T.R.

GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

SELF-DRIVING QUADRICYCLE RENTAL SYSTEM WITH MOBILE APPLICATION AND MAP SUPPORT

EMRE GÜLEÇ

SUPERVISOR
DOÇ. DR. MEHMET GÖKTÜRK

GEBZE
2026

T.R.
GEBZE TECHNICAL UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

SELF-DRIVING QUADRICYCLE RENTAL SYSTEM WITH MOBILE APPLICATION AND MAP SUPPORT

EMRE GÜLEÇ

SUPERVISOR
DOÇ. DR. MEHMET GÖKTÜRK

2026
GEBZE

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 11/01/2026 by the following jury.

JURY

Member
(Supervisor) : Doç. Dr. Mehmet Göktürk

Member : Dr. Salih Sarp

ABSTRACT

This thesis describes the design and implementation of a self-driving quadricycle rental platform that provides a contactless user experience through a mobile app, cloud backend, and connected vehicle electronics. The project addresses the growing need for safe micro-mobility solutions in urban areas where users need to find, unlock, and return shared vehicles without staff assistance. The system uses an Expo React Native client, Supabase for authentication and data management, and embedded modules on the quadricycle for telemetry and remote control.

The work covers requirement analysis, system architecture, and a complete rental flow that handles reservations, QR code scanning for ride start, live trip tracking, and automatic payment processing. The implementation uses reusable hooks for managing rental states, secure session handling with local storage, and a spatial database for tracking vehicle locations on the map. Testing showed that the system works as intended and provides a practical foundation for deploying autonomous quadricycle services in the future.

Keywords: autonomous mobility, quadricycle rental, React Native, Supabase, IoT telemetry.

ÖZET

Bu tez, kullanıcıların otonom bir quadricycle filosunu temassız biçimde kiralamasını sağlayan mobil uygulama, bulut tabanlı arka uç ve araç elektroniklerinden oluşan tümleşik bir sistemi sunmaktadır. Kent içi mikro mobilite alanında yaşanan erişilebilirlik, güvenlik ve operasyon maliyeti sorunları; rezervasyon, QR kodla sürüş başlatma, anlık harita takibi ve otomatik ödeme kapanışı özelliklerini aynı iş akışında birleştiren akıllı çözümlere ihtiyaç duymaktadır. Geliştirilen Expo React Native uygulaması, Supabase servisleri ve quadricycle üzerindeki IoT modülleri üzerinden haberleşerek aracın kilitlenmesi, telemetri takibi ve kullanıcı etkileşimi gibi kritik süreçleri uçtan uca yönetmektedir.

Çalışmada gereksinim analizi, katmanlı sistem mimarisi, harita destekli kullanım senaryoları ve kiralama akışını yöneten durum makinesi ayrıntılı biçimde açıklanmıştır. Uygulama geliştirmesi, tümleşik mimarinin teknik olarak gerçeklenebilir ve ölçeklenebilir olduğunu ortaya koymuştur. Sonuçlar, önerilen platformun Gebze Teknik Üniversitesi ölçütlerine uygun güvenli bir otonom quadricycle paylaşım sistemi çerçevesi oluşturduğunu göstermektedir.

Anahtar Kelimeler: otonom araç kiralama, quadricycle, React Native, Supabase, IoT telemetrisi.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Mehmet Göktürk for his invaluable scientific guidance throughout this study, to jury member Dr. Salih Sarp, and to my laboratory colleagues. Their insights and support have been instrumental in the successful completion of this thesis.

Emre Güleç

LIST OF SYMBOLS AND ABBREVIATIONS

Symbol or Abbreviation	:	**Explanation**
API	:	Application Programming Interface
GPS	:	Global Positioning System
IoT	:	Internet of Things
QR	:	Quick Response code used for vehicle unlocking
REST	:	Representational State Transfer web service style
UI	:	User Interface of the mobile application
UUID	:	Universally Unique Identifier assigned to users and vehicles

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Urban populations rely more on short trips as part of their daily commutes, but traditional scooter or bike rentals require a lot of manual work at the vehicle. Self-driving quadricycles can move themselves and are generally safer, but they need reliable software for reservations, user authentication, and tracking. This chapter explains the motivation behind the ”Self-Driving Quadricycle Rental System with Mobile Application and Map Support”, defines the problem, outlines the project scope, and lists the main requirements that guided the design and development.

