RideShare Architecture Documentation

Mayer Lukas, Radic Ivo, Sarzi Sartori Nicoletta, Veigel David

Table of Contents

Introduction and Goals

Requirements Overview

The main purpose of RideShare is to provide a convenient, affordable, and sustainable transportation solution by connecting passengers with nearby drivers for shared rides.

Title	Description
User Registration and Profile Management	Users can register, create, and update their profiles.
Ride Search and Booking	Users search for available rides and book one.
Real-Time Ride Tracking	Users can view the current location of the vehicle in real time.
In-App Payment System	Payments are processed securely within the app.
Driver Verification	Drivers are verified before activation (e.g., ID check).
Rating and Review System	Users and drivers can rate and review each other after a ride.

Quality Goals

Priority	Quality	Motivation
1	Reliability	If our app is down we will lose potential costumers to the competition
2	Security	If our system gets compromised we will lose our reputation and therefore customers
3	Usability	If our users don’t understand how our app works they will switch to an alternative

Stakeholders

Role/Name	Expectations
Investor / John	*Growth in share valueProfitability*
Developer Team	*Clear deadlinesClear requirementsEnough resources*
Compliance Regulator	*Set regulations and licensing*
Driver Representative	*Fair compensationTransparency*
Marketing Team	*BudgetMarket & user dataCreative freedom*

Architecture Constraints

Constraints	Background and/or motivation
Using low-cost Api's (OpenStreetMap)	Helps reduce recurring costs from expensive APIs like Google Maps.
Must comply with GDPR	App handles personal and location data of EU users, so legal compliance is essential.
Deploy on cloud infrastructure (e.g. AWS)	Enables scalability, elasticity, and global availability of microservices and storage.
Must use React Native	Allows cross-platform development (iOS & Android) with one codebase — reduces dev time and costs.
Stick to one deployment region	Minimizes cloud billing complications and keeps performance manageable for a local audience.

