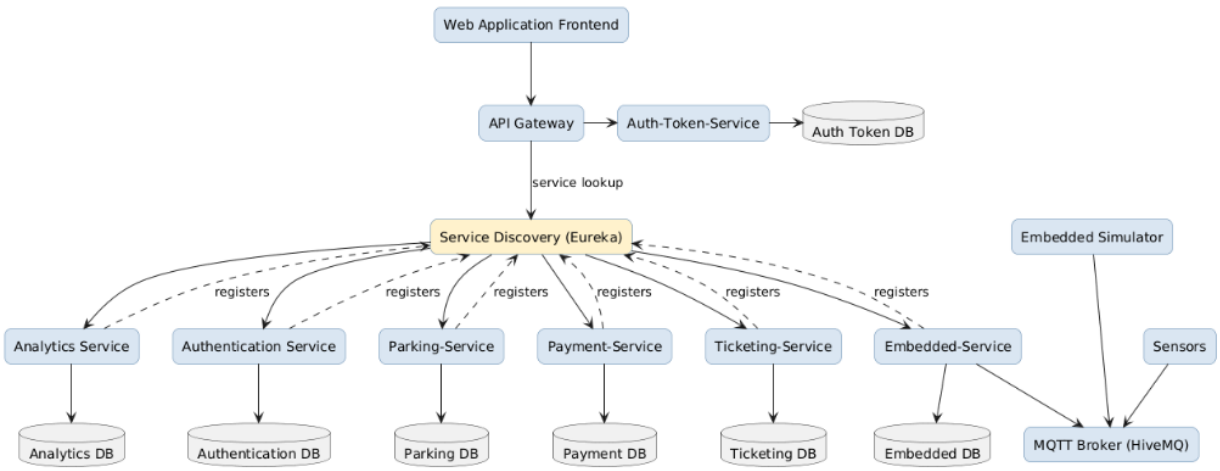

Figure 10: System Architecture Schema

The backend consists of six core microservices: *Authentication, Parking, Ticketing, Payment, Analytics* and *Embedded-Service*. Each service is fully isolated, owns its database, and exposes its behaviour through HTTP REST APIs and for the embedded also an Event Architecture interface able to listen the sensors.

Communication from the external to the micro services are mediated by the API Gateway and dynamically routed through Eureka. MQTT Messages and Broker is used for asynchronous communication with the embedded devices.

Each microservice is deployed as an independent Docker container, allowing selective updates, rolling deployments, and system-wide resilience through fault isolation. Horizontal scalability is enabled at the service level.

4.1 Microservice Architectural Patterns Adopted

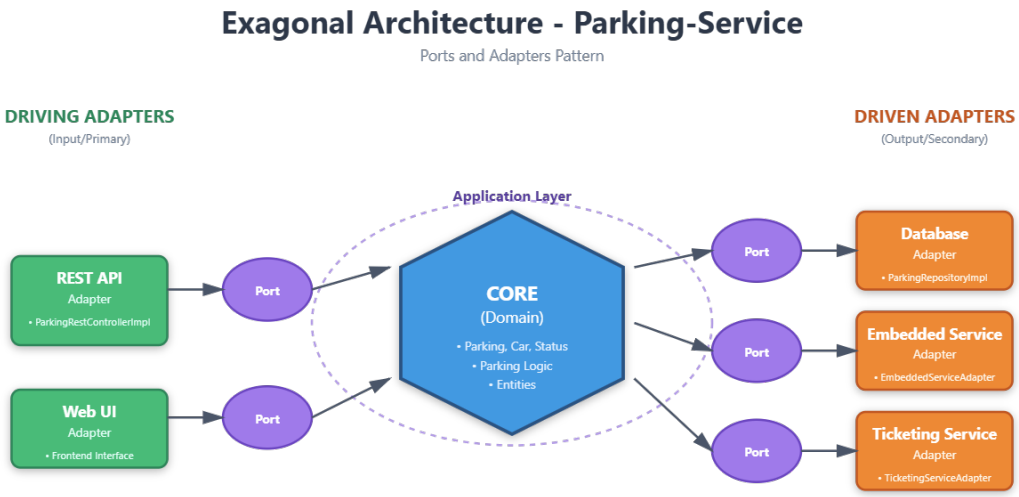

Figure 11: Parking Service Hexagonal Architecture

The Smart Parking system adopts the **Hexagonal Architecture** as the foundational architectural style for every microservice in the platform. This choice, aligned with the guidelines presented in the course, ensures that each service remains modular, maintainable, technology-agnostic, and fully compliant with the quality attributes required for distributed and event-driven systems.

As shown in figure 11, each microservice is structured around a **central Domain Core**, which contains the business logic, domain models, and application rules. The Core is surrounded by a set of **Ports**, which act as stable interfaces defining how the Domain interacts with its external world. The actual implementations of these ports are provided by **Adapters**, separated into two groups:

- **Driving Adapters (Input)**: REST controllers and the Web UI trigger the execution of use cases by invoking the exposed ports. They represent all external actors that initiate interactions with the microservice.
- **Driven Adapters (Output)**: These adapters implement the ports used by the Domain to interact with external technologies such as MongoDB (database adapter which is repository interface), the Embedded-Service (MQTT and event-bus adapters), and other microservices (Ticketing-Service, Payment-Service). By

relying on interfaces rather than concrete implementations, the Domain remains independent from infrastructure and communication details.

This architectural style is consistently applied across all microservices of the Smart Parking project.

4.1.1 Decomposition by Subdomain

The system is decomposed according to **subdomains and bounded contexts** as recommended by Domain-Driven Design. Each context maps directly to one microservice, ensuring clear business ownership and avoiding a shared, monolithic model. This decomposition is reflected also in package diagram as shown Figure 12.

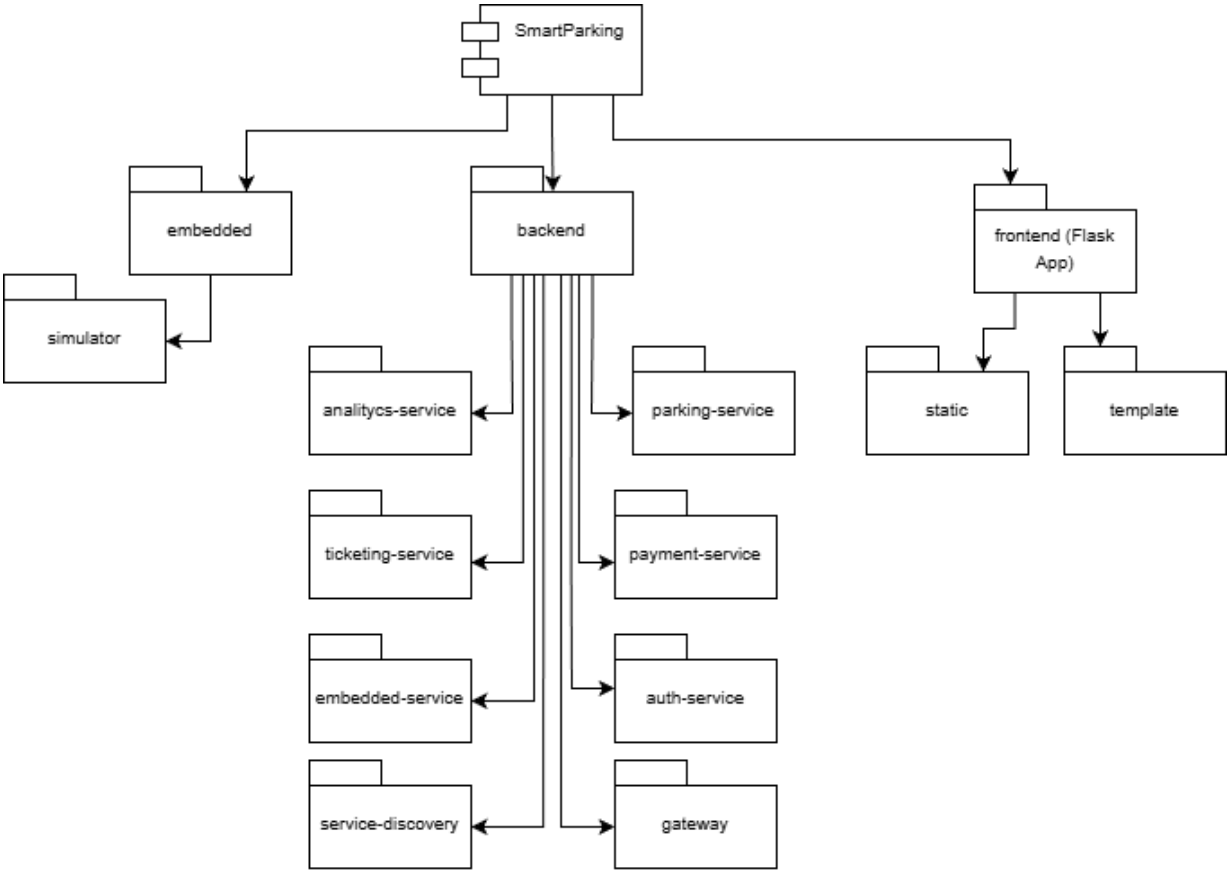

Figure 12: Package Diagram

4.1.2 Communication Patterns

Synchronous Communication Some of internal and external requests use REST request, enabling structured request-response semantics.