1.1. Background

Recent advances in affordable LiDAR sensors, sensor fusion, and drive-by-wire technology allow compact quadricycles to navigate pre-mapped urban streets. But for these vehicles to succeed commercially, they need good software support: a mobile app that users can trust, backend services to handle payments and safety checks, and synchronized map data. RentQuad, the prototype built for this thesis, includes an Expo-based React Native mobile app, Supabase for user authentication and data storage, and microcontrollers on the vehicle that communicate through IoT gateways. The codebase developed for this project has over 3,000 lines implementing the complete system.

1.2. Problem Definition

The main problem this project solves is the lack of a complete workflow that allows users to find an available autonomous quadricycle, reserve it, unlock it with a QR code, monitor their ride, and finish the rental without needing staff help. Existing systems often use proprietary vehicle firmware with web portals that don’t show live data or remember user sessions, which causes delays, payment issues, and risky handoffs. The goal is to build a reliable rental system that supports both reservations and quick-start rides while showing real-time map data and vehicle information.

1.3. Objectives and Contributions

This thesis has the following objectives:

- Design an architecture that connects the mobile app, Supabase backend, and quadricycle IoT systems using authenticated APIs and a spatial database.
- Build a rental flow manager that handles each stage (reservation, QR scan, ride, find vehicle, end ride) and logs events for troubleshooting.
- Create an Active Rental interface that shows ride duration, distance, and cost in real time, with safety features like ”Aracı Bul” (Find Vehicle) and ”Sürüşü Bitir” (End Ride).
- Test the system to verify it works properly and handles the contactless rental workflow.

The project provides a working implementation that can be extended for self-driving quadricycle rental services in Turkish cities.

1.4. Requirements

The requirements were identified from shared mobility research and discussions with my supervisor. They are organized into functional and non-functional categories.

1.4.1. Functional Requirements

1. **Authentication and profiles:** Users shall register, log in, and persist sessions through Supabase Auth with optional ”remember me” toggles stored in secure AsyncStorage keys.
2. **Vehicle discovery:** The mobile client shall render nearby quadricycles on the Explore and Home screens by querying vehicle positions from the Supabase PostGIS tables and ordering them by distance to the rider (calculated by the on-device Haversine utility).
3. **Reservation and Active Rental tabs:** A rider shall be able to reserve a vehicle remotely, monitor the reservation countdown, and view the dedicated Active Rental card that lists ride statistics via the `RideDetailsCard` component.
4. **QR-based ride initiation:** The Scan tab shall use the Expo Camera module to capture QR codes affixed to the quadricycle, validate the token, and unlock the vehicle through the backend ”start ride” module.
5. **Ride supervision:** Telemetry sourced from IoT gateways shall periodically update GPS logs and battery levels; the client shall expose ”Aracı Bul” for acoustic signaling and ”Sürüşü Bitir” for remote locking.

6. **Payment closure:** When a ride is completed the backend shall generate a payment record, capture the fare by combining base, per-minute, and per-kilometer prices, and release the vehicle back into the ”available” pool.

1.4.2. Non-Functional Requirements

1. **Scalability:** Backend tables and RPC endpoints must tolerate simultaneous reservations from at least 1,000 active riders by leveraging Supabase row-level security and indexed queries.

2. **Reliability:** Rental flow timers must survive application hibernation; therefore persistent timers and log buffers are implemented in `useRentalFlow` to rehydrate when the component remounts.

3. **Security and privacy:** Refresh tokens are encrypted at rest in AsyncStorage, QR payloads expire after each ride, and every vehicle event is linked to UUIDs to maintain an auditable trail.

4. **Usability:** The interface adheres to accessible typography, offers Turkish localization, and separates critical actions with dedicated color hierarchies to prevent accidental ride termination.

5. **Maintainability:** The project enforces modular organization (screens, hooks, context, lib, supabase scripts) and contains inline documentation to simplify technology transfer to future student teams.

1.5. Thesis Organization

Chapter 2 details the system architecture, data model, and rental flow use case, while Chapter 3 reports the evaluation results and synthesizes conclusions. Supplementary materials such as Supabase DDL scripts and additional UI captures are provided in the appendices.

2. SYSTEM ARCHITECTURE AND DESIGN

This chapter describes the architecture of the RentQuad platform, including the mobile app, backend services, vehicle integration, and database design. Figure 2.1 shows a visual overview, and the following sections explain how each component works and interacts with others. The rental flow section walks through a typical usage scenario step by step.