System Scope and Context

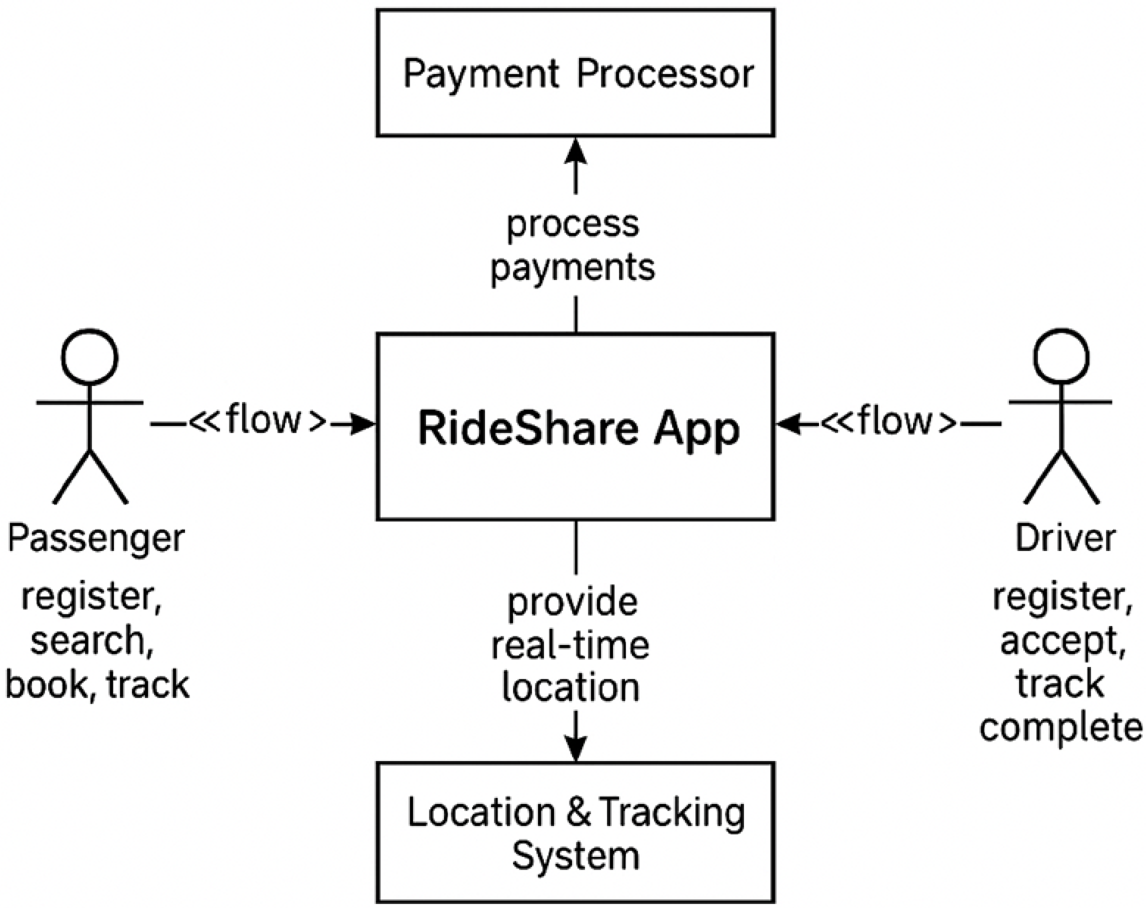


Figure 1: Business Context Image

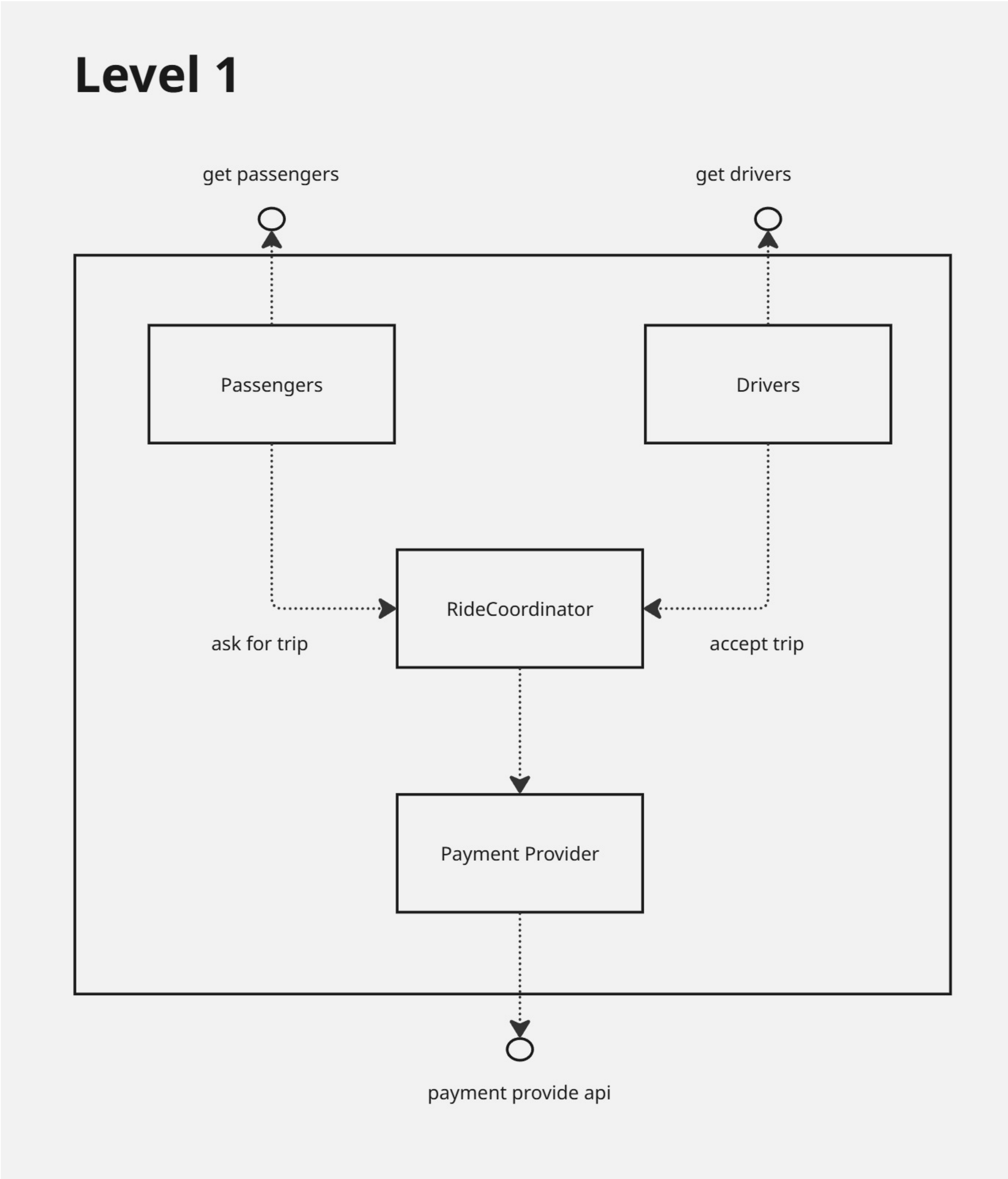
Solution Strategy

Context	Decision	Consequences
We need to secure communication and manage user sessions across mobile and backend.	We will implement JWT-based authentication and role-based access control.	Improves security and enables scalable session handling, but requires careful token management and role definition.
