Computer Science Department
Software Engineering

**Bachelor's Thesis**

# Design and implementation of a cloud platform for parking space rent and access control

## Role-based access, device integration, and end-to-end validation

Artem Nazarenko

*Supervisor*
Kyiv School of Economics, Vadym Yaremenko, PhD, vyaremenko@kse.org.ua

*Expert*
Expert, Heorhii Sandatsian, sand.hs350@gmail.com

*Submission date*
03 June 2026

# Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date Kyiv, 04.06.2026

Signature _____

# Contents

Acknowledgements ..... 1
Abstract ..... 2
1 Introduction ..... 4
1.1 Problem statement ..... 4
1.2 Relevance and significance ..... 4
1.3 Objective and scope ..... 4
1.4 Methodology and engineering approach ..... 5
1.5 Structure of the thesis ..... 5
2 Research ..... 6
2.1 Domain context ..... 6
2.2 Existing classes of solutions ..... 6
2.2.1 Manual and locally managed access ..... 6
2.2.2 Booking-oriented digital platforms ..... 7
2.2.3 Device-centric access control systems ..... 7
2.3 Authorization and operational requirements of the domain ..... 7
2.4 Comparative gap analysis ..... 8
2.5 Derived system requirements ..... 8
2.5.1 Functional requirements ..... 8
2.5.2 Non-functional requirements ..... 9
2.6 Chapter summary ..... 10
3 Design ..... 11
3.1 Requirements-driven architecture ..... 11
3.2 Architectural overview ..... 11
3.3 Major design decisions ..... 12
3.3.1 Separation between domain platform and device integration service ..... 12
3.3.2 Capability-based account model ..... 12
3.3.3 Persistent account and session architecture ..... 13
3.3.4 Cloud deployment strategy ..... 13
3.4 Data model and authorization context ..... 13
3.5 Technology stack and deployment ..... 14
3.6 Cross-cutting concerns ..... 16
3.6.1 Security model ..... 16
3.6.2 Reliability and observability ..... 16
3.6.3 Scalability limits and operational trade-offs ..... 16
3.7 Chapter summary ..... 16
4 Implementation ..... 18
4.1 Development approach ..... 18
4.2 Backend API and service structure ..... 18
4.2.1 Session refresh and authorization context ..... 19
4.3 Device integration service and control path ..... 19
4.3.1 Device lookup and command dispatch ..... 20
4.4 Frontend web application ..... 20

4.5 Authentication and email verification ..... 21
4.6 Operator and administrator workflow support ..... 22
4.7 Infrastructure as code and deployment ..... 23
4.8 Persistence, migration, and state derivation ..... 24
4.9 Configuration and secret propagation ..... 25
4.10 Operational UI refinements informed by testing ..... 25
4.11 Selected pseudocode excerpts ..... 25
4.11.1 Rent duration classification ..... 26
4.11.2 Lock and unlock authorization ..... 26
4.11.3 Device state refresh from events ..... 26
4.11.4 Owner assignment and inventory enablement ..... 27
4.12 Chapter summary ..... 27
5 Validation ..... 28
5.1 Validation environment ..... 28
5.2 Validation methodology ..... 28
5.3 End-to-end scenario execution ..... 29
5.3.1 Administrative bootstrap and inventory setup ..... 29
5.3.2 Device registration and assignment ..... 29
5.3.3 Owner onboarding and ownership assignment ..... 30
5.3.4 Customer onboarding, rent creation, and device control ..... 30
5.4 Validation matrix ..... 30
5.5 Requirement coverage discussion ..... 31
5.6 Failures encountered and engineering corrections ..... 31
5.6.1 Stale online state after abrupt device loss ..... 31
5.6.2 SES delivery failure in deployed runtime ..... 31
5.6.3 Mixed-capability authorization error ..... 32
5.6.4 Misleading frontend operational state ..... 32
5.7 Operational observations from integrated testing ..... 32
5.8 Threats to validity ..... 33
5.9 Current limitations ..... 33
5.10 Validation summary ..... 34
6 Conclusion ..... 35
6.1 Project summary ..... 35
6.2 Alignment with objectives ..... 35
6.3 Engineering lessons learned ..... 35
6.4 Limitations ..... 35
6.5 Future work ..... 35

# Acknowledgements

This draft keeps the acknowledgements intentionally brief. A final version of this section should thank the academic supervisor, reviewers, colleagues who supported the engineering process, and individuals who enabled access to the hardware and cloud environment used during validation.

# Abstract

Private and semi-private parking spaces are often managed through fragmented combinations of messaging, manual billing, physical keys, or standalone remote controls. Such approaches create operational friction for owners, weak auditability for administrators, and poor access guarantees for customers who pay for temporary use. These problems become more pronounced when physical access is mediated by connected electromechanical devices that must remain synchronized with digital permissions.

This thesis presents the design and implementation of a cloud platform for parking space rent and access control built around connected parking lock devices. The system combines a C++ backend API, a dedicated device gateway, a Flutter web application, PostgreSQL persistence, and AWS-based deployment. The platform supports multiple actor types, namely administrators, installers, owners, customers, and service personnel. It provides email-based user onboarding, capability-based authorization, space and device provisioning workflows, rent management, and controlled lock and unlock operations against a live device.

A major design goal of the system is to bind digital business actions to physical device control without collapsing all concerns into one service. To achieve this, the implementation separates domain-facing platform logic from the hardware-integration boundary. The backend API validates identity, role, ownership, rent state, and device assignment before forwarding commands to the gateway. The gateway maintains the active communication context required for reliable device control and reports resulting state back into the platform. This separation simplifies reasoning about authorization, improves operational visibility, and preserves hardware-specific behavior in a dedicated runtime component.

The implemented prototype was validated through an end-to-end scenario on real infrastructure. The scenario covered bootstrap administration, group and space creation, device registration and assignment, coordinate and pricing configuration, owner promotion, customer registration with email verification, rent creation, and lock and unlock operations performed by the active renter. During validation, several defects were discovered and corrected, including stale online state handling, authorization edge cases for mixed-capability accounts, and cloud email delivery integration issues. These corrections were treated as engineering findings rather than exceptional incidents and became part of the validated design.

The main contribution of this work is a coherent platform architecture that connects user-facing parking workflows, cloud identity management, and physical device control into one auditable system. The resulting prototype demonstrates that role-based access management, rent-driven authorization, and real hardware operation can be integrated into a practical parking access platform while remaining academically defensible as a software engineering project.

***Keywords***:

*parking access control, parking space rent, cloud platform, connected parking lock devices, role-based authorization, Flutter, AWS*

# 1 Introduction

## 1.1 Problem statement

Access to private parking spaces is frequently managed through arrangements that were never designed as integrated systems. In many cases, discovery of available spaces, agreement on price, user identification, physical access, and exception handling are distributed across messaging applications, spreadsheets, payment screenshots, local remote controls, and verbal agreements. This fragmentation makes the process inconvenient for customers, difficult to administer at scale, and error-prone whenever physical access must be synchronized with a paid time interval.

The problem is not limited to user experience. Once an electromechanical parking lock is introduced, digital permissions and physical state become coupled. A customer who has paid for a valid interval should be able to open the assigned device during the active rent period, while another customer should not. An owner should be able to manage pricing and visibility of their spaces without receiving unrestricted privileges over unrelated inventory. Installers need tools for registering and assigning devices, and administrators need a reliable path to bootstrap and maintain the platform. These needs suggest a systems problem rather than a single-interface problem.

## 1.2 Relevance and significance

The topic is relevant for software engineering because it sits at the intersection of cloud services, embedded-device integration, identity management, role-based authorization, and operational deployment. Unlike purely informational applications, the system studied in this thesis interacts with physical devices whose state can diverge from expectations due to connectivity loss, stale telemetry, or integration-level faults. This creates a useful academic context for discussing architecture, cross-cutting concerns, and validation against real constraints.

The project is also relevant from a broader practical perspective. Parking access control is a domain where modest amounts of automation can significantly improve reliability, traceability, and service quality. A functioning platform must reconcile human workflows such as registration, ownership assignment, and customer rent creation with lower-level concerns such as device-state synchronization, communication reliability, and cloud-hosted secrets. This makes the system a suitable capstone topic because the engineering complexity is real but still tractable within a bachelor-level scope.

## 1.3 Objective and scope

The objective of this thesis is to design, implement, and validate a role-based cloud platform that manages parking space inventory together with rent and access control according to user permissions and active rent periods.

The scope of the work includes:

- account registration, email verification, login, session persistence, and password reset;

- capability-based actor modeling for administrators, installers, owners, customers, and service users;
- creation and management of parking groups and spaces;
- registration and assignment of connected parking lock devices to parking spaces;
- pricing and visibility configuration for owner-managed inventory;
- customer rent creation and enforcement of device access based on active rent state;
- AWS deployment of the main platform services and database-backed persistence.

The work deliberately excludes several concerns from the current implementation scope:

- native mobile packaging beyond the web client path;
- payment processor integration and accounting-grade billing workflows;
- advanced security hardening such as MFA, production TLS termination policy, and device certificate enrollment;
- advanced high-availability and failover strategies for the hardware-integration boundary.

## 1.4 Methodology and engineering approach

The project followed an iterative engineering methodology. Initial work focused on establishing a minimal end-to-end vertical slice: a connected device, a backend service able to address that device, and a client capable of sending user actions. After this initial path was working, the implementation was expanded to cover persistent identity, role separation, owner workflows, inventory setup, and operator tooling.

Validation was scenario-driven rather than purely unit-driven. The system was repeatedly exercised through realistic workflows such as registering a customer, assigning a device to a space, creating a rent, and verifying that only the active renter or an authorized staff member could issue unlock and lock commands. Failures encountered during this process were treated as feedback for architectural and implementation refinement.