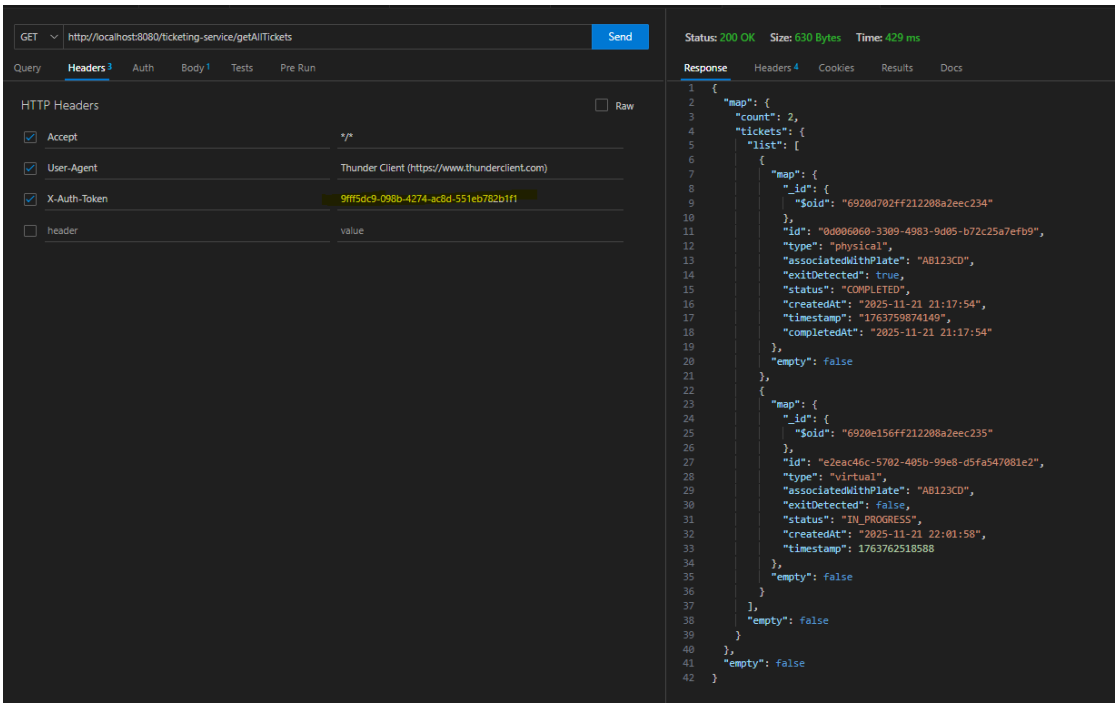

Figure 13: Example of API Rest request

Asynchronous Communication MQTT enables asynchronous event ingestion from the embedded subsystem (plates, coils, NFC), decoupling device events from backend processing. This reduces temporal coupling and increases resilience.

Service Discovery All services register in Eureka at startup. This implements the "Client-Side Discovery" pattern, where the gateway and services resolve instances dynamically instead of using static endpoints.

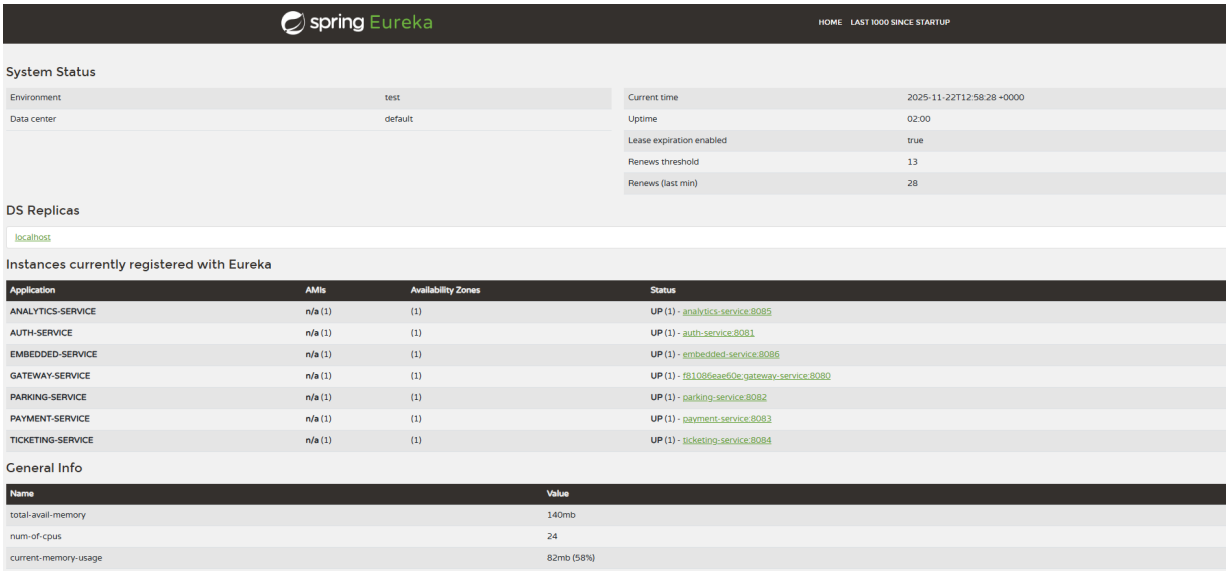

Figure 14: Eureka Spring Service Discovery

API Gateway The Gateway acts as a single entry point from external, applying routing and authentication by token as shown in Security Pattern subsection below.

4.1.3 Resilience and Fault Tolerance

All microservices are stateless and isolated in independent Docker containers, which prevents faults from propagating across the system. Moreover, every REST interaction is wrapped in standard Spring error handling: when a backend service encounters an invalid request, unreachable dependency, or domain violation, it returns a structured HTTP error response without blocking the service or compromising its stability. This ensures that failures remain local, predictable, and recoverable.

Additional architectural elements contribute to resilience:

- **Service isolation**: each microservice can fail independently without affecting others.
- **Controlled error propagation**: REST calls return consistent error responses instead of causing service crashes.
- **Eureka-based recovery**: if a service restarts, it re-registers dynamically without requiring external reconfiguration.

This architecture guarantees **operational fault tolerance**: services never block, the application continues functioning under partial failure, and errors are returned gracefully, an example in Figure 15, the API Request return status OK, but the error is explicited in the message.