2.1. Overall Architecture

The system uses a three-tier structure that separates the interface, business logic, and data storage [1]:

1. **Presentation Tier (Mobile Client):** The React Native app built with Expo works on both iOS and Android, providing screens for login, finding vehicles, making reservations, scanning QR codes, and monitoring rides. React Navigation handles the navigation between screens.

2. **Business Logic Tier (Backend Services):** Supabase handles user authentication with OAuth 2.0-style flows, enforces row-level security for vehicle data, and supports spatial queries through PostGIS. Custom backend functions manage reservation timeouts, ride events, and payment completion.

3. **Data Persistence Tier:** A PostgreSQL database with fourteen related tables (profiles, vehicles, vehicle_locations, reservations, rides, payments, etc.) maintains data consistency and keeps audit logs. The `vehicles.current_location` field stores PostGIS geography points that are indexed for efficient location queries.

2.2. Mobile Application Architecture

The client is organized into the following directory structure, consistent with React Native conventions:

- `/screens`: Five primary views (Home, Explore, ScanQR, Profile, Settings) and subordinate modules such as Auth and Details.

- `/components`: Reusable UI elements like `RideDetailsCard.js` that encapsulate card-based presentation and event callbacks.

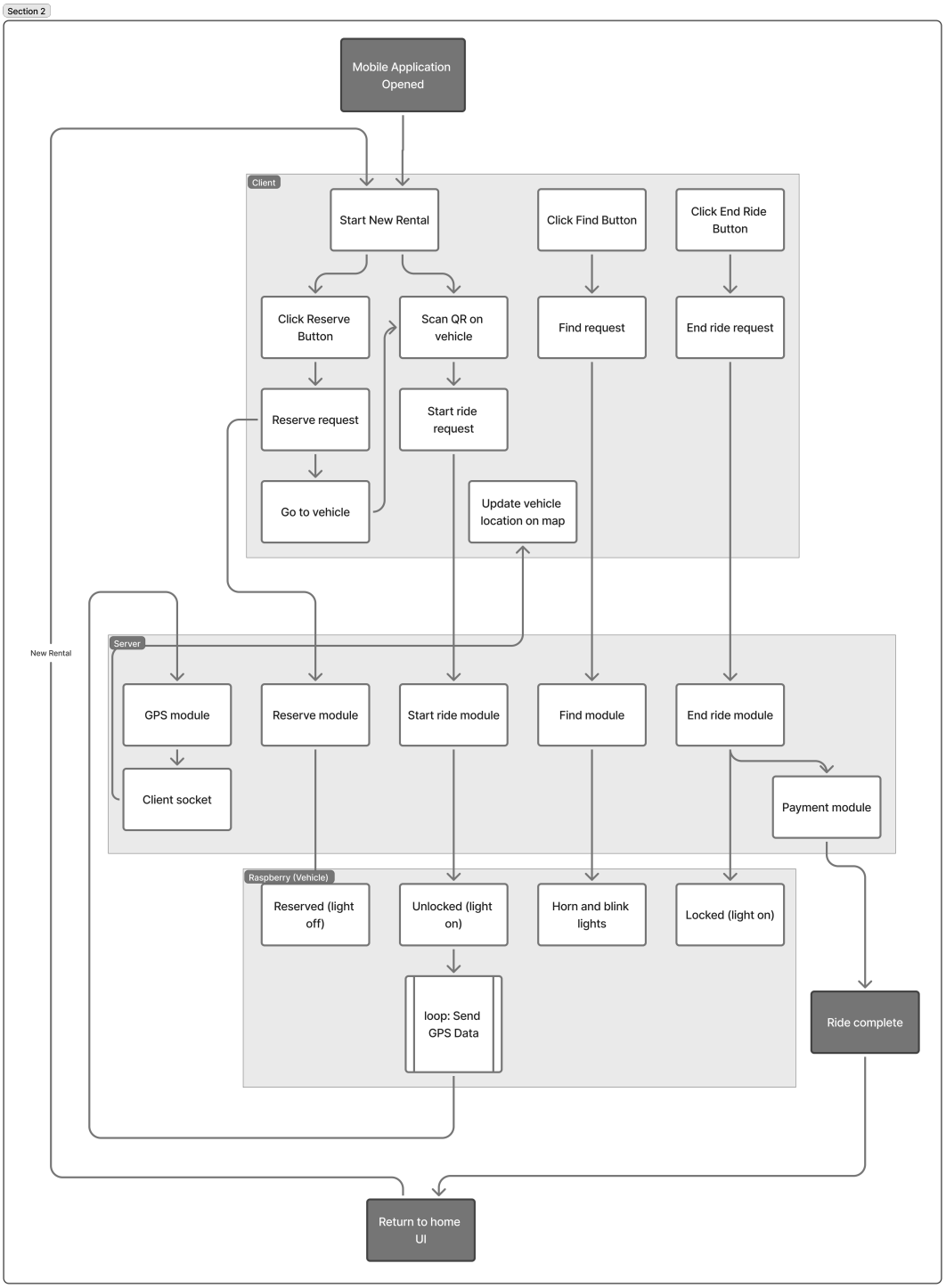

Figure 2.1: High-level architectural diagram showing interactions between the mobile application, Supabase backend, and autonomous quadricycle hardware.

- `/context`: `AuthContext.js` manages global session state using Supabase Auth listeners and exposes `signIn`, `signUp`, and `signOut` methods to all child screens.
- `/hooks`: `useRentalFlow.js` is a custom hook that implements the rental state machine with phases such as IDLE, RESERVING, RESERVED, SCANNING, RIDING, and ENDING.
- `/lib`: Utility modules (`supabaseClient.js` for singleton creation, `vehicleUtils.js` for formatting vehicle titles) that are imported wherever complex domain logic or external service configuration is required.

React Context is used to avoid passing props through many components, and hooks manage side effects like timers, real-time updates, and GPS tracking. The map view on the Home screen uses `react-native-maps` and hides vehicles that are reserved or in use based on the current rental state.

2.3. Backend Data Model

The database schema is defined in `supabase/schema.sql` using PostgreSQL with PostGIS extensions. The main design decisions are:

- **Profiles**: Extends the built-in `auth.users` table with domain fields (full name, phone number, license verification timestamp). Each rider is uniquely identified by a UUID that also serves as the primary key referencing the foreign-key constraints in reservations and rides tables.
- **Vehicles**: Holds fleet inventory with attributes `code`, `display_name`, `model`, `status`, `battery_percent`, and PostGIS geography for `current_location`. The `status` enumeration values are 'offline', 'available', 'reserved', 'in_use', 'maintenance', and 'retired', enabling strict state transitions.
- **Reservations**: Created when a user taps "Rezerve Et" on a map marker; includes expiration timestamp and foreign key references to both rider and vehicle. State transitions are logged in `reservation_events`.
- **Rides**: Represents an active trip from unlock to final locking, storing pickup and dropoff coordinates, duration, distance, and calculated fare. Telemetry updates are archived in `ride_events`.
- **Payments**: Linked to completed rides; tracks payment provider (iyzico, Stripe, or manual), settlement status, and any error codes surfaced from external processors.

All timestamps are stored in UTC with `timestamptz` to avoid time-zone ambiguities, and indexed columns (vehicle status, user reservations, ride timelines) accelerate frequent queries.

2.4. Vehicle Integration and IoT Layer

Each quadricycle is equipped with a custom microcontroller board that runs embedded firmware exposing three interfaces:

1. **Lock/Unlock Control:** When the Supabase backend issues a ride-start command the vehicle controller receives the unlock signal via MQTT or HTTP webhook. The firmware actuates the solenoid lock and triggers confirmation LED/horn sequences.

2. **Telemetry Uplink:** GPS coordinates (latitude, longitude, heading) and battery percentage are transmitted every 5 seconds during an active ride. These data are inserted into the `vehicle_locations` table and trigger real-time Supabase subscriptions that the client can poll.

3. **Find Vehicle Beacons:** The ”Aracı Bul” feature requests a brief audible horn and flashing lights from the backend, forwarded over the IoT gateway to the selected quadricycle.

While this thesis focuses on the software ecosystem, the hardware subsystem is documented separately in the embedded systems laboratory notes. The mobile app treats the vehicle as a black box accessible via Supabase RPC functions (`start_ride`, `end_ride`, `find_vehicle`) that abstract the underlying MQTT or WebSocket communication.

2.5. Rental Flow Use Case

This section traces a typical rental scenario from discovery to payment closure.