## 1.5 Structure of the thesis

The remainder of this thesis is organized as follows. Chapter 2 examines the problem domain, compares classes of existing approaches, and derives system requirements. Chapter 3 presents the architecture and design decisions, including the role model, data model, deployment topology, and major trade-offs. Chapter 4 explains how the design was implemented in the backend, device gateway, frontend, authentication flows, and operational tooling. Chapter 5 validates the system through an end-to-end scenario executed on the deployed environment and documents important failures and corrections. Chapter 6 concludes the thesis by summarizing results, limitations, and future work.

# 2 Research

## 2.1 Domain context

Parking access control becomes a software engineering problem when a parking space is no longer granted by physical presence, private agreement, or a handed-over remote control, but by digitally managed entitlement linked to a physical access mechanism. In such a setting, the system must coordinate identity, ownership, temporary usage rights, operational administration, and physical enforcement. The difficulty lies not only in listing available spaces, but in maintaining a trustworthy connection between digital state and the real access conditions of a specific parking space.

The target domain therefore combines two different kinds of coordination. The first is organizational: who owns a space, who may configure it, who may rent it, and who may intervene in exceptional situations. The second is technical: how the system translates those relationships into a reliable control path for a connected electromechanical parking lock. Any platform developed for this domain must address both dimensions simultaneously.

The main stakeholder groups are administrators, installers, owners, customers, and service personnel. Administrators govern the overall platform and bootstrap inventory. Installers introduce and attach hardware to configured spaces. Owners define the commercial and operational parameters of the spaces assigned to them. Customers interact with the platform in order to discover, rent, and temporarily use a space. Service personnel require limited operational access for troubleshooting or maintenance. These actors interact through one domain object, namely the parking space, but each of them does so under different rights and responsibilities.

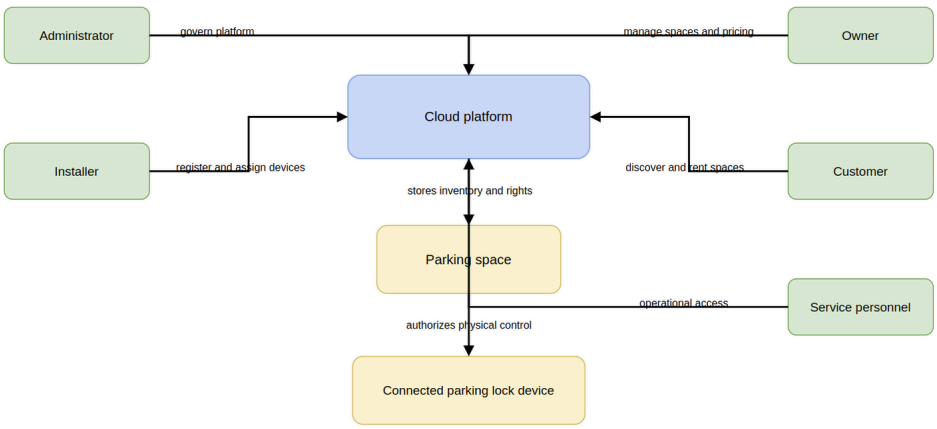


Figure 1: Stakeholders and control relationships in the parking access domain.

## 2.2 Existing classes of solutions

### 2.2.1 Manual and locally managed access

The most common baseline is manual coordination. In this arrangement, availability is communicated directly between participants, price is negotiated or known implicitly, and

access is granted using physical remotes, keys, or local assistance. This class of solution performs adequately for a small number of trusted users because it minimizes technical overhead. It also preserves flexibility when exceptions occur.

Its limitations become clear once repeatability and accountability matter. Manual coordination does not provide a durable identity layer, structured ownership management, or auditable records of who was allowed to access a space and when. It also does not provide a dependable way to bind a paid usage interval to the actual operation of a connected parking lock. For the problem addressed in this thesis, this approach is therefore insufficient.

### 2.2.2 Booking-oriented digital platforms

A second class of solutions emphasizes digital discovery and reservation workflows. Such platforms typically support searchable inventory, time-bound booking, and in some cases pricing and payment support. They solve the user-facing problem more effectively than manual coordination, particularly when demand spans a larger user base or multiple owners.

However, these systems often treat physical access as an external concern. Access may remain open, be handled by staff, or rely on an out-of-band mechanism not directly governed by the booking platform. As a result, they are well suited for inventory exposure and reservation management, but they do not necessarily solve the problem of synchronizing authorization with a specific connected parking lock.

### 2.2.3 Device-centric access control systems

A third class of solutions begins from the physical control device rather than from the broader platform workflow. Such systems often provide onboarding utilities, telemetry, and control capabilities for connected barriers or parking locks. Their strength lies in direct interaction with field hardware and operational visibility into device state.

Their weakness is usually the surrounding domain model. A device-centric system may offer control over hardware while under-specifying long-term ownership, customer-facing onboarding, or rent-aware authorization. In other words, it may solve the control problem without solving the broader platform problem in which different actors need distinct rights over the same space.

## 2.3 Authorization and operational requirements of the domain

The domain imposes an additional authorization constraint beyond simple user authentication. In an ordinary web application, access control may be determined largely by the role attached to the current account. In this domain, that is not enough. A valid decision also depends on the current relationship between the account, the parking space, and the relevant usage interval.

An owner should be able to manage only the spaces assigned to that owner. A customer should be able to operate a connected parking lock only when the customer holds a currently active rent period for the corresponding space. Installers and service personnel require operational privileges that differ from both ownership and customer usage. The

platform must therefore combine capability-based identity with persisted domain relationships and temporal checks.

Another domain-specific requirement is that the physical access path must remain coupled to the current space assignment of a device. It is not enough to know that a command is authorized in general terms; the system must also resolve which space is being targeted and whether the connected lock device currently associated with that space is the correct physical endpoint. This makes contextual authorization a first-class architectural concern.

## 2.4 Comparative gap analysis

The three classes of solutions considered above each address part of the problem but do not fully satisfy the needs of the target system. Manual approaches provide flexibility but lack digital traceability and enforceable access rules. Booking-oriented platforms improve discoverability and usage scheduling but often stop at the boundary of physical control. Device-centric systems provide control over hardware but may not model ownership, customer onboarding, and rent-based authorization with sufficient richness.

The gap, therefore, is not the absence of isolated features. It is the absence of a coherent platform that combines persistent identity, multi-actor governance, space inventory management, context-aware authorization, and physical access enforcement in one system.

| **Approach** | **Digital onboarding** | **Physical access integration** | **Role separation** | **Time-bound authorization** | **Operational traceability** |
|---|---|---|---|---|---|
| Manual coordination | Low | External or ad hoc | Low | No | Low |
| Booking-oriented platforms | High | Partial or indirect | Medium | Partial | Medium |
| Device-centric systems | Low to medium | High | Low to medium | Rare | High |
| The system developed in this thesis | High | High | High | Yes | High |

Table 1: Comparison of common parking access management approaches against the requirements of the target system.

## 2.5 Derived system requirements

### 2.5.1 Functional requirements

The domain analysis yields the following functional requirements for the system developed in this thesis.

- FR1: the system shall support persistent account registration, email verification, login, session restoration, and password reset.
- FR2: the system shall represent multiple actor capabilities within one account rather than forcing one account per role.

- FR3: the system shall support creation and management of parking groups and spaces by authorized staff.
- FR4: the system shall support registration of connected parking lock devices and assignment of those devices to configured spaces.
- FR5: the system shall allow owners to manage the commercial and operational parameters of spaces assigned to them.
- FR6: the system shall allow customers to create valid rent periods for accessible spaces.
- FR7: the system shall restrict physical lock and unlock operations to authorized actors, including the active renter of a space during a valid rent period.
- FR8: the system shall persist and expose device state and event information for operational inspection.

### 2.5.2 Non-functional requirements

The same analysis also leads to the following non-functional requirements.

- NFR1: the platform shall preserve consistent enough alignment between digital entitlement and physical access control for operational use.
- NFR2: the deployment process shall be reproducible in a cloud environment.
- NFR3: important state transitions shall be persisted in a form suitable for auditing and troubleshooting.
- NFR4: hardware-facing communication concerns shall remain isolated from general domain logic.
- NFR5: the user-facing application shall support both operational setup and customer workflows through a coherent interface.
- NFR6: the architecture shall remain extensible enough to support future billing, security, and reporting enhancements.

| ID | Requirement | Motivation | Addressed in |
|---|---|---|---|
| FR1 | Persistent account onboarding | The platform needs auditable, recoverable identity | Chapters 3-5 |
| FR3-FR4 | Inventory and device provisioning | Spaces and devices must be created and linked explicitly | Chapters 3-5 |
| FR5-FR7 | Ownership and rent-aware authorization | Access depends on actor relationship and current rent state | Chapters 3-5 |
| NFR2 | Reproducible cloud deployment | Validation requires a realistic managed environment | Chapters 3-5 |
| NFR4 | Isolation of device integration concerns | Device communication must not dominate domain-layer design | Chapters 3-4 |

Table 2: Consolidated functional and non-functional requirements derived from the domain analysis.

## 2.6 Chapter summary

The domain analysis shows that the central challenge is not merely booking parking spaces and not merely controlling connected hardware. Rather, the challenge is to construct a platform in which identity, ownership, temporary usage rights, and physical access remain aligned under one operational model. These requirements directly motivate the architectural decisions presented in the next chapter.

# 3 Design

## 3.1 Requirements-driven architecture

The architecture of the system developed in this thesis follows directly from the requirements derived in Chapter 2. The platform must support several actor types, persist inventory and commercial state, authorize operations against contextual relationships, and maintain a reliable control path to a connected parking lock device. These requirements impose an architecture in which digital business logic and device integration remain closely coordinated but structurally separated.