We need scalable and independently deployable components for each service (e.g. ride handling, user profile, payments).	We will use a microservices architecture.	Each service can be developed and scaled independently, but requires DevOps setup, inter-service communication, and monitoring overhead.
Deploy on cloud infrastructure (e.g. AWS)We want to maintain a clean separation between front-end logic, backend services, and data access.	We will use the Model-View-Controller (MVC) pattern.	Code will be more maintainable and testable, but may require stricter discipline in structuring features.

Building Block View

White Box: Overall System

Overview Diagram



Motivation

At the top level, the system is divided into four key building blocks: - Passengers - Drivers - Ride

Coordinator - Payment Provider

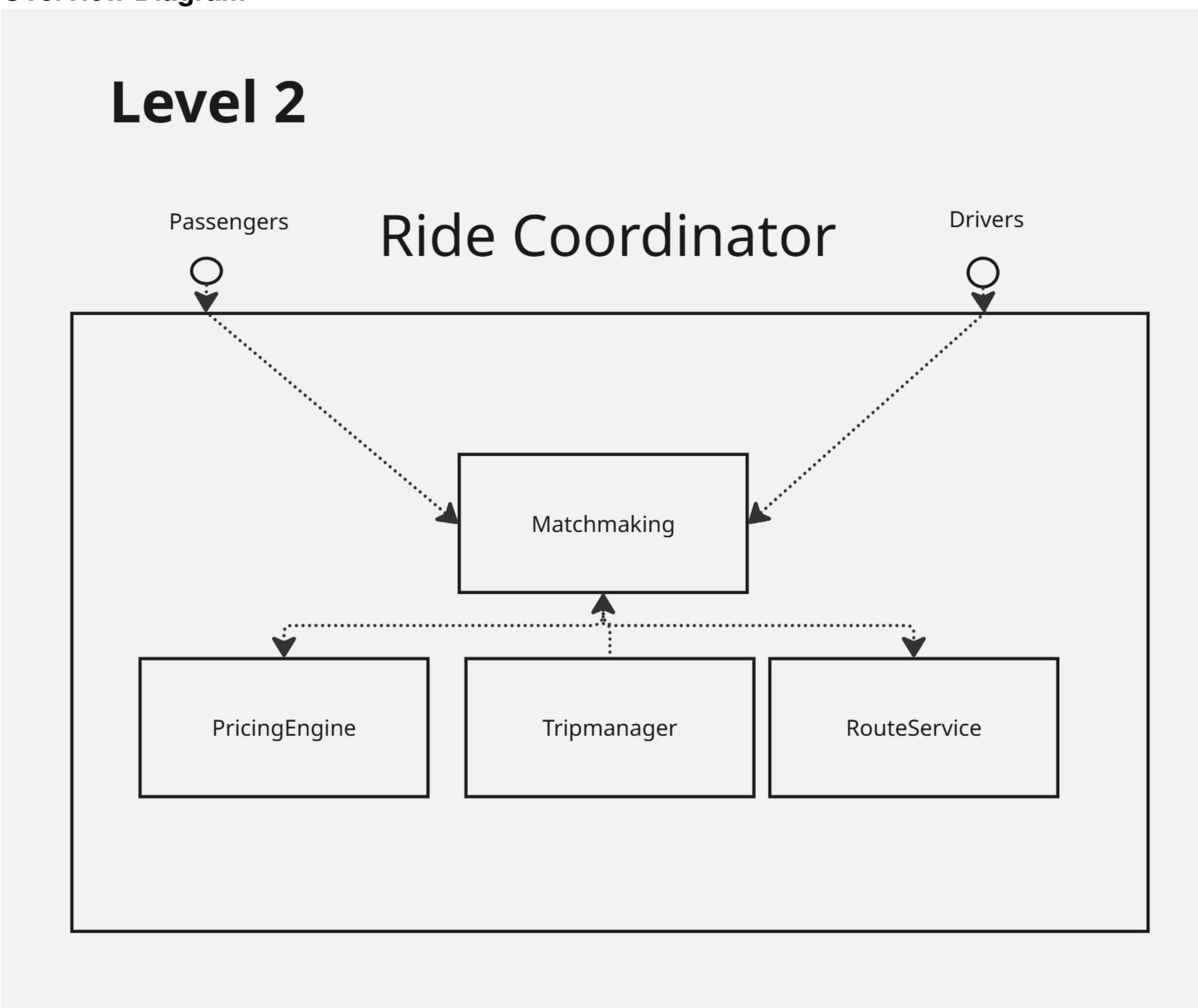
This clear separation of concerns supports scalability and simplifies maintenance.