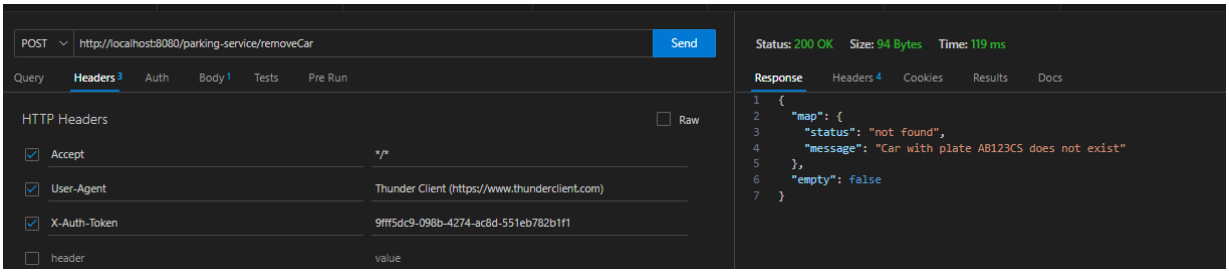

Figure 15: Fault Tolerance in case of error

4.1.4 Querying Data Patterns

API Composition Each microservice exposes its own REST API for reading and writing data, and no component in the system is ever allowed to access another service's database directly. This ensures strong service autonomy, clear ownership of data, and full compliance with the principles of Domain-Driven Design and microservice isolation. In the Smart Parking system, the API Composition pattern is applied inside the API Gateway for specific read operations that require information from multiple microservices. This is particularly relevant for views such as the admin dashboard or the system monitoring page, where data needs to be aggregated from the Parking-Service, Ticketing-Service, Payment-Service, and Analytics-Service. This centralizes the aggregation logic and prevents the frontend from having to orchestrate several remote calls independently. The pattern is intentionally applied only to cross-service queries exposed to the admin frontend.

4.1.5 Deployment Pattern

Each microservice is deployed as a Docker container as shown in Figure 16.

Name	Container ID	Image	Port(s)	CPU ...	Memory usage...	Memory (%)	Disk read/write	Network I/O	PIDS	Actions
embedded-service	ae7ff1af99e9	backend-embedded-service	8086:8086	0.12%	503.4MB / 15.48(	3.18%	0B / 0B	3.65MB / 70.8MB	119	
auth-service	d0293c93b000	backend-auth-service	8081:8081	0.26%	480MB / 15.48GE	3.03%	0B / 0B	3.73MB / 73.8MB	69	
analytics-service	8663d9685cee	backend-analytics-service	8085:8085	0.18%	531.7MB / 15.48(	3.35%	0B / 0B	4.81MB / 74MB	81	
parking-service	e6adf75383a9	backend-parking-service	8082:8082	0.11%	516.8MB / 15.48(	3.26%	0B / 0B	4.65MB / 73MB	80	
ticketing-service	914d9399d8a4	backend-ticketing-service	8084:8084	0.1%	548.3MB / 15.48(	3.46%	0B / 0B	4.82MB / 74.7MB	78	
gateway	f81086eae60e	backend-gateway	8080:8080	0.09%	752.1MB / 15.48(	4.74%	0B / 0B	2.14MB / 57.9MB	87	
payment-service	90db76ae0227	backend-payment-service	8083:8083	0.13%	547.1MB / 15.48(	3.45%	0B / 0B	4.65MB / 73.6MB	80	
service-discovery	68799e319dfa	backend-service-discovery	8761:8761	0.67%	465.8MB / 15.48(	2.94%	0B / 0B	4.91MB / 61MB	89	
grafana	ec080fbbcd74	grafana/grafana	3000:3000	0.2%	105.5MB / 15.48(	0.67%	0B / 0B	180KB / 79KB	34	
prometheus	419d6828c221	prom/prometheus:latest	9090:9090	0.07%	62.39MB / 15.48(	0.39%	0B / 0B	549MB / 16.7MB	20	
mongo-auth	4b7e8c9a7532	mongo	27017:27017	0.33%	159MB / 15.48GE	1%	0B / 0B	467KB / 1.24MB	46	
mongo-analytics	6eb9d4ccb763	mongo	27021:27017	0.39%	150.8MB / 15.48(	0.95%	0B / 0B	1.75MB / 3.92MB	60	
mongo-parking	ed685450a56d	mongo	27018:27017	0.38%	151.6MB / 15.48(	0.96%	0B / 0B	822KB / 2.3MB	49	
mongo-payment	61a3e53ff366	mongo	27019:27017	0.44%	152.9MB / 15.48(	0.96%	0B / 0B	822KB / 2.3MB	49	
mongo-ticketing	e4f1e2f63483	mongo	27020:27017	0.42%	167.9MB / 15.48(	1.06%	0B / 0B	825KB / 2.3MB	49	
mongo-embedded	576bef2aebe8	mongo	27022:27017	0.1%	155.5MB / 15.48(	0.98%	0B / 0B	1.26MB / 2.73MB	55	

Figure 16: Smart-Parking containers in Docker Desktop

4.1.6 Observability Patterns

Health Checks All microservices in the Smart Parking system expose a standard health endpoint through Spring Boot Actuator. The `/actuator/health` endpoint provides a real-time indication of the service status, reporting whether the application is operational and whether its dependencies such as the dedicated MongoDB instance of each microservice are reachable and functioning correctly.

These health endpoints are also integrated into the Docker Compose environment, where health checks are defined to automatically verify the availability of each service before it is considered ready. This ensures that dependent services do not start until the underlying components are fully operational. We have also implemented *Prometheus Server* that periodically queries these endpoints as part of the monitoring pipeline, enabling *Grafana* dashboards to display live service health information.

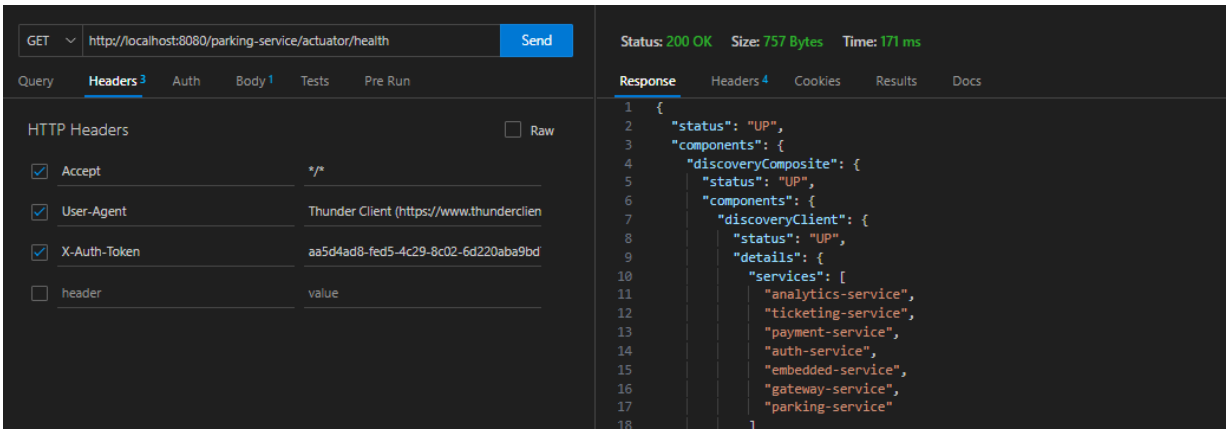

Figure 17: Health-check endpoint REST API request

Application Metrics The Smart Parking system includes a full monitoring pipeline based on *Prometheus*, Figure 18 and *Grafana* Figure 19. Each microservice exposes metrics through the Spring Boot Actuator endpoint `/actuator/prometheus`, which are periodically scraped by the Prometheus server defined in the deployment environment. The collected metrics include request rates, response times, error counts, memory usage, and operational counters related to ticket creation, sensor event processing, and parking state updates.

Prometheus stores these measurements and makes them available for real-time analysis, while Grafana provides the visualization layer. A preconfigured *Grafana* instance is included in the Docker Compose environment, allowing developers and operators to inspect dashboards that display system metrics.