Four architectural pressures were especially influential. First, the platform requires a conventional domain service capable of handling identity, inventory, ownership, and rent management. Second, the system requires a dedicated integration boundary for stateful communication with connected lock devices. Third, authorization decisions must be resolved against persisted domain relationships rather than role names alone. Fourth, the platform must be deployable in a reproducible cloud environment.

## 3.2 Architectural overview

At a high level, the platform consists of a user-facing web application, a backend API that owns domain state and authorization, a device integration service that mediates control of connected parking lock devices, a relational database, and supporting cloud services for email delivery and deployment. This decomposition ensures that user workflows remain understandable at the platform level while hardware-facing behavior remains confined to a service designed for that purpose.

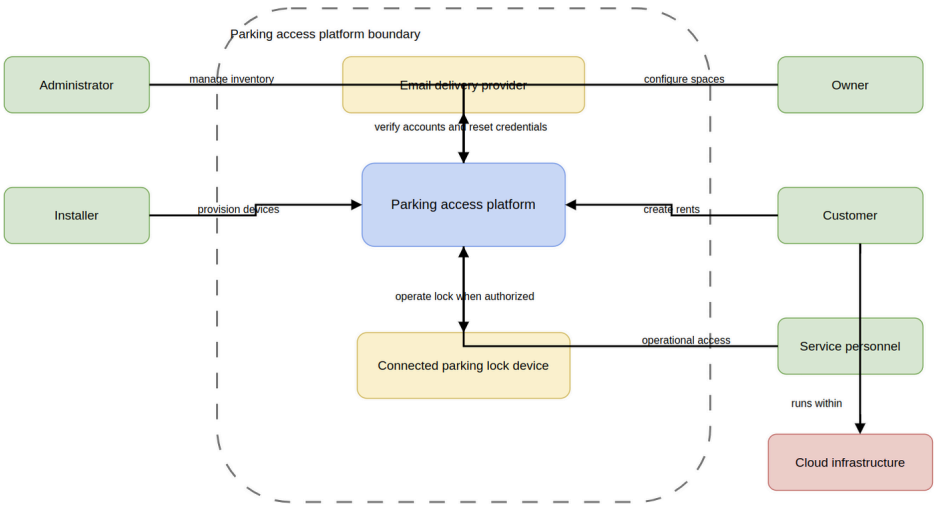


Figure 2: System context of the parking access platform.

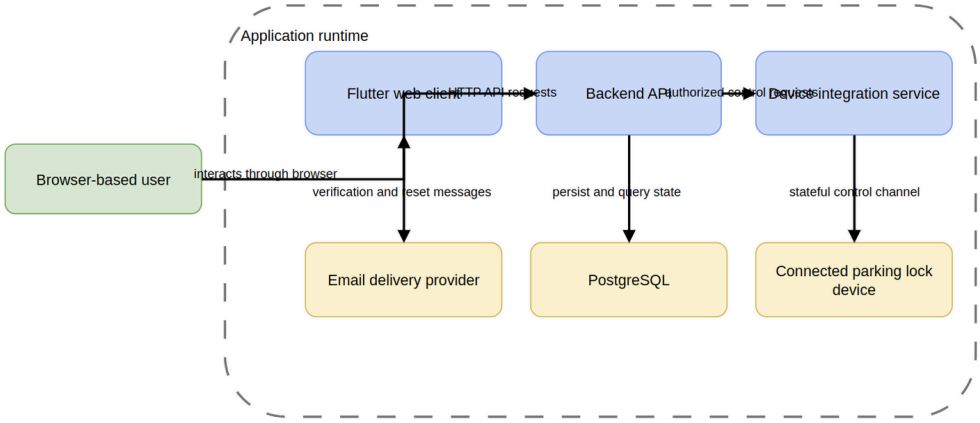


Figure 3: Container-level architecture of the parking access platform.

The web client provides the interaction surface for customers, owners, installers, service users, and administrators. The backend API persists and governs the domain model. The device integration service translates platform-level control requests into operations against the connected device and reports resulting state changes back to the domain platform. The database acts as the source of truth for accounts, inventory, assignment, pricing, and operational events. This separation reflects the fact that the system spans both ordinary web-application concerns and physically consequential control actions.

## 3.3 Major design decisions

### 3.3.1 Separation between domain platform and device integration service

The platform does not embed hardware-facing communication directly inside the main backend. Instead, it uses a separate device integration service that is responsible for maintaining a persistent device session and executing control actions on behalf of the domain platform.

The main alternative would have been to place all concerns into a single backend process. That option was rejected because it would blur the boundary between user-facing business rules and stateful hardware integration, making the system harder to reason about, scale, and troubleshoot.

The adopted design yields a clearer separation of responsibilities. The backend API remains the policy enforcement point for identity, ownership, and rent validation, while the integration service remains the execution boundary for device control. The trade-off is additional inter-service coordination and the need for a carefully defined internal command path.

### 3.3.2 Capability-based account model

The account model allows one account to hold multiple capabilities, such as customer and owner simultaneously. This decision reflects the real structure of the domain, where the same person may manage some spaces while renting others.

A simpler single-role model was rejected because it would either fragment identity across multiple accounts or require brittle role switching semantics. The capability-based model provides a cleaner representation of user identity while allowing the actual scope of rights

to be constrained by database-backed relationships. Its trade-off is a more sophisticated authorization layer, because capabilities must be evaluated together with ownership and rent context.

### 3.3.3 Persistent account and session architecture

The platform uses persistent email-based onboarding together with short-lived access tokens and longer-lived refresh sessions. This design supports realistic account lifecycle behavior, including verification, reset, logout, and re-authentication across devices.

A purely sessionless or purely manually seeded identity model was rejected because it would not support the desired onboarding flow or the later promotion of a customer account to owner capability. The chosen design creates a stronger foundation for future hardening while preserving a manageable prototype scope. The trade-off is a larger persistence surface, including session and token-related tables.

### 3.3.4 Cloud deployment strategy

The system is deployed on AWS using containerized services, a managed relational database, and cloud-based email delivery. This decision was motivated by the need for reproducibility, realistic validation, and operational separation between the main application runtime and its supporting infrastructure.

A purely local or single-host deployment would have been simpler, but it would not have exercised important failure modes related to secrets, email delivery, service restart behavior, and cloud-hosted persistence. The chosen strategy improves realism and demonstrates deployment maturity. Its trade-off is higher operational complexity during setup and testing.

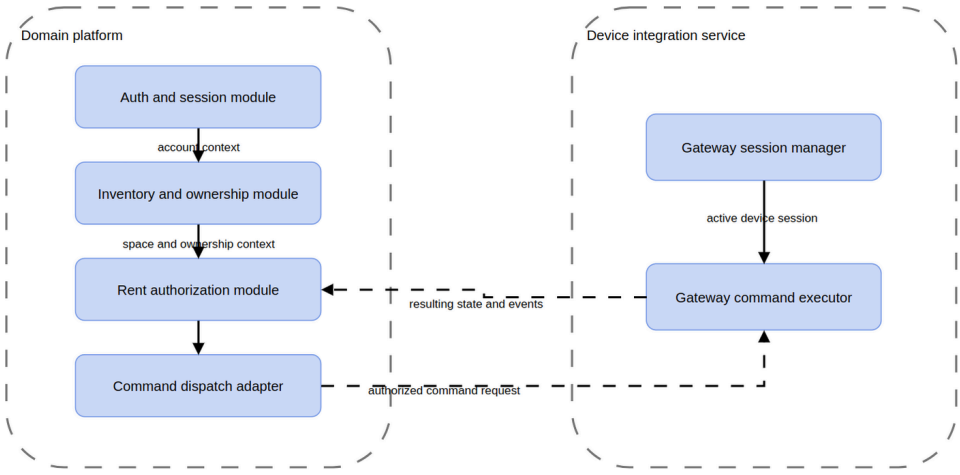


Figure 4: Responsibility split between domain services and device integration services.

## 3.4 Data model and authorization context

The persistence model is organized so that authorization decisions can be resolved against stable domain facts. Identity is represented through accounts, credentials, capabilities, and session-related records. Inventory is represented through parking groups and spaces. Connected lock devices are modeled separately so they can be registered, assigned, and

observed independently from user identities. Rent periods are persisted in reservation records because they define time-bounded customer rights. Operational visibility is supported through event storage and related operational entities.

This structure matters because the platform must resolve questions such as whether a particular account currently owns a space, whether the account is the active renter of that space, and which connected device is currently assigned to it. Authorization is therefore not reduced to token claims alone; it depends on persisted relationships within the domain model.

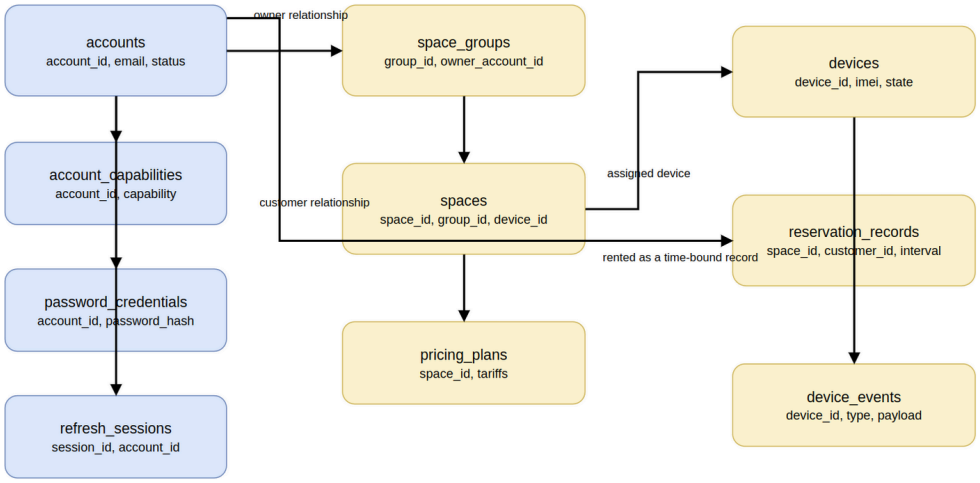


Figure 5: Core persistence model for identity, inventory, device, and rent management.