Contained Building Blocks

Name	Responsibility
Passengers	Enables passengers to request trips and communicate with the system.
Drivers	Enables drivers to accept trips and provide transportation services.
Ride Coordinator	Orchestrates trip requests, driver matching, and payment processing.
Payment Provider	Manages secure payment transactions with external payment services.

White Box: Ride Coordinator

Overview Diagram



Motivation
The Ride Coordinator is further decomposed into focused services: - Matchmaking - Pricing Engine - Trip Manager - Route Service

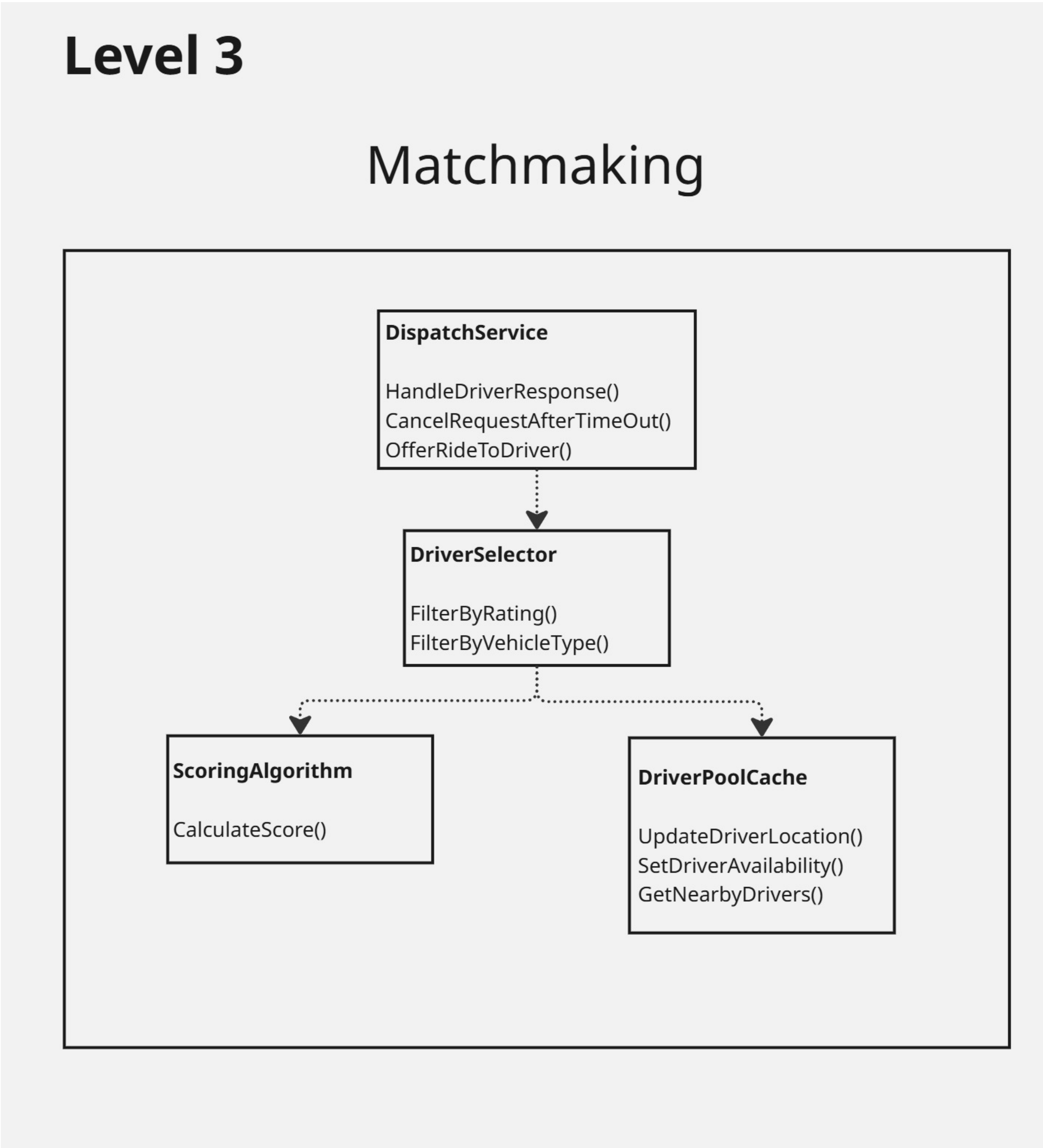
This modularization allows independent scaling and development.