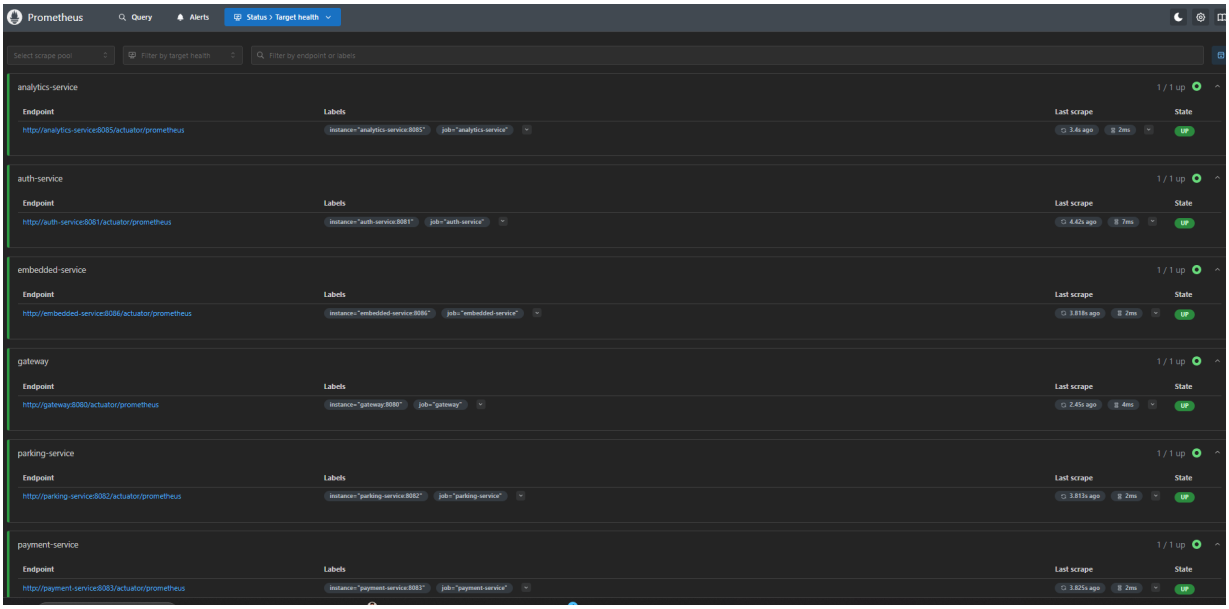

Figure 18: Smart-Parking Prometheus metrics

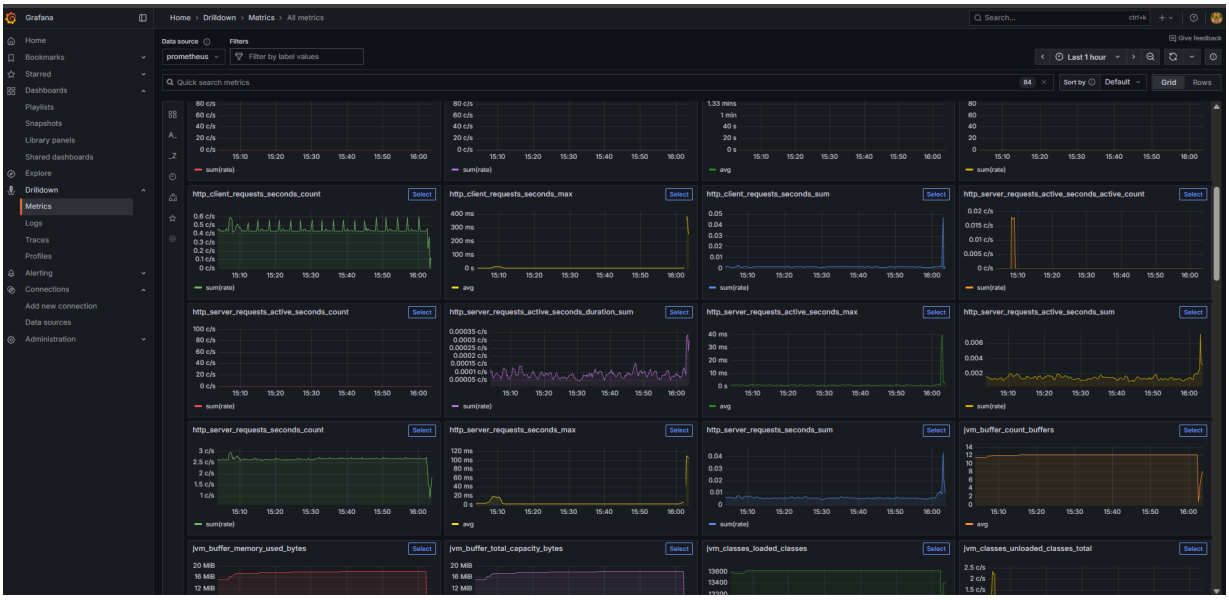

Figure 19: Smart-Parking Grafana Dashboard

4.1.7 Cross-Cutting Concerns

The system uses externalized configuration such as environment variables in .env file and YAML files as shown in Figure 20 to encapsulate the configuration of entire systems. This avoid to use connection strings or other architecture variables in code.

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  #Services / Infrastructure
  - job_name: 'analytics-service'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['analytics-service:8085']

  - job_name: 'embedded-service'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: [ 'embedded-service:8086' ]