## 3.5 Technology stack and deployment

The technology stack was selected to support both conventional platform behavior and physically grounded device integration. The backend API and device integration service were implemented in C++ to preserve close control over performance, networking behavior, and system-level integration. The frontend was implemented in Flutter to provide a web-first user interface with a path toward broader client reuse. PostgreSQL was selected as the authoritative transactional store. AWS was selected to provide a realistic environment for container orchestration, database hosting, image management, and email delivery.

| Layer | Selected technology | Role | Selection summary |
|---|---|---|---|
| Web client | Flutter | User-facing interaction layer | Shared web-first workflow; browser constraints remain relevant. |
| Domain platform | C++20 with Drogon | Identity, policy, inventory, and rent management | Fine control over HTTP and integration boundaries at the cost of higher implementation complexity. |
| Device integration service | Custom C++20 service | Execution boundary for device control | Stable handling of persistent live sessions, with specialized service logic as the trade-off. |
| Persistence | PostgreSQL | System source of truth | Transactional consistency and flexible relations, with deliberate schema management required over time. |

Table 3: Core implementation technologies used in the platform architecture.

The table above covers the technologies most directly responsible for application behavior. Deployment support technologies deserve separate mention because they contribute to reproducibility and validation quality even though they are not part of the day-to-day domain model. Kubernetes provides the workload abstraction used to run the backend API and the device integration service, Terraform provides repeatable provisioning of cloud-side dependencies, and AWS-managed services reduce the amount of locally improvised infrastructure that would otherwise weaken the realism of the evaluation.

| Support layer | Selected technology | Role | Selection summary |
|---|---|---|---|
| Container orchestration | Kubernetes on AWS EKS | Run isolated service workloads | Repeatable deployment and restart semantics, with cluster configuration overhead. |
| Infrastructure provisioning | Terraform | Create cloud-side dependencies | Declarative and auditable environment setup, with additional operational tooling to maintain. |
| Image distribution | AWS ECR | Store deployable container images | Native fit with the selected runtime, while registry lifecycle still has to be managed. |
| Email transport | AWS SES | Deliver verification and reset email | Managed provider with realistic runtime constraints and cloud-side identity setup requirements. |

Table 4: Deployment support technologies that improve reproducibility and validation realism.

The validated prototype was deployed in a dedicated Kubernetes namespace. The backend API and the device integration service were deployed as separate workloads, while the relational database was hosted externally as a managed RDS instance. Container images were stored in ECR, and outbound account verification and reset emails were delivered through SES. Secrets were injected through Kubernetes-managed configuration rather than hardcoded into application images.

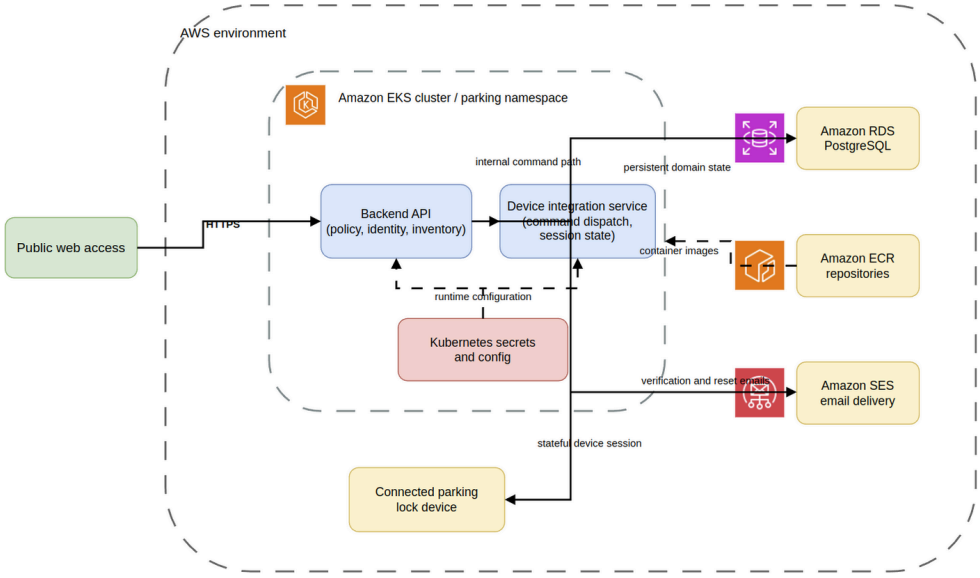


Figure 6: Cloud deployment topology used for the validated prototype.

## 3.6 Cross-cutting concerns

### 3.6.1 Security model

The current security model combines persistent account onboarding, email verification, password reset, capability-aware authorization, and contextual validation of space ownership or rent state. This does not yet constitute a fully hardened production posture, but it establishes the correct architectural boundary for future additions such as stricter transport security, stronger secret management, and multi-factor authentication.

### 3.6.2 Reliability and observability

The platform persists important state transitions and device-related events in PostgreSQL, providing a traceable operational history. Metrics endpoints are exposed by the main services, and operator tooling supports event inspection and repeatable setup. This combination improves diagnosability, particularly in a system where user-facing failures may originate from identity, state, deployment, or device-integration layers.

### 3.6.3 Scalability limits and operational trade-offs

The two primary services do not scale in the same way. The domain platform can be replicated more freely because its state is persisted externally and its responsibilities are mostly request-driven. The device integration service is constrained by the need to maintain persistent communication state with connected devices. This asymmetry is acceptable for the prototype, but it is an important architectural boundary that would need further work in a larger-scale production environment.

## 3.7 Chapter summary

The architecture of the system developed in this thesis is shaped by the need to keep digital entitlement, operational administration, and physical access control aligned without collapsing all concerns into one runtime component. The resulting design separates the

domain platform from the device integration boundary, uses a capability-based identity model, grounds authorization in persisted relationships, and relies on cloud deployment for realistic validation. The next chapter explains how this architecture was realized in the implementation.

# 4 Implementation

## 4.1 Development approach

Implementation proceeded iteratively rather than in one monolithic pass. The earliest working milestone was deliberately narrow: a connected parking lock device could maintain a live session with the device integration service, the backend could issue a control request, and a client action could trigger that request. This first slice was intentionally chosen because every later workflow depends on the credibility of the control path. If physical actuation cannot be reached reliably, the surrounding account, inventory, or rent logic remains academic but not operational.

Once the basic control path existed, development expanded outward in layers. Persistence was introduced to make device registration, inventory, and account lifecycle durable. Capability modeling was then added so that one account could participate in several workflows without being duplicated across identities. Ownership assignment, rent validation, and operator tooling followed. This order reduced risk because each phase reused and stress-tested the previous one instead of forcing several unstable abstractions to evolve simultaneously.

The implementation process also benefited from keeping the system deployable throughout development. Even while local work continued, the target architecture remained recognizable: a backend API, a device integration service, a PostgreSQL data store, and a web client. This consistency made it possible to identify issues that only appear when several layers interact, such as secrets propagation, email transport failures, stale device state, and mixed-capability authorization.

## 4.2 Backend API and service structure

The backend API was implemented in C++ using Drogon. At the HTTP boundary it exposes account, inventory, rent, administrative, and device-control endpoints. Internally, however, its more important role is to act as the policy enforcement point of the system. The backend does not merely relay requests to storage or to hardware-facing services. It evaluates whether the current account is entitled to perform the requested operation, whether the request is coherent with the persisted domain state, and whether downstream control should proceed at all.

This responsibility shaped the internal organization of the service layer. Domain-facing operations are expressed through application services rather than by placing non-trivial logic directly inside controller methods. Controllers remain responsible for request parsing, response generation, and authentication context extraction. Application services evaluate capability rules, ownership relations, rent validity, and operator privileges. Persistence adapters then translate those higher-level decisions into PostgreSQL operations. This separation reduces coupling between HTTP mechanics and business rules, which is particularly useful in a system where the same entity relationships are reused across several workflows.

The account subsystem illustrates this separation clearly. Registration, login, refresh, verification, reset, and logout are presented to the client as separate endpoints, but they share the same domain view of an account. A registered account can later be promoted to owner capability without becoming a different identity. That behavior would be awkward in a purely role-switched design, but it is natural when capability records and account state are first-class domain entities. The backend therefore treats identity as persistent and multi-context rather than ephemeral and single-purpose.

The same pattern appears in inventory management. Parking groups, spaces, device assignments, and pricing plans are stored as related domain objects rather than independent configuration fragments. This gives the backend enough context to answer questions such as which account owns a given space, which device is currently assigned to it, and whether the current rent period is active. The rent path is particularly sensitive because it is the bridge between customer-facing booking behavior and physical access control. As a result, rent validation lives in the backend policy layer rather than in the web client or the device integration service.

### 4.2.1 Session refresh and authorization context

```
load refresh session by token
if session missing or revoked:
    reject refresh request

load account and capabilities
if account is disabled or not verified where required:
    reject refresh request

rotate refresh token
issue new access token with account_id context
return access token, refresh token, and resolved capabilities
```

This flow matters because the authorization model depends on more than static claims encoded at login time. The system needs current account state, current capability set, and current session validity in order to make reliable decisions across customer, owner, and operator workflows.

## 4.3 Device integration service and control path

The device integration service forms the boundary between domain logic and field hardware. Its purpose is not to decide who may perform an operation; that remains the backend's responsibility. Instead, the service maintains persistent device sessions, resolves the active connection associated with a device identifier, dispatches control requests to that live session, and reports resulting state back into the platform. This division of labor is what allows the thesis to keep hardware-specific concerns out of the general application runtime without pretending that they do not exist.