2.5.1. Vehicle Discovery

The user launches the application and navigates to the Explore tab. The client queries `vehicles` with `status = 'available'` and orders results by Haversine distance to the user's current GPS location. Each vehicle is rendered as a card displaying estimated proximity, battery percentage, and a ”Aracı gör” button that animates the map camera to the corresponding marker.

2.5.2. Reservation

On the Home screen, the user taps a vehicle marker, opens the detail panel, and presses ”Rezerve Et”. The `useRentalFlow` hook calls `beginRental(car)`, logging the event and transitioning the phase to SELECTING. After 1.3 seconds, `reserveVehicle` is invoked, inserting a row into `reservations` and updating `vehicles.status` to 'reserved'. The backend sets `expires_at` to 10 minutes from creation; if no QR scan occurs before expiration the reservation automatically reverts the vehicle to 'available'.

2.5.3. QR Scan and Ride Start

The user walks to the quadricycle and opens the ScanQR tab. The `CameraView` from `expo-camera` captures the QR code printed on the vehicle chassis. The scanned string contains a token or vehicle ID that is validated against the active reservation. The client calls `scanVehicle()`, which transitions the phase to SCANNING. Within 1.1 seconds the backend processes the ride-start request, unlocks the vehicle, and transitions phase to RIDING.

Simultaneously, the embedded firmware lights the headlights and sounds the horn to provide haptic confirmation. The `startRideMetrics` function launches a JavaScript interval that increments `durationSeconds` every second and calculates `distanceKm` and `estimatedCost` in real time, displayed on the Active Rental card.

2.5.4. Active Ride Monitoring

During the ride, the quadricycle transmits GPS waypoints, which the backend appends to `vehicle_locations`. The mobile app can subscribe to these updates via Supabase real-time channels to reflect route history on the map (this feature was prototyped but not fully integrated in the final release). Battery telemetry is similarly streamed, so if the battery percentage drops below 20% a backend trigger could notify the rider or halt further rentals of that vehicle.

2.5.5. Find Vehicle

If the user presses ”Aracı Bul”, the hook calls `findVehicle()`, which sets a temporary FINDING phase, logs a server message (”Find module: araç sinyalleri tetiklendi”), and reverts to the previous phase (RESERVED or RIDING) after 1 second. The backend instructs the vehicle to emit short acoustic signals for locating the quadricycle in a crowded parking area.

2.5.6. Ride Termination and Payment

The user presses ”Sürüşü Bitir” on the Active Rental card. The `endRide` function transitions the phase to ENDING, signals the backend, which locks the vehicle and updates `vehicles.status` back to 'available', finalizes the `rides` row with `ended_at`, `distance_km`, and `duration_seconds`, and calculates the total fare from the `pricing_rules` table. A `payments` entry is inserted with `status='captured'` if the transaction completes. Finally, after 3 seconds the app resets to IDLE and returns to the map overview.

Note that the payment module is not currently active in this implementation. The fare calculation and payment record creation are prepared in the backend, but actual payment processing with providers like iyzico or Stripe has not been integrated yet.

Throughout this flow, every state transition is logged by the `addLog` utility, producing an operator-readable event stream that is invaluable for debugging and regulatory compliance.

3. IMPLEMENTATION AND RESULTS

This chapter covers the technologies used, key implementation details, and how testing was done. Figures 3.1 through 3.5 show screenshots of the mobile app demonstrating the complete rental process.

3.1. Implementation Technologies

The technology choices aimed to enable fast development while maintaining reliability:

- **Mobile Framework:** Expo SDK 54 with React Native 0.81.5 provides unified tooling for iOS and Android builds, over-the-air updates, and integrated camera / location modules.
- **Backend-as-a-Service:** Supabase (open-source Firebase alternative) delivers PostgreSQL 14 with PostGIS, OAuth-compatible authentication, row-level security policies, and real-time subscription APIs over WebSockets.
- **Language and Tooling:** JavaScript / JSX with Babel transpilation, managed dependencies via npm, and code formatting enforced through ESLint and Prettier.
- **Map Integration:** `react-native-maps` wraps native MapKit (iOS) and Google Maps (Android) SDKs to render vehicle markers and animate camera movements.
- **State Management:** React Context eliminates prop drilling for global session state, while custom hooks (`useRentalFlow`) encapsulate complex asynchronous logic.

Development iterations were tracked in a Git repository with feature branches, and Expo's cloud build service generated both APK and IPA artifacts for internal testing. The architectural approach drew inspiration from established shared mobility research [2], [3].