  - job_name: 'ticketing-service'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: [ 'ticketing-service:8084' ]
```

Figure 20: Part of Prometheus YAML configuration file

Also Docker container configuration is explicited in docker-compose.yml.

4.1.8 Security Pattern

Security is enforced following the Access Token Pattern. Tokens are requested to the gateway issued by the Auth-Token-Service that store the token in a secure DB. Before each request, the gateway validate the token and check if it's available in DB ensuring it's authorized. In positive case the gateway redirect the request to proper microservices. In Figure 21 the sequence Diagram of authentication flow is shown.

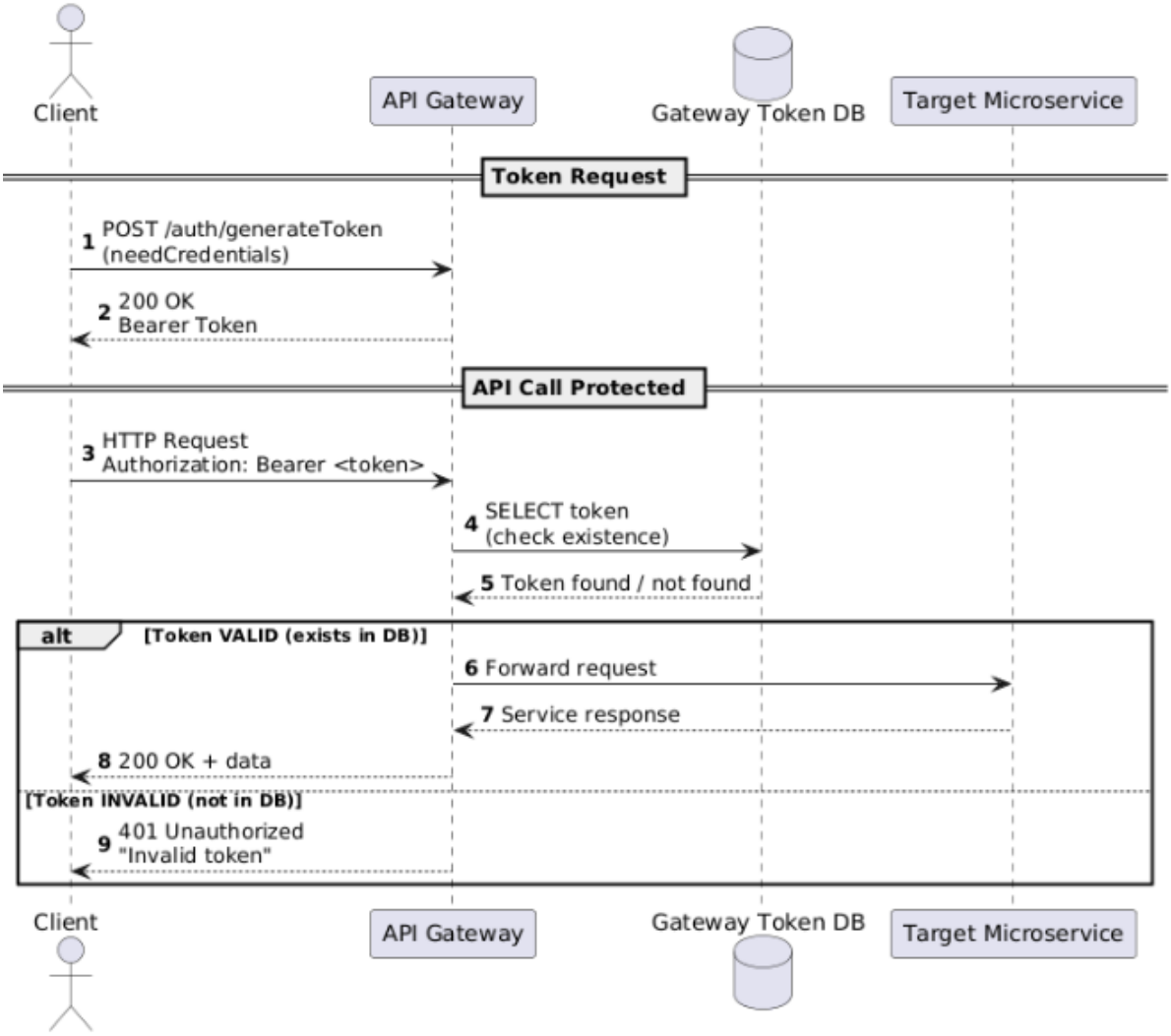

Figure 21: Sequence Diagram of Authorization in gateway

4.2 Event-Driven Architecture for Reactive Systems

The Smart Parking system relies on a fully event-driven architecture to manage the real-time flow of information coming from the IoT subsystem. This architecture is implemented inside the *Embedded-Service*, whose responsibility is to ingest MQTT sensor events, preprocess them, route them internally through a Vert.x Event Bus, and trigger the appropriate microservices in the backend.

4.2.1 MQTT-Based Ingestion Layer

All sensors devices publish their data to the HiveMQ broker using structured MQTT topics and subscribe to them following the Publish-Subscribe pattern. The system supports coil detection, NFC scans, help button requests, keyboard manual input, and license plate recognition. Topics include:

- `parking/sensor/coildetection`
- `parking/sensor/platedetection`
- `parking/sensor/nfc`
- `parking/sensor/cmd`
- `parking/sensor/helpbutton`
- `parking/sensor/keyboardInsertion`
- `parking/status/<deviceId>`

The ingestion component is implemented in the `VertxMqttClientAdapter`, a custom Vert.x verticle configured via the `MqttConfig` class. This component:

- creates and manages the MQTT client instance,
- subscribes to all relevant sensor topics at startup,
- parses incoming JSON payloads using Jackson,
- extracts domain fields such as timestamp, plate number, NFC Data or coil activation state.

A shared *sensorStatusMap* (based on a thread-safe `ConcurrentHashMap`) is also maintained to store the last-known status of each device. This enables the system to expose the aggregated state of all sensors through REST endpoints as shown in Microservices Architecture section.

4.3 Internal Event Bus Architecture

Once an MQTT message is received and validated, the adapter forwards it to a Vert.x Event Bus channel. Each type of event is routed to a different event address, ensuring clean separation of responsibilities: Each verticle processes its event asynchronously and encapsulates the logic that transforms raw device data into backend actions. An example of this flow of operation is illustred in Sequence Diagram in Figure 22. This pattern decouples ingestion from processing, allowing the system to handle sensor bursts, device delays, and network interruptions without compromising responsiveness.

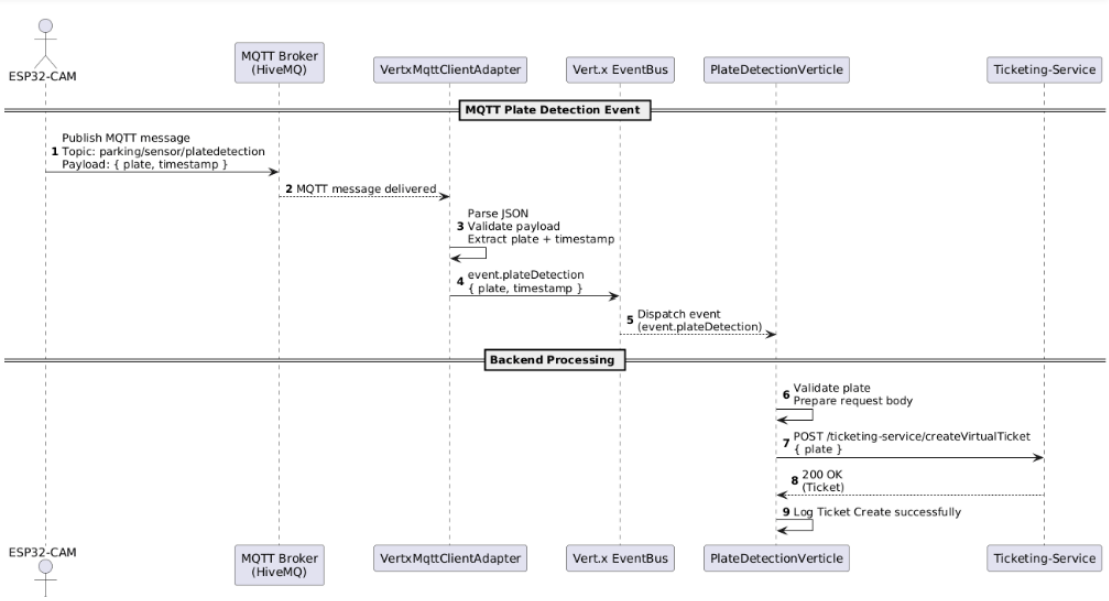

Figure 22: Sequence Diagram of MQTT Messages arriving into SmartParking System

All events are also saved into the embedded microservice database ensuring logging and history of events.

4.3.1 System Advantages

The event-driven design provides several benefits:

- **Loose coupling** between sensors and backend logic.
- **Real-time behaviour** thanks to asynchronous Vert.x pipelines and mqtt topic.
- **Scalability**: multiple verticles can be deployed to scale event throughput.