From an implementation perspective, the service has to solve three related problems. First, it must track live device connectivity and distinguish recently active devices from stale ones. Second, it must accept internal control requests from the backend and map them to the correct connected parking lock device. Third, it must turn device-originated

signals into persisted operational state and event history. None of those responsibilities fit naturally inside the browser client, and placing them directly inside the backend API would have entangled account and inventory policy with long-lived communication state.

An important consequence of this design is that the device integration service is stateful in a way that the backend API is not. The backend can be replicated more freely because its source of truth lives in PostgreSQL and its requests are mostly short-lived. The device integration service, in contrast, depends on currently active device sessions. That asymmetry is not a defect in itself; it reflects the real requirements of controlling connected hardware. What matters is that the thesis architecture recognizes the asymmetry and gives it a dedicated boundary.

This service also became the place where operational correctness could be improved through observation. When testing showed that a powered-off device might remain incorrectly marked online if no explicit disconnect event arrived, the corrective change did not belong in the browser and did not belong in rent logic. It belonged in the device-state handling path, where freshness, telemetry, and commandability could be derived from the most recent device evidence.

### 4.3.1 Device lookup and command dispatch

```
receive internal control request for space_id
resolve currently assigned device_id for the space
resolve active session for that device_id
if no active session exists:
    reject request as non-commandable

submit command through the device integration service
wait for resulting status or timeout
persist resulting device event and derived state
return command outcome to backend
```

The decision to make the lookup space-centric rather than device-centric is intentional. The user interacts with a rented or owned space, not with an IMEI. The device identifier is therefore a consequence of the current space assignment and not the primary domain object exposed to the client.

## 4.4 Frontend web application

The client was implemented in Flutter and executed in web-server mode during testing. Although the prototype runs as a web application, its structure already reflects the fact that different actor types must be supported within one interface. The client therefore does not behave as a single-purpose consumer application. It conditionally exposes navigation and actions according to the capabilities returned by the backend, while still treating the account as one continuous identity.

This design became particularly useful after mixed-capability behavior was introduced. A user can be both customer and owner at the same time. In practice that means the same account may browse public spaces and create a rent period while also managing owned inventory. The client responds to this by presenting different sections instead of forcing

a hard mode switch or requiring separate logins. That decision aligns with the capability-based account model described in Chapter 3 and demonstrates how the architectural choice influences user experience.

The frontend also became the main surface for validating the clarity of user actions. Several refinements emerged not from abstract design discussion but from observed confusion during integrated testing. The rent flow was collapsed into a single action with interval-based interpretation so that the customer does not need to decide between inconsistent short-term and long-term forms. Grouped list expansion was made persistent so that lock and unlock actions do not collapse the current browsing context unnecessarily. Expired rent periods were filtered more strictly so that stale operational cues do not remain visible after a reservation record has lost relevance for control. These improvements are modest in isolation, but together they show that the client evolved through direct interaction with the system rather than only through static specification.

The client also plays a useful explanatory role in the architecture. It demonstrates that the browser should never decide authorization by itself. The web application can hide or reveal actions for usability reasons, but the authoritative decision still belongs to the backend. This distinction became especially important in lock and unlock workflows: the interface may avoid presenting a customer action that is obviously invalid, but the backend must still re-evaluate ownership or active rent state for every request.

## 4.5 Authentication and email verification

The current authentication flow is backend-owned and email-based. Accounts are registered by email, verified through link-based confirmation, and authenticated through password-based login. Access tokens are intentionally short-lived, while refresh sessions remain persisted so that the platform can support logout, session replacement, and future hardening work. Verification and password-reset flows share the same email-delivery path, which made the cloud email integration an essential part of validation rather than a cosmetic add-on.

This subsystem has two important implementation characteristics. First, it treats account lifecycle as part of the application domain rather than as an external identity-provider concern. Second, it keeps the account model flexible enough for later owner promotion without identity fragmentation. A newly registered customer therefore remains the same account even after being granted additional capability and later assigned ownership of a parking group. That continuity is important both for usability and for auditability.

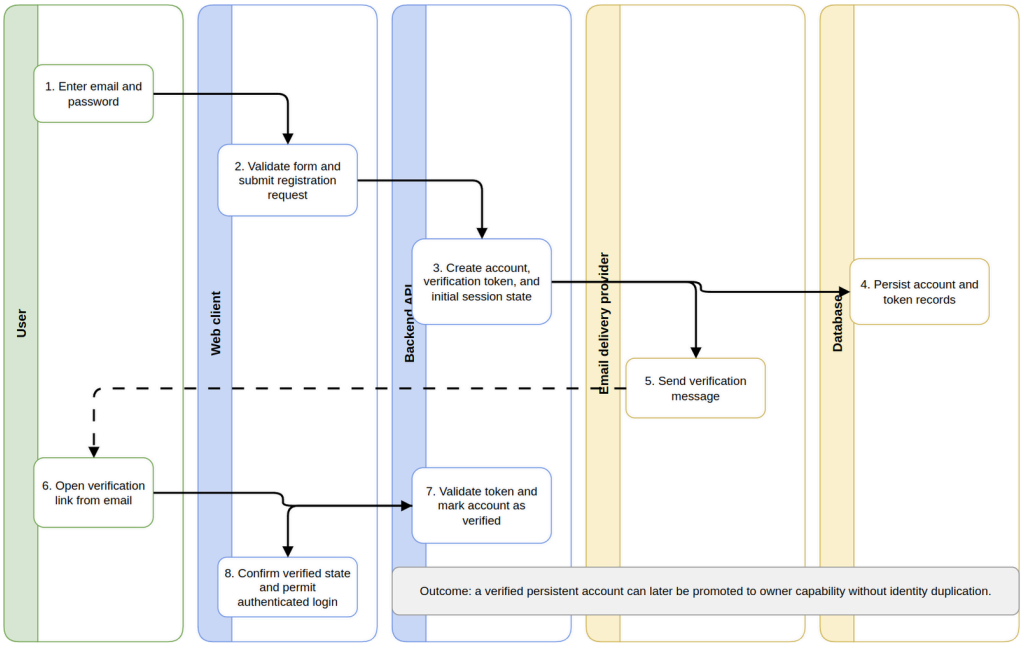


Figure 7: Account registration and email verification flow.

## 4.6 Operator and administrator workflow support

A dedicated operator CLI was introduced to make environment setup repeatable and to separate deterministic administrative tasks from user-facing browser flows. This distinction was practical rather than ideological. During validation, the important question was not whether every setup action could be placed in the web UI, but whether the platform could be configured and reconfigured reliably while the user-facing flows remained realistic and uncluttered.

The CLI supports bootstrap login, group and space creation, device registration, device assignment, pricing updates, owner promotion, group-owner assignment, and direct operational actions such as lock and unlock. These commands encode the minimum repeatable sequence required to bring a clean environment into a state where a customer can create a rent period and a real connected parking lock device can be controlled. That made the CLI not only a convenience tool but also an implementation artifact that improved the scientific quality of the validation process by making repeated scenario execution feasible.

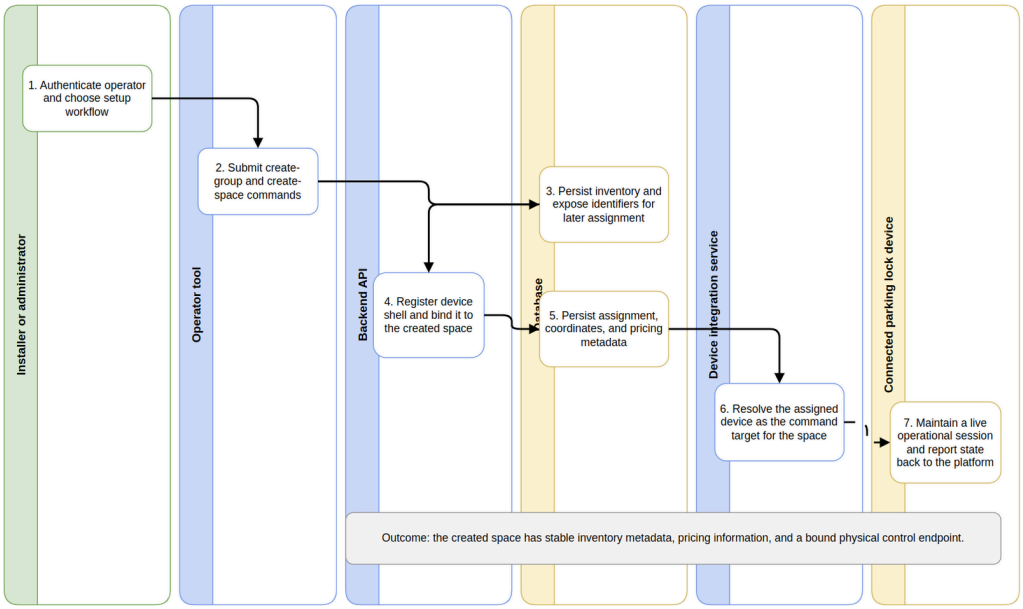


Figure 8: Device registration and assignment workflow.

## 4.7 Infrastructure as code and deployment

Infrastructure provisioning and deployment were implemented through Terraform and Kubernetes manifests, with each tool serving a distinct purpose. Terraform defines and provisions the cloud resources that exist outside the cluster, such as networking primitives, the managed relational database, and container registry support. Kubernetes manifests define the runtime composition of the deployed application, including service workloads, secrets integration, exposure rules, and namespace-scoped organization.

This split is beneficial for two reasons. First, it preserves a clear distinction between cloud estate creation and application deployment. Second, it allowed the platform to move from local development to a managed cloud environment without changing the logical architecture of the system. The backend API did not need to be redesigned for cloud hosting; instead, the deployment artifacts expressed where the already-defined services would run and how they would receive configuration. That is a more defensible engineering result than one in which cloud deployment requires application-level special cases.