Contained Building Blocks

Name	Responsibility
Matchmaking	Matches passengers with suitable drivers.
Pricing Engine	Dynamically calculates trip fares based on distance, demand, etc.
Trip Manager	Manages the trip lifecycle (creation, updates, completion).
Route Service	Provides optimal routing and navigation for trips.

White Box: Matchmaking

Overview Diagram



Motivation

The Matchmaking component is detailed to show its internal structure for optimal driver-passenger pairing.
It consists of dispatch logic, driver selection, scoring, and real-time driver pool management.

Contained Building Blocks

Name	Responsibility
Dispatch Service	Manages driver responses, cancellations, and offers rides to drivers.
Driver Selector	Filters drivers by rating, vehicle type, and other constraints.
Scoring Algorithm	Calculates scores for potential matches to optimize match quality.
Driver Pool Cache	Keeps real-time data of driver locations, availability, and proximity searches.

Important Interfaces

- **Passenger API:** Interface for passengers to request trips and receive status updates.
- **Driver API:** Interface for drivers to accept/reject trips and update availability.
- **Payment Provider API:** Secure integration with external payment gateways for processing payments.

Runtime View

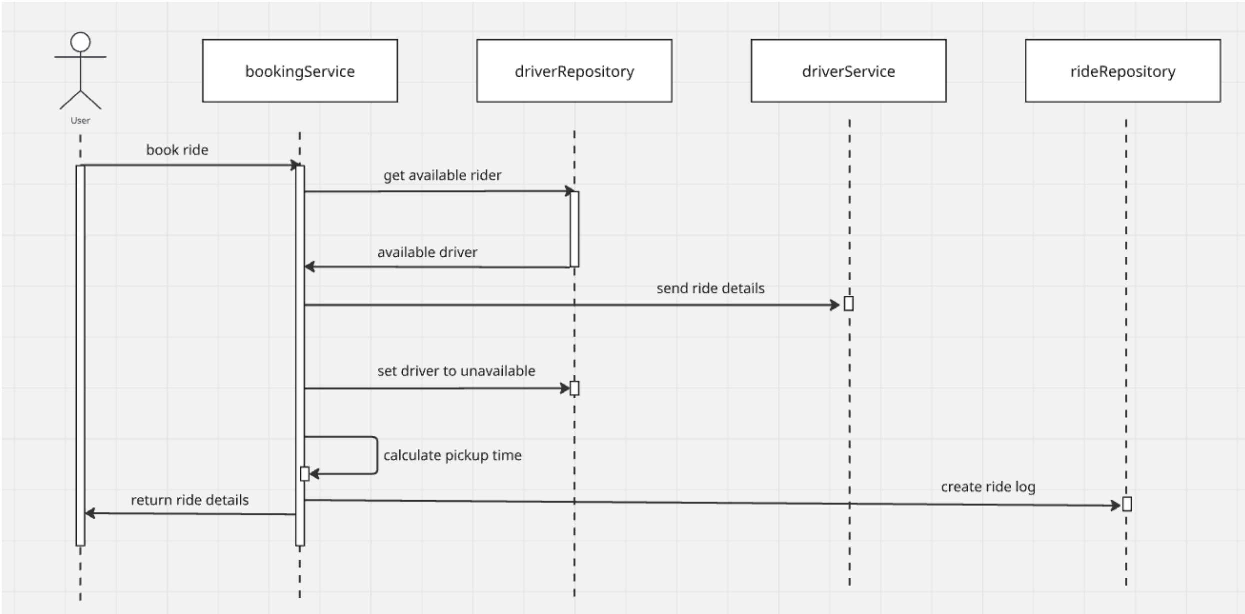


Figure 2: Runtime View Image

Deployment View

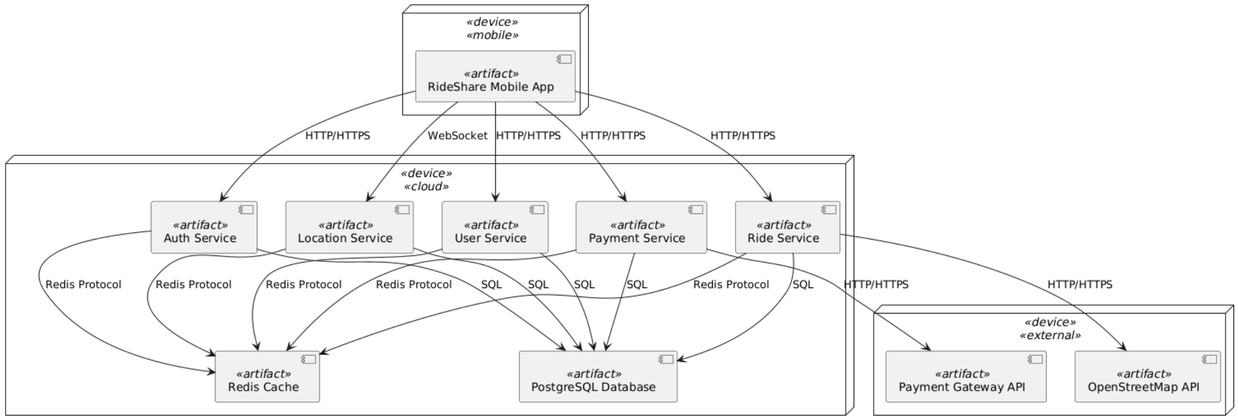


Figure 3: deployment_view.png

Cross-cutting Concepts

Component-based Microservice Architecture

Our system is structured using a component-based approach with microservices. This promotes loose coupling, independent scalability, and easier development. Each service encapsulates a specific business capability and can be deployed independently, supporting team autonomy and fault isolation.

Model-View-Controller (MVC)

We use the MVC pattern to separate user interface, business logic, and data handling across both the frontend and backend layers. This increases maintainability and testability by keeping concerns cleanly divided and responsibilities clearly assigned.

Event-driven Architecture

For internal communication and workflow orchestration (e.g. ride matching and notifications), we implement event-driven architecture using asynchronous messaging. This helps decouple services, improves scalability, and allows the system to react in near real-time.

JWT Authentication and RBAC

Security is enforced using JWT (JSON Web Token) for stateless authentication across services. Role-based access control (RBAC) is used to restrict access to protected resources, depending on whether the user is a driver, rider, or admin.

Input Validation and Sanitization

To prevent common vulnerabilities such as injection attacks or malformed input, we apply strict validation rules and sanitize all incoming data at the controller level. This protects internal services and the database layer from untrusted input.