- **Fault isolation**: failures in one event type do not affect others.
- **Resilience**: MQTT ingestion continues even if a backend microservice is temporarily unavailable.

- **Consistent state**: the status map maintains the latest sensor readings for dashboards.

5 Admin Dashboard - Frontend

In this section we describe how we implemented the Frontend Layer of the system to make the Administrators able to interact with It. In the previous chapters we introduce the admin dashboard, this refers to a group of pages that call through an API the useful EndPoint that execute the admin operations.

5.1 Dashboard

This page is the entry point to manage the park. Here the administrator can watch the status of the Embedded Service, change the tariff, add a new car and control the barriers as shown in Figure 23.

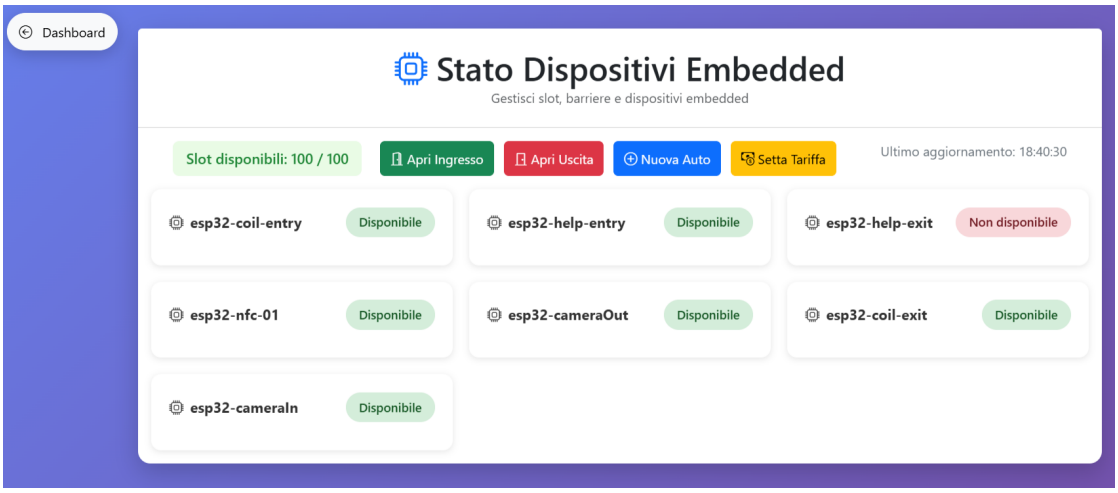

Figure 23: Administrator Dashboard

5.2 Analytics

This page is the entry point for the analytics service. The admin can see the list of the generated report and request a new report through a pop-up with date selection, this form is shown in Figure 24 and 25.

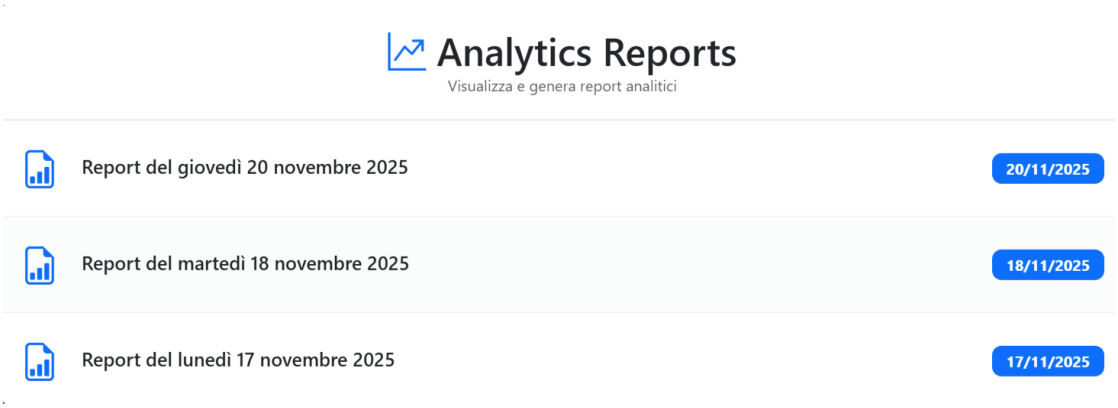

Figure 24: Analytics

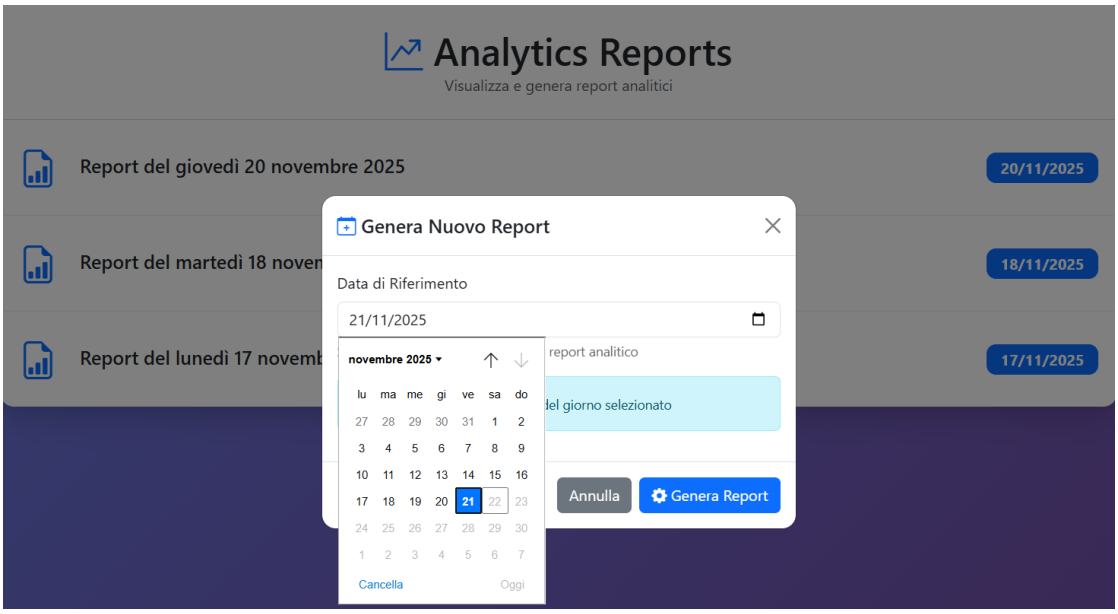

Figure 25: Request Report Pop-Up

5.3 Tickets Viewer

This page, shown in Figure 26, is a list of all the tickets generated from the system. It is necessary to permit to the administrator to have an overview of the actual situation of the parking, and assists a customer: for example if a customer can't exit the park for a problem with the camera, the admin can verify if the customer have paid and then he can open the barrier.

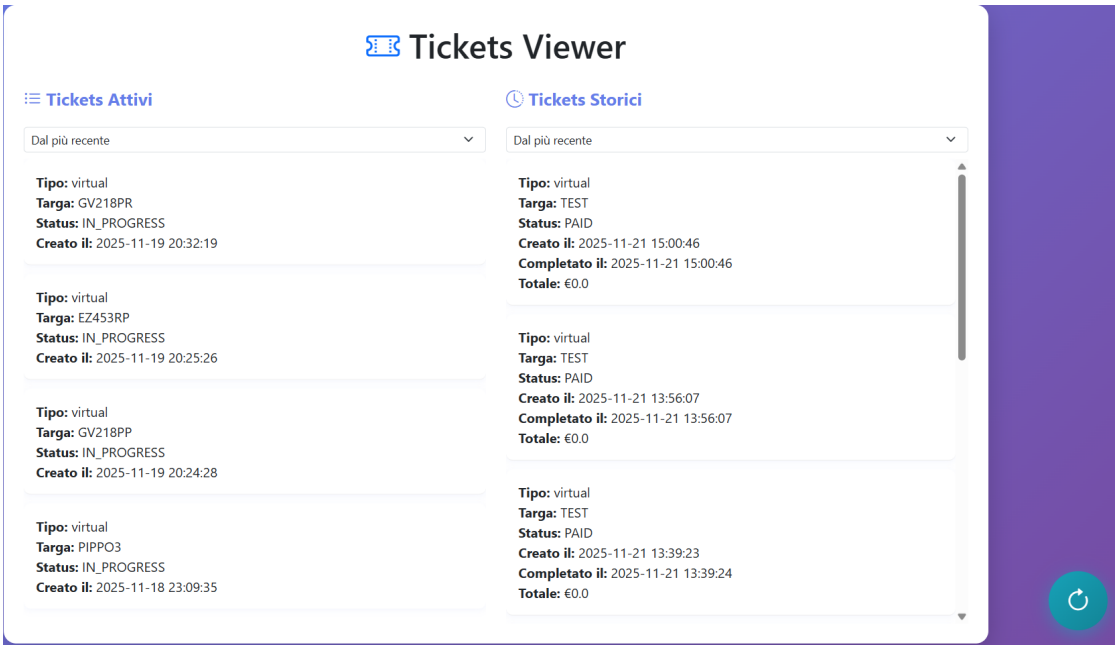

Figure 26: Tickets List

6 Test & Validation

The validation of the Smart Parking system required a structured and multi-layered testing approach. Given the nature of the project a distributed, event-driven, cyber-physical system the testing phase needed to account not only for software correctness, but also for communication between microservices and, at the highest level, the simulation of real physical events. For this reason, the testing process follows the architectural progression: first the isolated logic, then the interactions, and finally the full system behaviour.

The testing strategy follows the principles of the **Test Pyramid**, combining different levels of verification, each addressing a specific part of the system. This ensures both reliability of individual components and correctness of the whole workflow when all microservices interact together. In order to validate the whole system we execute each test with all docker infrastructure in up state. Ensuring that all work fine in every condition. At this scope we use differents technologies: *JUnit* for Unit Test, Integration and Component Test and *Gerchin* to simulate entire end to end test.

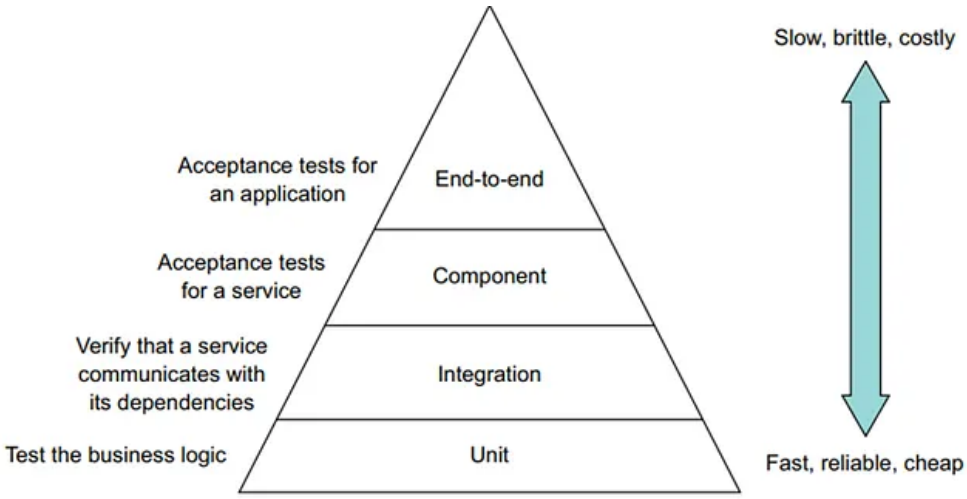

Figure 27: Testing Pyramid adopted in the project

6.1 JUnit Test & Integration Test

Unit testing represents the foundation of the validation process. **JUnit tests** were used extensively to verify the correctness of the core business logic inside each microservice. Since the architecture follows the Hexagonal pattern, it was possible to test the domain logic in isolation without the need to start the whole service or connect to external resources.

These tests confirmed that each microservice appeared internally coherent and that its input/output behaviour matched the system requirements as shown in Figure 28 with the integration test for the Ticketing-service.