The deployment approach also contributed directly to validation realism. Because email delivery, secrets propagation, service restart behavior, and database persistence were all exercised in the target runtime, several integration defects became visible that would not have appeared in a purely local demonstration. In other words, infrastructure as code was not only an operational convenience but also part of the thesis methodology.

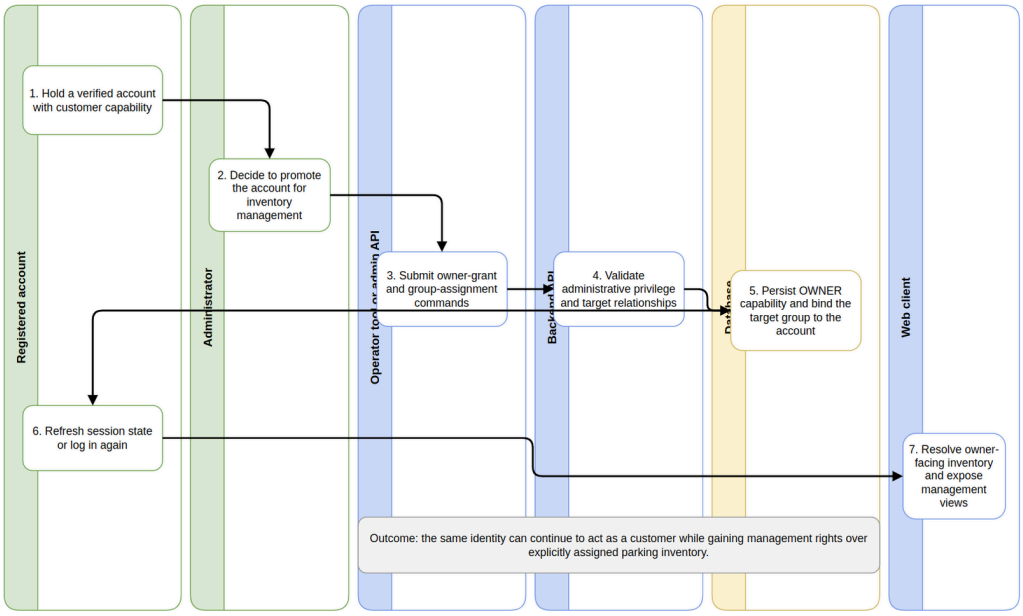


Figure 9: Promotion of a registered account to owner capability and assignment of owned inventory.

## 4.8 Persistence, migration, and state derivation

The persistence layer had to serve two different but connected purposes. On one side, it stores ordinary application data such as accounts, capability assignments, parking groups, spaces, and pricing plans. On the other side, it stores operational evidence, especially refresh sessions, email-verification or reset tokens, device events, and derived device state. Treating these as one coherent persistence model was important because the system frequently resolves authorization and operational status by combining business relations with recent technical evidence.

Schema management was intentionally conservative. Rather than depending on heavy runtime migration automation, the prototype uses explicit SQL migration assets committed alongside the backend source tree. This keeps schema evolution inspectable and suitable for an academic setting in which the underlying data model must be understandable to the reader. The trade-off is that migration authorship remains manual, but that trade-off is acceptable for a thesis-scale system because clarity of structural change is more valuable than abstract automation.

State derivation became especially important for device behavior. The platform does not merely store raw events and display them verbatim. Instead, the persistence path derives higher-level operational facts, such as whether a device should currently be regarded as online, whether it is commandable, and which lock state is most recently known. This is what turns a historical event stream into a state model that can support user-facing decisions and backend authorization checks. Without that derivation layer, the interface would expose raw telemetry fragments rather than a coherent operational picture.

## 4.9 Configuration and secret propagation

The deployment runtime depends on several configuration classes with different lifecycles. Static application settings, such as service ports or namespace-level routing assumptions, change relatively rarely. Sensitive values, such as database credentials, JWT signing keys, internal service tokens, and email-provider credentials, require stricter handling. The implementation therefore distinguishes between plain configuration and secret-bound configuration, even though both ultimately enter the running container environment through Kubernetes-managed mechanisms.

This distinction became operationally meaningful during validation. Several defects were not caused by incorrect source code alone, but by the relationship between the code and its runtime configuration. Examples include missing environment-variable wiring, stale secret values that were not yet reflected in running pods, and email delivery failures caused by incomplete provider configuration. These experiences justify documenting configuration and secret propagation as part of the implementation chapter rather than treating them as external deployment trivia.

A related design decision was to keep bootstrap administrative access in configuration rather than in hardcoded seed data. That choice supports clean test resets and makes it possible to restore a minimal operator entry point without repopulating broader business data. It also aligns with the thesis objective of validating the system from an intentionally sparse starting state. In a larger production setting, this approach would require stronger operational controls, but for the validated prototype it provides a transparent and repeatable initialization path.

## 4.10 Operational UI refinements informed by testing

Although the thesis is not centered on frontend aesthetics, the client implementation still required several operational refinements before the full scenario felt trustworthy. A recurrent pattern during testing was that the underlying backend state would be correct while the interface still communicated something misleading or awkward. For example, group expansion state could collapse after a lock operation, expired rent information could remain visible longer than appropriate, and the early rent interaction split made it too easy to produce confusing duration behavior.

These refinements matter academically because they show that system correctness includes representational correctness. A platform that enforces the right authorization rule but consistently presents stale or misleading context still undermines user trust and operator comprehension. The improved interface therefore does more than make the prototype pleasant to use. It demonstrates that the platform can present backend truth in a way that supports the real workflows used in validation.

## 4.11 Selected pseudocode excerpts

The following excerpts summarize important logic paths without reproducing large source files. They are intended to clarify behavior rather than to mirror implementation line by line.

### 4.11.1 Rent duration classification

```
if end_at <= start_at:
    reject request

duration = end_at - start_at

if duration < 1 hour:
    reject request
else if duration < 1 day:
    classify as short-term rent
else if duration <= 31 days:
    classify as long-term rent
else:
    reject request
```

This logic was introduced after earlier testing exposed inconsistent handling of long-term inputs. Its main purpose is to keep rent classification a backend concern instead of trusting the client to classify duration correctly.

### 4.11.2 Lock and unlock authorization

```
load space by space_id
if space does not exist:
    reject request

if account has ADMIN or INSTALLER or SERVICE:
    allow command
else if account owns the group containing the space:
    allow command
else if account has an active current rent on the space:
    allow command
else:
    reject request
```

This logic is intentionally space-centric. It avoids granting access only on the basis of capability names and instead resolves the current account-to-space relationship before any physical control request is forwarded.

### 4.11.3 Device state refresh from events

```
receive device event
persist raw event payload
update last_seen_at
if event carries lock state or telemetry:
    derive device state fields
if last_seen_at is older than freshness threshold:
    mark device offline and non-commandable
```

The freshness check was added after observing that a powered-off device could remain incorrectly marked as online if no explicit disconnect event was received.

### 4.11.4 Owner assignment and inventory enablement

```
load target account by account_id
grant OWNER capability if missing

load target parking group by group_id
set group.owner_account_id = account_id

after next account refresh or login:
    expose owner-facing inventory management paths
```

This sequence emphasizes that owner capability and owned inventory are related but distinct concepts. One grants the type of authority an account may hold; the other binds that authority to a concrete set of managed spaces.

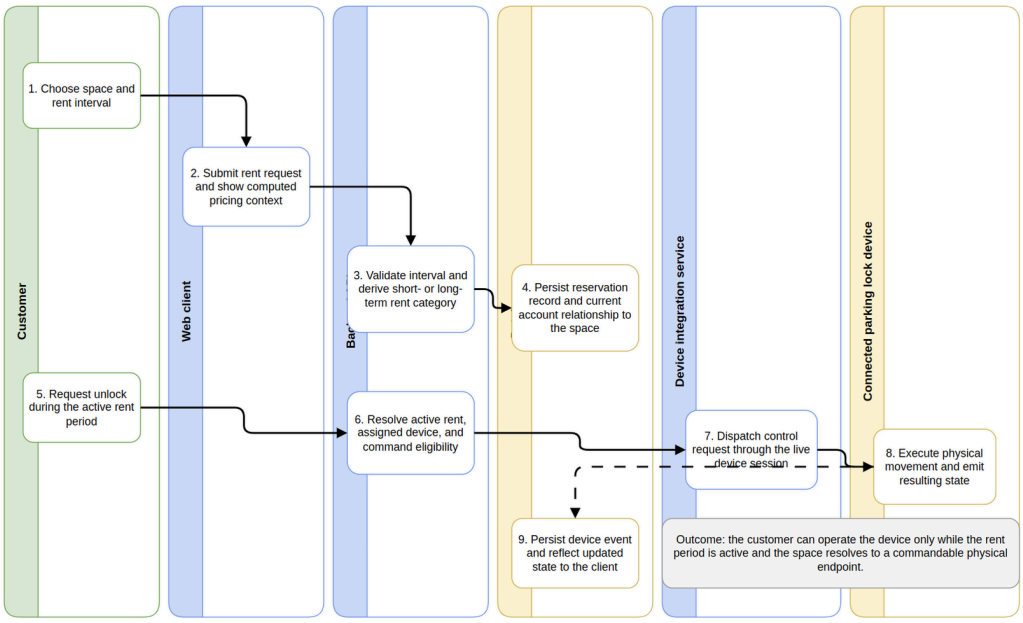


Figure 10: Rent creation and subsequent authorized lock operation flow.

## 4.12 Chapter summary

The implementation translates the design into an operational system by keeping concerns clearly separated: the backend owns identity and policy, the device integration service owns hardware-facing session and command execution, the frontend owns user interaction, and infrastructure code owns reproducible deployment. The resulting system is not only functionally complete enough for end-to-end validation, but also structured in a way that makes its key engineering decisions visible. The next chapter evaluates how well this implementation satisfies the requirements in practice.