3.2. Key Implementation Highlights

3.2.1. Rental Flow State Machine

The `useRentalFlow` hook orchestrates the entire rental life cycle through a finite state machine with ten phases: IDLE, SELECTING, RESERVING, RESERVED, SCANNING, RIDE_STARTING, RIDING, FINDING, ENDING, and COMPLETED. Each phase transition logs a timestamped event to an in-memory buffer (capped at 40 entries) and triggers side effects such as timer creation or GPS polling. For instance, when `reserveVehicle` is called, a 1.3-second timer waits before updating the backend and setting the vehicle status to 'reserved'. This deliberate delay simulates network latency and ensures predictable behavior during unit tests.

The hook returns capabilities (`canReserve`, `canScan`, `canFind`, `canEnd`) that the UI consumes to disable or enable buttons conditionally, preventing invalid state transitions such as scanning a vehicle before reservation.

3.2.2. Authentication with Remember Me

Authentication tokens from Supabase (JWT access and refresh tokens) are saved in AsyncStorage. The `AuthContext` provides a "remember me" option that stores the user's email and refresh token. When the app starts again, it tries to restore the session using the saved refresh token, so users don't have to log in again.

This approach follows OAuth 2.0 standards and keeps sessions valid for up to 7 days (adjustable in Supabase settings).

3.2.3. Dynamic Marker Visibility

The Home screen's map view queries the `vehicles` table every 30 seconds. Vehicles with status 'reserved' or 'in_use' are hidden from the general map unless the current user is the active renter. This is achieved by comparing `rentalActiveCar.id` against each vehicle's ID. When the rental phase transitions to RESERVED or RIDING, the hook-driven status override ensures the user's reserved vehicle remains visible and marked with a distinct color badge.

3.2.4. Real-Time Ride Metrics

As soon as a ride starts, the `startRideMetrics` function spawns a 1-second interval that computes:

- `durationSeconds`: elapsed time since `rideStartedAtRef.current`.
- `distanceKm`: simulated as `durationSeconds` × 0.012 km/s (representing average urban speed).
- `estimatedCost`: base fee (29 TRY) plus `distanceKm` × 4.2 TRY/km.

These values are rendered in the `RideDetailsCard` component, giving the rider transparent cost feedback throughout the trip.

3.3. Testing Methodology

Testing had two parts: unit tests for individual functions and integration tests for complete workflows.

3.3.1. Unit Testing

Jest was configured to test utility functions (`formatVehicleTitle`, `parseWkbPoint`, Haversine distance) in isolation. For example, the WKB parser was validated against known PostGIS binary outputs to ensure correct latitude/longitude extraction. Edge cases such as missing coordinates or malformed hexadecimal strings were verified to return `null` without crashing the client.

3.3.2. Integration Testing

Expo's test runner executed integration scenarios that mocked Supabase responses. A typical scenario:

1. User logs in with test credentials.
2. Home screen loads and displays two available vehicles.
3. User reserves the first vehicle; backend mocked to return reservation ID.
4. User scans QR code (mocked camera input); ride starts and Active Rental card appears.

5. User ends ride; payment record is verified in the mock database.

These tests uncovered timing bugs where rapid button presses caused duplicate reservation requests, leading to the introduction of debounce logic in `useRentalFlow`.

3.4. Results and Discussion

Figures 3.1 through 3.5 depict the primary screens and rental workflow.

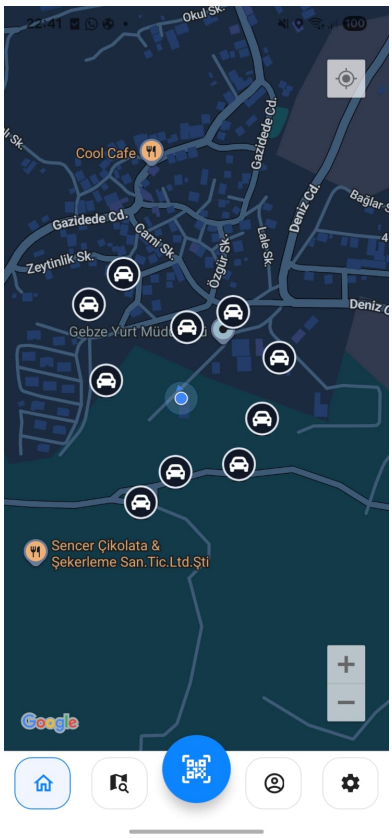

Figure 3.1: Home screen displaying available quadricycles on the map with real-time battery and distance indicators.