```java
class JUnitTicketingServiceTest {

    @Test  davidedimarco00
    @Order(1)
    void testCreateTicket() {
        TicketMessage request = new TicketMessage();
        request.setAssociatedWithPlate(PLATE);
        request.setExitDetected(false);

        ResponseEntity<TicketMessage> resp = rest.postForEntity(
                url: baseUrl + "/createTicket",
                request,
                TicketMessage.class
        );

        assertEquals(HttpStatus.OK, resp.getStatusCode());
        assertNotNull(resp.getBody());
        assertEquals(PLATE, resp.getBody().getAssociatedWithPlate());
        assertEquals( expected: "IN_PROGRESS", resp.getBody().getStatus());

        createdTicketId = resp.getBody().getId();
        assertNotNull(createdTicketId);

        System.out.println("✅ Created ticket ID: " + createdTicketId);
    }

    @Test  davidedimarco00
    @Order(2)
    void testGetTicketByPlate() {
        ResponseEntity<String> resp = rest.getForEntity(
                url: baseUrl + "/getTicketByPlate/" + PLATE,
                String.class
        );

        assertEquals(HttpStatus.OK, resp.getStatusCode());
        assertTrue(resp.getBody().contains(PLATE));
        assertTrue(resp.getBody().contains("IN_PROGRESS"));

        System.out.println("✅ Ticket fetched successfully");
    }
```

Figure 28: Example of JUnit Test Methodology

6.2 Component Test

In addition to integration testing, we executed component-level validations to test individual microservices running inside Docker. To check the integrity and consistency of the entire system we check the state autonomously and manually inside mongodb after each test. For manual test *Mongo Compass Suite* has helped us in Figure 29.

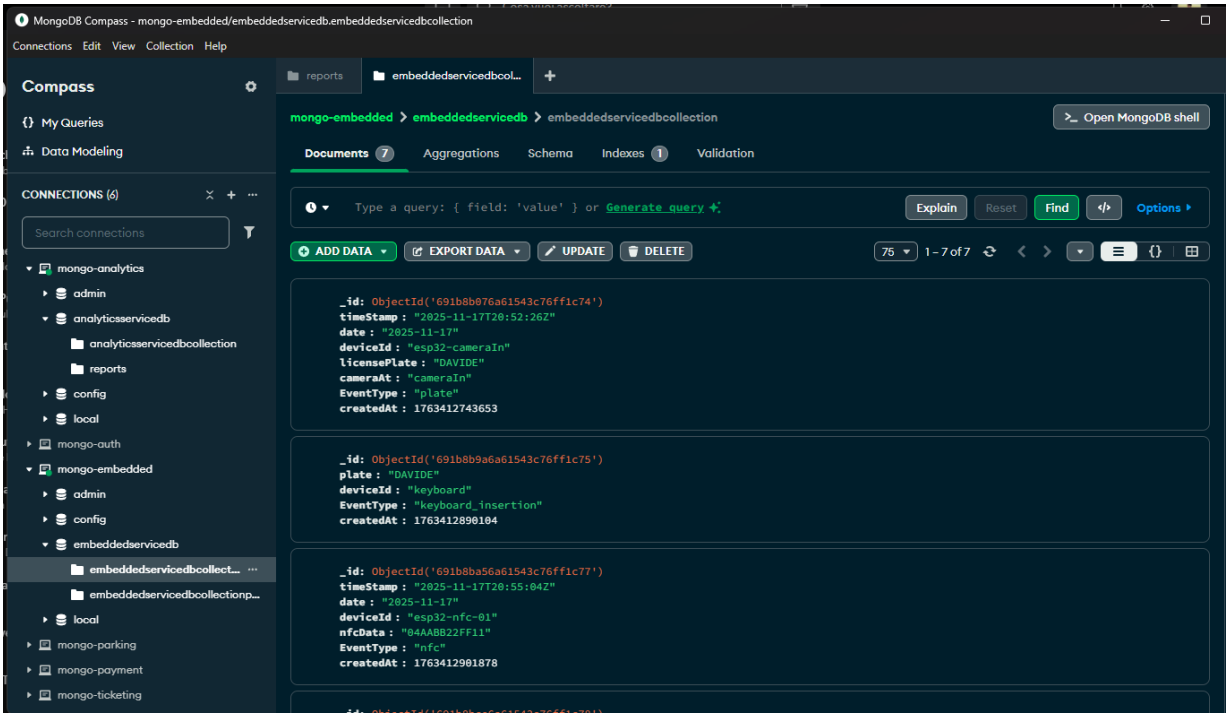

Figure 29: Mongo Compass View

6.3 End to End (e2e) Test

The end-to-end phase validates the behaviour of the entire Smart Parking system. All microservices, MongoDB databases, the Eureka Discovery Server, the API Gateway and the HiveMQ MQTT broker were deployed through Docker Compose, enabling realistic validation of inter-service communication, MQTT event ingestion and the full domain workflow.

Two main scenarios were tested using Gherkin specifications. The first scenario concerns the **Admin Parking Management Flow** shown in Figure 30.

```
Feature: Admin parking management flow

  Scenario: Admin login and token retrieval
    Given The admin sends a POST request to auth-service with credentials
    When The system register the authentication
    Then The admin can access protected endpoints

  Scenario: Admin sets the daily tariff
    Given The admin is authenticated
    When The admin calls POST payment-service with hourFee, dailyFee, and thresholdHours
    Then The system updates the tariff configuration

  Scenario: Admin generates the previous day's report
    Given The admin is authenticated
    When The admin calls GET analytics-service to generate reports for the previous day
    Then The system generates or retrieves the report

  Scenario: Admin manually adds a car for entry
    Given The admin is authenticated
    When The admin calls POST parking-service to addCar with the customer's license plate
    Then The system adds the car to the parking lot

  Scenario: Admin assists customer exit after payment verification
    Given The admin is authenticated
    When The admin calls GET getAllTickets to view active tickets
    And The admin calls GET getAllHistoryTickets to check payment status
    And If payment is confirmed, the admin calls POST openExitBarrier
    Then The system opens the exit barrier for the customer
```

Figure 30: Admin Parking Management Flow

The second scenario in Figure 31 describes the **Standard Customer Parking Flow**.

Both scenarios were executed successfully, demonstrating that the distributed architecture behaves correctly as a unified system and that all core domain workflows are reliable under production conditions.