# 5 Validation

## 5.1 Validation environment

Validation was performed against the deployed cloud environment rather than only on a local machine. This decision matters because the system developed in this thesis spans several boundaries that are easy to idealize in a local-only demonstration: persistent account onboarding, secrets propagation, container restarts, database durability, and live communication with a connected parking lock device. An environment that omits those boundaries risks overstating correctness.

The validated runtime therefore consisted of five relevant parts. First, the user-facing web client was executed through Flutter's web-server mode and used for owner and customer workflows. Second, the backend API was deployed to the Kubernetes runtime and exposed the public HTTP surface for account, inventory, and rent operations. Third, the device integration service was deployed alongside it and maintained the stateful control path to the connected parking lock device. Fourth, PostgreSQL was hosted through Amazon RDS and used as the persistent source of truth for accounts, inventory, sessions, pricing, and device events. Fifth, account verification and password-reset messages were delivered through Amazon SES so that email-based onboarding was exercised under realistic transport conditions.

The presence of a real connected parking lock device is especially important. Without it, the thesis could still demonstrate reservation logic and interface behavior, but it could not validate the core claim that a digitally managed rent period can govern access to a physical parking space. The device therefore serves not merely as a demonstration artifact, but as a validation dependency.

## 5.2 Validation methodology

The chosen methodology was scenario-driven end-to-end validation. Instead of evaluating features in isolation, the system was tested through full operational sequences that resemble realistic setup and use. This approach was selected because the architectural value of the platform lies in correct coordination across several layers: identity, inventory, ownership, rent state, device assignment, and physical control. Verifying those layers separately would have been useful for unit correctness but insufficient for the thesis-level claim.

The validated scenario sequence contained the following major stages:

- bootstrap administrator access in a clean environment;
- creation of a parking group and a space;
- registration and assignment of a real device;
- coordinate and pricing configuration;
- owner registration and promotion;
- assignment of the created group to the owner;
- customer registration and email verification;

- creation of a rent interval;
- lock and unlock operations by the active renter.

This methodology has two advantages. First, it demonstrates whether the platform can actually be operated from an empty state to a working state. Second, it forces the system to reveal cross-layer defects that remain invisible when only one service is inspected at a time. The methodology is therefore aligned with the thesis objective: to validate an integrated parking access platform rather than an isolated algorithm.

## 5.3 End-to-end scenario execution

### 5.3.1 Administrative bootstrap and inventory setup

**Objective.** Confirm that a clean environment can be brought into a usable administrative state without relying on manually seeded business data.

**Actions performed.** The database was reset, the backend was restarted with bootstrap administrator credentials supplied through configuration, and the operator CLI authenticated with that administrator account. A parking group and a space were then created through the repeatable operator workflow.

**Expected result.** The environment should contain one administrator account, no residual operational data, and a newly created group-space pair ready for device assignment.

**Observed result.** The bootstrap account was restored correctly and the operator workflow could create the required inventory. This stage was important because later owner and customer flows depend on the environment being clean and deterministic. It also verified that the platform no longer relied on opaque demo seeding behavior during end-to-end testing.

### 5.3.2 Device registration and assignment

**Objective.** Confirm that a connected parking lock device can be represented in the platform, attached to the previously created space, and surfaced through the operator workflow.

**Actions performed.** The operator CLI registered a device shell by identifier, assigned it to the created space, and attached coordinates and pricing information to that space. Device connectivity and commandability were then inspected through the user-facing interface and service logs.

**Expected result.** The space should expose both inventory metadata and device-backed operational state. The assigned device should appear as the physical endpoint associated with that space rather than as a disconnected object.

**Observed result.** The operator path succeeded, and the space became visible with attached pricing and location context. This stage also established the data relationship later required for customer authorization: a rent period is created against a space, and the space in turn resolves to the currently assigned device.

### 5.3.3 Owner onboarding and ownership assignment

**Objective.** Confirm that a normal registered account can be elevated to owner capability and bound to specific inventory without becoming a separate identity.

**Actions performed.** A new account was registered through the browser workflow, verified by email, and inspected as an ordinary customer account. An administrative action then granted owner capability, and a second administrative action assigned the previously created parking group to that account.

**Expected result.** The same account should remain valid for customer behavior but also gain the ability to manage the assigned group and its spaces after capability refresh.

**Observed result.** The promotion and group-owner assignment path behaved as expected. This stage validated an important design distinction: capability alone does not imply ownership of all inventory, and ownership assignment must remain explicit.

### 5.3.4 Customer onboarding, rent creation, and device control

**Objective.** Confirm that a separate customer can complete the full path from registration to an authorized physical control action.

**Actions performed.** A second user account was registered, verified through email, and signed into the web application. The customer created a rent period for the configured space, after which unlock and lock operations were attempted during the active interval.

**Expected result.** The rent should be persisted and classified correctly, the current user should be permitted to operate the device only while the rent period is active, and the resulting control outcome should be reflected back into the operational state.

**Observed result.** After corrections described later in this chapter, the full path succeeded. This was the most important acceptance scenario in the entire validation because it linked digital entitlement, temporal validation, space assignment, and physical control of the connected parking lock device.

## 5.4 Validation matrix

| Requirement | Validation method | Result |
|---|---|---|
| FR1: persistent account onboarding | Register, verify, login, and reset through SES-backed email flow | Satisfied |
| FR3 and FR4: inventory and device setup | CLI-driven group, space, and device provisioning | Satisfied |
| FR5: owner management path | Promotion plus group-owner assignment | Satisfied |
| FR6: rent creation | Web-based rent flow with interval validation | Satisfied |
| FR7: authorized lock and unlock | Active renter and staff-only control tests | Satisfied after correction |
| NFR2: reproducible deployment | Terraform and Kubernetes deployment and restart cycles | Satisfied |

Table 5: Condensed validation matrix for the most important requirements.

## 5.5 Requirement coverage discussion

The validation matrix shows that the platform satisfies the most important requirements directly, but the meaning of those satisfactions differs slightly across categories. The account and onboarding requirements were satisfied through repeated registration, verification, login, logout, and reset flows executed against the live email-delivery path. Inventory and device provisioning requirements were satisfied through deterministic operator setup rather than by exposing every setup action directly in the browser. This is acceptable because the requirement concerns the platform capability, not the exclusive location of the operator interface.

Owner management and rent creation were both satisfied under more constrained conditions. Owner management depends on two separate operations: granting the relevant capability and assigning actual inventory ownership. Rent creation depends not only on storing a reservation record but also on validating interval boundaries and interpreting that record in later authorization decisions. These requirements therefore count as satisfied because the validated workflows exercised the entire required chain rather than just the first persistence step.

The most operationally significant requirement was authorized lock and unlock control. It is the clearest example of why scenario-driven validation was necessary. This requirement was not truly satisfied until the system correctly handled active-rent checks, mixed-capability accounts, stale device status, and real command dispatch through the device integration service. The final validated behavior satisfies the requirement, but only after several engineering corrections were made.

## 5.6 Failures encountered and engineering corrections

The validation process exposed several meaningful defects. They are important not as embarrassments, but as evidence that the system was exercised under realistic conditions rather than demonstrated only in idealized paths.

### 5.6.1 Stale online state after abrupt device loss

**Symptom.** A powered-off device could remain visible as online, which was misleading because customers or operators could believe a control path still existed.

**Likely cause.** The original state derivation relied too heavily on explicit disconnect events. That assumption is fragile for field hardware because abrupt power loss does not guarantee a graceful disconnect signal.

**Corrective change.** A freshness threshold based on **`last_seen_at`** was introduced. After that threshold expires, the device is treated as offline and non-commandable even if no explicit disconnect event has been recorded.

**Engineering lesson.** Device truth must be inferred from recent evidence, not only from explicit teardown signals.

### 5.6.2 SES delivery failure in deployed runtime

**Symptom.** Registration and password-reset flows could create account state but fail to deliver the required email in the cloud environment.

**Likely cause.** The initial sender implementation contained transport and request-signing mistakes that were not visible in purely local or log-only testing.

**Corrective change.** The HTTPS request path was corrected and AWS SigV4 signing was delegated to a stable `curl`-based execution path in the sender implementation.

**Engineering lesson.** Email delivery should be validated against the real managed provider because transport-level correctness cannot be assumed from stubbed flows.

#### 5.6.3 Mixed-capability authorization error

**Symptom.** An account that was both customer and owner could be evaluated only along the owner branch when attempting to control a space it rented but did not own.

**Likely cause.** The authorization path privileged one capability interpretation too early instead of resolving all valid relationships to the requested space.

**Corrective change.** The lock and unlock logic was rewritten so that authorization is determined against the requested space and any valid relationship to it, including active current rent.

**Engineering lesson.** In multi-capability systems, precedence rules must be designed around domain context rather than around capability labels alone.

#### 5.6.4 Misleading frontend operational state

**Symptom.** The web client could show expired rents or visually stale device status that no longer reflected actual operational conditions.

**Likely cause.** Frontend filtering rules and device-state presentation lagged behind the refined backend model.

**Corrective change.** Time-based filtering was tightened, rent display became more explicit, and state derivation was aligned with the backend freshness logic.

**Engineering lesson.** User-facing correctness is not only about authorization; it also requires that the visible operational narrative stays synchronized with backend truth.

### 5.7 Operational observations from integrated testing

Several observations emerged only because the system was tested as a cloud-hosted, hardware-integrated whole.

First, user-facing defects and infrastructure defects often presented themselves through the same symptom: a customer action failed. Without integrated testing, it would have been easy to misclassify such failures as purely frontend, purely backend, or purely hardware problems. In practice, the debugging path often crossed several layers before the true cause became visible.