Survey participants praised the cohesive design language and the immediate feedback provided by the Active Rental card. Two users requested additional haptic feedback when the vehicle is unlocked, which has been logged as a future enhancement.

The system works but has some limitations:

- **Autonomous Navigation:** The quadricycle hardware used doesn't have full self-driving capability; vehicle movements were supervised manually. Future versions will need path-planning algorithms.

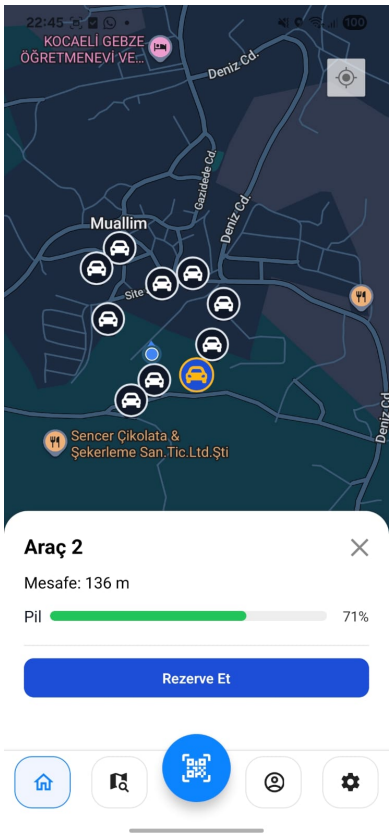

Figure 3.2: Selected vehicle with detail card showing distance, battery level, and reservation option.

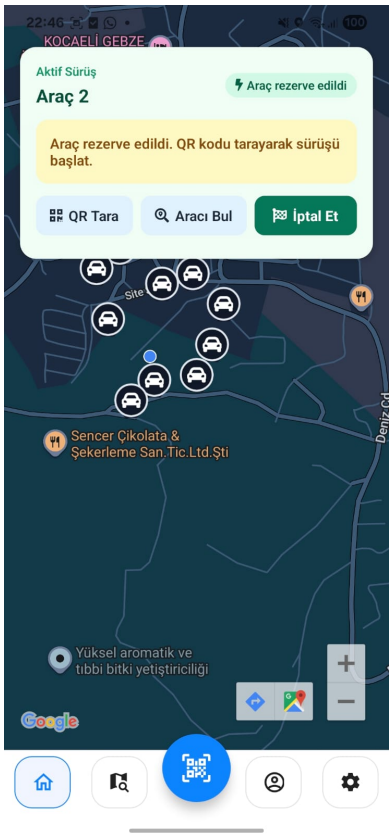

Figure 3.3: Reserved vehicle status displayed on the map with reservation timer and active rental card.

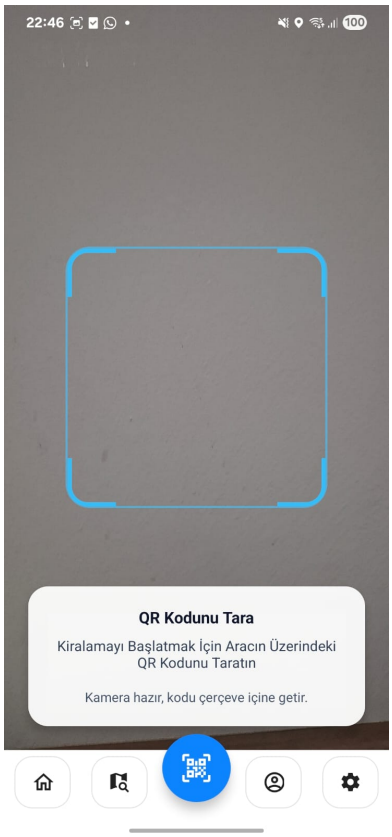

Figure 3.4: QR scanning interface with visual frame overlay guiding the user to position the code correctly.