```
Feature: Standard customer parking flow

  Scenario: Customer parks, pays and exits without administrator intervention
    Given The car with license plate "TEST" arrives at the parking lot
    When The system reads the license plate and calls the addCar endpoint of Parking Management
    And The system creates and saves the virtual ticket associated with the license plate in the database
    And The system calls the openEntryBarrier endpoint of Embedded to open the entry gate
    Then The car enters the parking lot

    Given The car is inside the parking lot
    When The virtual ticket is active and saved in the database
    Then The customer parks without issues

    Given The customer returns to the cashier and enters the license plate "TEST"
    When The system calls the calculateFee endpoint of Payment Service with the license plate
    And Payment Service calls Ticketing Service to get the associated virtual ticket
    And Payment Service calls getTariff to get the hourly rate
    Then The system calculates the total amount to pay and shows it to the customer

    Given The customer pays the calculated amount
    When The system calls the processPayment with nfcTag endpoint of Payment Service
    Then The payment is confirmed and the ticket is updated

    Given The customer heads to the exit
    When Embedded Service reads the license plate "TEST" at the exit
    And Embedded Service calls the removeCar endpoint of Parking Management
    And Embedded Service calls openExitBarrier to open the exit gate
    Then The customer leaves the parking lot without administrator intervention
```

Figure 31: Customer Test Flow

6.4 Embedded Simulator (Sensor-Communication Validation)

To emulate the physical world, a dedicated **Embedded Simulator** was developed in Python and shown in Figure 32. This tool reproduces the behaviour of the physical devices installed in the parking infrastructure.

It publishes MQTT events including:

- Coil detection events
- Plate recognition data
- NFC payment card scans
- help button activations
- device status updates

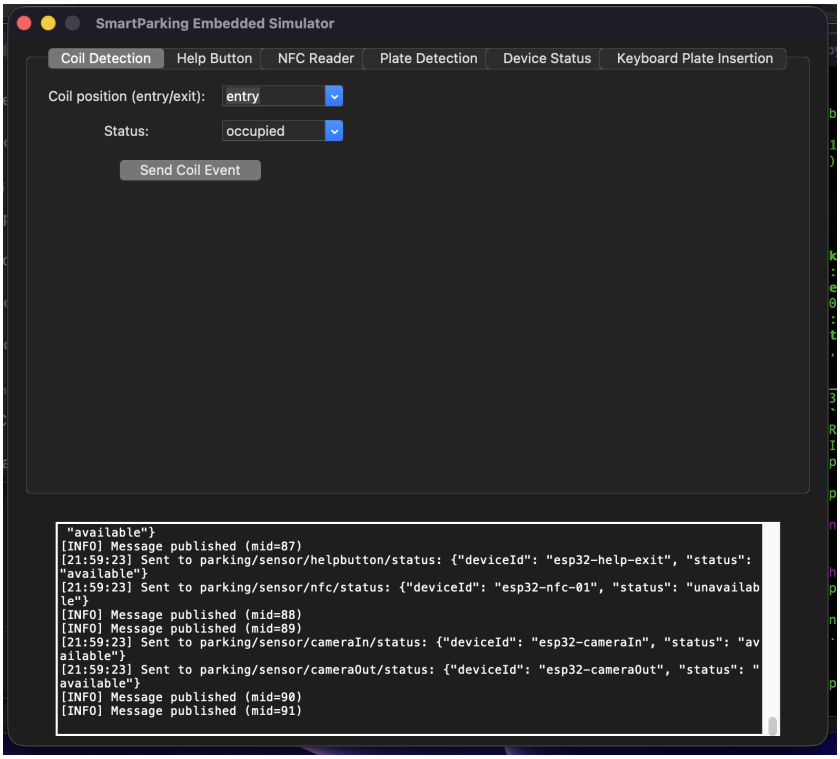

Figure 32: Embedded Simulator

6.5 CI/CD Pipeline

To ensure continuous reliability, the entire project was integrated into a **CI/CD pipeline** directly in GitLab Environment; see Figure 33. The pipeline executed is:

- docker build;
- docker deploy;
- end to end test;

This pipeline ensured that every commit produced a fully validated and deployable system.

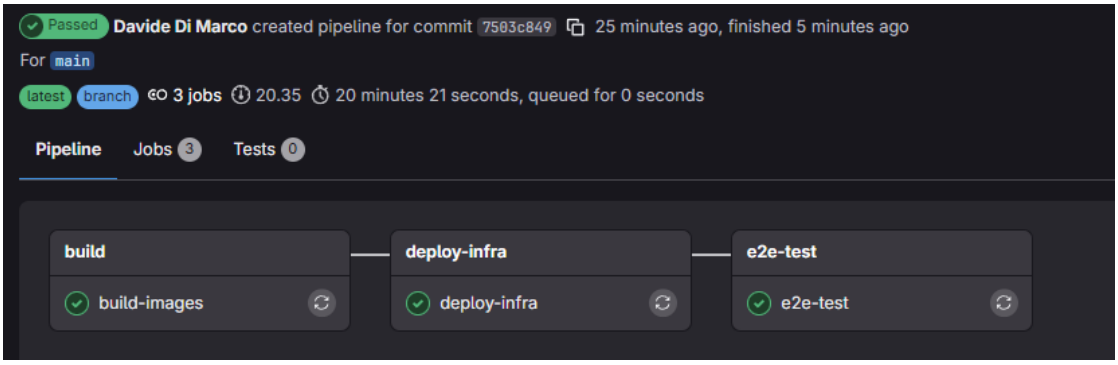

Figure 33: CI/CD adopted in the project

7 Conclusions

The Smart Parking project successfully achieved its initial goal: designing and implementing an intelligent parking system that integrates IoT devices with a microservices architecture based on Domain-Driven Design principles.

The decomposition of the system into bounded contexts guaranties the separation of concerns, reducing complexity, and improving maintainability, where each microservice operates independently and access to other services through API following an exagonal architecture. To simplify the use of the endpoint and the API, It was necessary to introduce a single access point for every service; made with the adoption of an API Gateway, service discovery, and observability tools. Through the combined use of MQTT, Vert.x, and an internal Event Bus, the Embedded-Service effectively transforms raw IoT signals into structured domain actions. This establishes a robust event-driven backbone for the Smart Parking system, enabling fast, reactive, and resilient interaction with the physical world.

In conclusion, the project validates the effectiveness of combining Domain-Driven Design with microservices and Event-Driven-Architecture for building robust, maintainable, and scalable software applications.

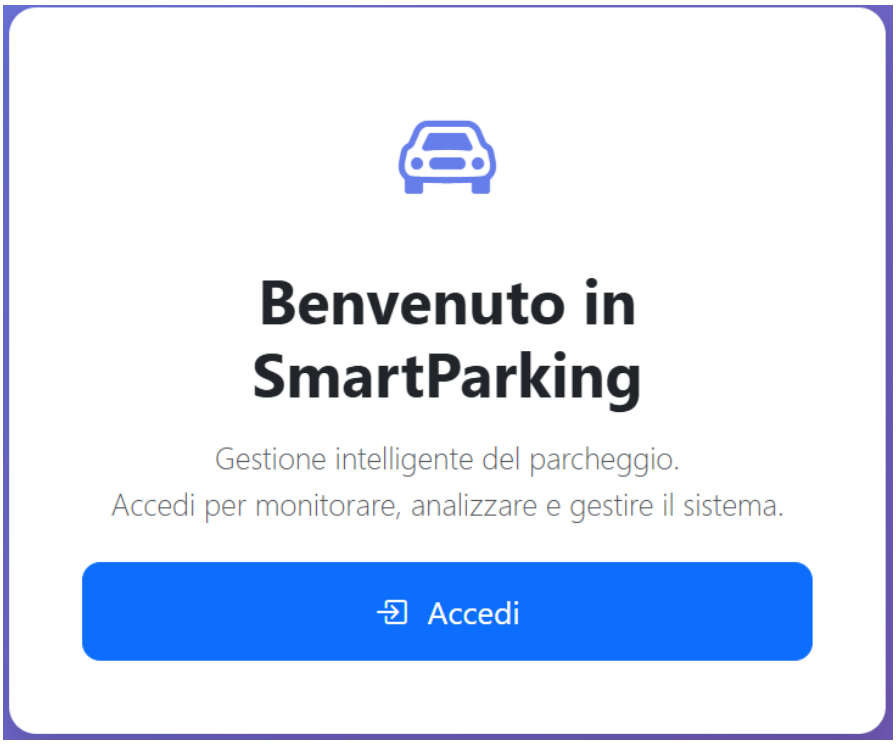

Figure 34: Welcome on the Smart Parking