Second, deployment realism materially improved architectural understanding. Secrets injection, managed email delivery, runtime restarts, and externally hosted persistence all influenced the confidence that could be placed in the platform. This demonstrates that cloud deployment in the thesis is not merely cosmetic. It is part of the empirical basis for claiming that the architecture behaves coherently outside a local-only laboratory setup.

Third, the operator workflow proved to be an important validation asset. By separating repeatable administrative setup from browser-centered end-user behavior, the testing process could focus on the platform's most meaningful interactions without turning every environment reset into a manual exercise. This improved repeatability and made defect reproduction considerably easier.

## 5.8 Threats to validity

The validation results are informative, but they should not be overgeneralized. The most obvious threat to validity is scale. The validated prototype was exercised against a real cloud deployment and a real connected parking lock device, but not under large concurrent load or wide device fleets. This means that the thesis can credibly claim integrated correctness for the tested workflow, but it cannot yet claim production-grade throughput characteristics or large-scale operational resilience.

Another threat concerns environmental specificity. The runtime was intentionally realistic, yet it still reflects one particular stack choice: a web client driven from a browser, a Kubernetes-hosted backend, AWS-managed support services, and one device integration path. The architectural conclusions remain useful beyond that exact environment because they concern separation of domain logic from device integration and the relationship between authorization and persisted context. Nevertheless, quantitative conclusions about performance, cost, or operational burden would need broader comparative study.

A third threat concerns the human role in scenario execution. End-to-end validation necessarily included operator actions such as promoting an owner, assigning a group, and resetting the environment between runs. The presence of the operator CLI reduced ambiguity and improved repeatability, but it did not eliminate the human factor entirely. This is acceptable for an engineering thesis whose objective is to validate an integrated prototype, provided that the workflow remains explicit and repeatable, which it does.

## 5.9 Current limitations

The current implementation remains a prototype and has clear limitations:

- production-grade HTTPS policy and broader security hardening are not yet finalized;
- the web geolocation path is constrained by browser secure-context rules and therefore is not uniformly available on every access path;
- the device integration service currently favors correctness of live device sessions over horizontal replication;
- formal load testing and resilience testing under high concurrency were not part of the present scope;
- payment processor integration and accounting-grade billing remain outside the implemented feature set.

Some of these limitations are strategic rather than accidental. For example, the thesis intentionally stops short of implementing formal payment processing because that would introduce a new compliance and trust domain that is orthogonal to the central architectural question of digitally governed physical access. Similarly, the device integration service was optimized for correctness of persistent communication and state handling

rather than for multi-node horizontal balancing. The absence of those features is therefore best understood as a scope boundary, not as an unnoticed omission.

These limitations do not negate the validated result, but they define the boundary between a credible integrated prototype and a fully hardened production system. Their explicit acknowledgement is therefore part of the thesis argument rather than an afterthought.

## 5.10 Validation summary

The validation results show that the implemented platform satisfies its core functional goals. The system can provision inventory, onboard users, assign ownership, register and attach a real device, create a valid rent period, and permit lock and unlock actions only for authorized participants. Just as importantly, the validation process uncovered and resolved defects that are characteristic of real integrated systems rather than toy examples. This strengthens the claim that the prototype is a credible software engineering result and that its architectural decisions were tested under conditions close enough to reality to be informative.

# 6 Conclusion

## 6.1 Project summary

This thesis presented the design and implementation of a cloud platform for parking space rent and access control using connected electromechanical parking lock devices. The resulting system connects role-based user management, inventory administration, rent handling, cloud deployment, and physical device control within a single operational prototype. The work demonstrates that the problem can be treated as an integrated software engineering challenge rather than as a collection of unrelated user-interface and hardware tasks.

## 6.2 Alignment with objectives

The central objective was to build a working platform that could register users, provision parking inventory, bind spaces to devices, assign ownership, create rents, and enforce physical access through role-aware and rent-aware authorization. The validated prototype meets this objective. The platform supports bootstrap administration, installer workflows, owner enablement, customer onboarding, email verification, and lock and unlock operations against a live connected device.

## 6.3 Engineering lessons learned

Several lessons emerged from the project. First, physical-device systems require explicit treatment of stale state, because disconnection is not always signaled cleanly. Second, capability-based identity models are more practical than single-role models in multi-actor domains where one person may participate in different capacities. Third, cloud validation should be started early enough to expose infrastructure-specific defects such as email delivery issues and secret wiring problems. Finally, operational tooling such as the administrator CLI is not merely a convenience; it is an essential part of making a complex workflow testable.

## 6.4 Limitations

The current system remains a prototype. Security hardening is not complete, the hardware-integration boundary still requires more mature high-availability strategy, and payment integration is not yet implemented. The frontend is validated primarily in its web form, and some usability improvements remain possible. These limitations do not invalidate the current result, but they define the boundary between the implemented prototype and a production-ready commercial system.

## 6.5 Future work

Future work should focus on four areas. The first is production hardening: HTTPS enforcement, stronger secret management, MFA, and improved audit support. The second is commercial completion: payment integration, settlement, and clearer financial reporting. The third is operational maturity: richer observability, resilience testing, and improved

gateway failover strategy. The fourth is product usability: deeper owner tooling, more refined installer workflows, and an expanded native mobile path if required.

Overall, the project achieved its academic goal by producing a technically coherent system and validating it through a realistic end-to-end scenario. The prototype is therefore not only an implementation artifact, but also a useful foundation for further engineering and research.

# A Appendix

## AA Operator CLI reference

The project includes a dedicated operator tool at **`scripts/operator_cli.py`**. The following commands were used during the validated end-to-end scenario:

- **`login`**
- **`me`**
- **`create-group`**
- **`create-space`**
- **`register-device`**
- **`assign-device`**
- **`set-coordinates`**
- **`set-pricing`**
- **`grant-owner`**
- **`assign-group-owner`**
- **`device-events`**
- **`rent`**
- **`unlock`**
- **`lock`**

## AB Representative setup commands

```
python3 scripts/operator_cli.py login \
  --base-url http://<backend-host> \
  --email <bootstrap-admin-email> \
  --password <bootstrap-admin-password>

python3 scripts/operator_cli.py create-group \
  --group-id group-rsa \
  --name "Residential Gate A" \
  --code-prefix RSA \
  --visibility PUBLIC

python3 scripts/operator_cli.py create-space \
  --group-id group-rsa \
  --space-id space-rsa-a1 \
  --space-code A1 \
  --name "RSA1" \
  --rent-mode BOTH
```

## AC Selected API payload examples

A minimal example of a registration payload:

```
{
  "email": "customer@example.com",
```

```
  "password": "StrongPass123"
}
```

A minimal example of a device assignment payload:

```
{
  "groupId": "group-rsa",
  "spaceId": "space-rsa-a1"
}
```

# AD Figure plan and source inventory

| Figure | Placement | Working caption |
|---|---|---|
| Figure 2.1 | Chapter 2, after domain context | Stakeholders and control relationships in the parking access domain. |
| Figure 3.1 | Chapter 3, architectural overview | System context of the parking access platform. |
| Figure 3.2 | Chapter 3, architectural overview | Container-level architecture of the parking access platform. |
| Figure 3.3 | Chapter 3, design decisions | Responsibility split between domain services and device integration services. |
| Figure 3.4 | Chapter 3, data model and authorization context | Core persistence model for identity, inventory, device, and rent management. |
| Figure 3.5 | Chapter 3, technology stack and deployment | Cloud deployment topology used for the validated prototype. |
| Figure 4.1 | Chapter 4, authentication flow | Account registration and email verification flow. |
| Figure 4.2 | Chapter 4, provisioning workflow | Device registration and assignment workflow. |
| Figure 4.3 | Chapter 4, rent-control workflow | Rent creation and subsequent authorized lock operation flow. |
| Figure 4.4 | Chapter 4, ownership workflow | Promotion of a registered account to owner capability and assignment of owned inventory. |

Table 6: Planned thesis figures, their placement, and their working captions.

| Figure | Purpose | Source asset |
|---|---|---|
| Figure 2.1 | Establish the multi-actor structure of the problem domain. | `resources/img/fig-2-1-domain-context.png` |
| Figure 3.1 | Define the platform boundary and external actors. | `resources/img/fig-3-1-system-context.png` |
| Figure 3.2 | Show deployable units and their communication. | `resources/img/fig-3-2-container-architecture.png` |
| Figure 3.3 | Explain why platform logic and device integration are separated. | `resources/img/fig-3-3-backend-gateway-components.png` |
| Figure 3.4 | Support explanation of contextual authorization and ownership. | `resources/img/fig-3-4-er-model.png` |
| Figure 3.5 | Document the validated AWS runtime environment. | `resources/img/fig-3-5-aws-deployment.png` |
| Figure 4.1 | Explain account onboarding at a high level. | `resources/img/fig-4-1-registration-sequence.png` |
| Figure 4.2 | Explain inventory and hardware provisioning. | `resources/img/fig-4-2-device-onboarding-sequence.png` |
| Figure 4.3 | Show the bridge between rent state and physical control. | `resources/img/fig-4-3-rent-lock-sequence.png` |
| Figure 4.4 | Explain owner enablement workflow. | `resources/img/fig-4-4-owner-assignment-sequence.png` |

Table 7: Planned thesis figures, their purpose, and the rendered assets used in the draft.

## AE Workflow figure guidance

The workflow diagrams planned for Chapter 4 should remain abstract and avoid hardware-specific communication details. In particular, they should use labels such as **`device integration service`**, **`connected lock device`**, **`validate active rent`**, and **`submit control request`** instead of low-level transport or message terminology. This keeps the diagrams academically useful while avoiding unnecessary disclosure of internal integration mechanics.

## AF Notes on disclosure

This appendix intentionally includes only moderate operational detail. It is sufficient to explain the engineering work and the validation scenario, but it does not attempt to document every internal mechanism or every environment-specific secret-management detail.