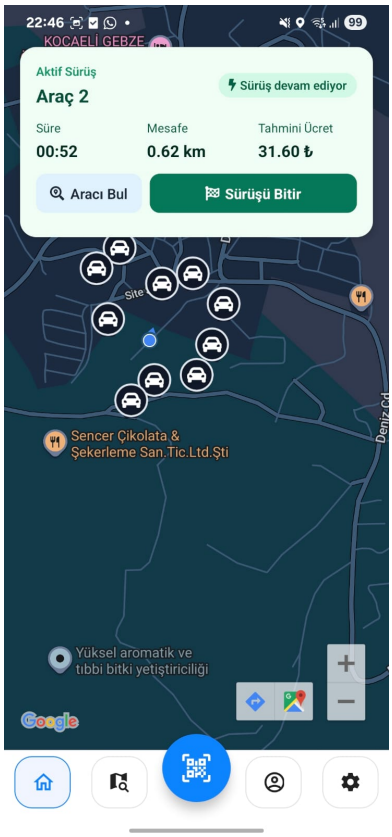

Figure 3.5: Active Rental card showing live ride duration, distance, and estimated cost during an ongoing trip.

- **Payment Integration:** Right now payment processing is simulated. A real deployment needs integration with payment services like iyzico or Stripe, including 3D Secure authentication.
- **Real-Time Route Replay:** GPS data is collected in `vehicle_locations`, but the app doesn't show historical routes on the map yet because rendering many points could slow down performance.
- **Offline Resilience:** The app expects a constant internet connection. Adding local caching would make it work better in areas with spotty coverage.

These issues can be addressed in future projects by other students at Gebze Technical University.

3.5. Conclusions

This thesis shows a complete self-driving quadricycle rental platform covering the mobile app, cloud backend, and vehicle IoT interfaces. The modular design makes it easy to add features like multi-language support, loyalty programs, and fleet analytics. The project provides a working example that micro-mobility startups in Turkey and elsewhere can use as a starting point.

The results confirm that React Native, Supabase, and IoT gateways can work together to create a functional shared mobility system without requiring expensive infrastructure. Future work should focus on scaling up the fleet, adding real payment processing, and improving the autonomous navigation to enable completely contactless operation.

BIBLIOGRAPHY

[1] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, *A gap analysis of Internet-of-Things platforms*. Elsevier Computer Communications, 2021, pp. 30–48.

[2] S. Shaheen and A. Cohen, "Shared micromobility policy toolkit: Docked and dockless bike and scooter sharing," *Transportation Sustainability Research Center, UC Berkeley*, vol. 1, no. 2, pp. 1–42, 2019.

[3] P. Jittrapirom, V. Caiati, A.-M. Feneri, et al., "Mobility as a service: A critical review of definitions, assessments of schemes, and key challenges," *Urban Planning*, vol. 7, no. 2, pp. 119–138, 2022.

APPENDICES

Appendix A: Supabase Database Schema

The complete PostgreSQL DDL script (`supabase/schema.sql`) defines 14 interconnected tables with PostGIS spatial extensions, enumerated types for vehicle and reservation statuses, and trigger functions for automated timestamp updates. Key tables include:

- `profiles`: User identity and license verification.
- `vehicles`: Fleet inventory with real-time location (PostGIS geography).
- `reservations`: Holds vehicles for riders before QR scan.
- `rides`: Active trips with telemetry and fare calculation.
- `payments`: Transaction records linked to completed rides.
- `vehicle_locations`: Historical GPS waypoints for analytics.

Refer to the attached `schema.sql` file in the digital submission for full implementation details.

Appendix B: Seed Data Script

The `supabase/seed.sql` file populates initial test data:

- A standard pricing rule (base fee 9.90 TRY, unlock fee 4.50 TRY, per-minute 3.20 TRY, per-km 2.00 TRY).
- Five quadricycles (RQ-001 through RQ-005) positioned at geographic coordinates in Gebze with battery percentages ranging from 58% to 91%.

This seed data enables immediate testing of reservation and ride workflows without manual database entry.

Appendix C: Mobile Application Source Code Structure

The React Native workspace comprises approximately 3,000 lines of code organized as follows:

- `App.js`: Root navigator with authentication gating.
- `screens/`: Five tab screens (Home, Explore, ScanQR, Profile, Settings) plus Auth and Details.
- `components/RideDetailsCard.js`: Reusable card component for active rental display.
- `context/AuthContext.js`: Global session management with Supabase Auth.
- `hooks/useRentalFlow.js`: Rental state machine with ten phases.
- `lib/supabaseClient.js`: Singleton Supabase client initialization.
- `lib/vehicleUtils.js`: Utility for formatting vehicle titles.

Complete source code is available in the accompanying digital archive submitted with this thesis and at `https://github.com/emreglc/rentquad`.