Cross-cutting Concepts

Problem	Considered Alternatives	Decision
How to implement real-time communication between rider and driver	1. REST API polling2. WebSockets3. Realtime Database	Use WebSockets for bi-directional communication to support live ride status updates and driver location tracking
How to scale the system to handle sudden demand spikes (e.g. during events)	1. Manually scale servers2. Use Docker with Kubernetes	Chose Kubernetes for better control and autoscaling
How to store and manage geolocation data efficiently	1. Relational DB with spatial extension (PostGIS)	Decided on PostgreSQL with PostGIS for reliable geospatial queries

Quality Requirements

Quality Tree

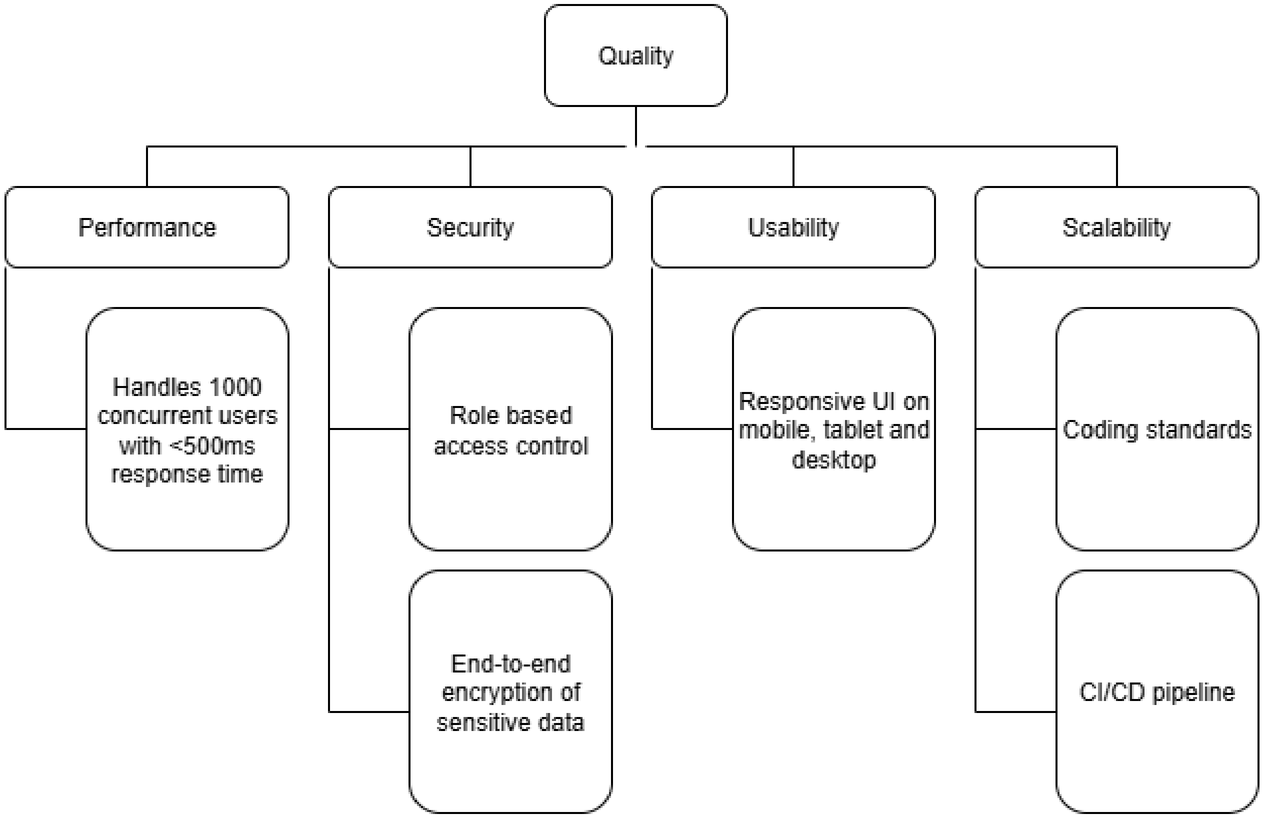


Figure 4: quality_tree.png

Quality Scenarios

Reliability
A user updates their account email address. Due to a bug in the email service, the confirmation email fails to send — but the system logs the issue, queues the email for retry, and does not corrupt the user's data or prevent further account actions.

Security
A malicious actor attempts unauthorized access to customer data via a public API. The system immediately blocks the request, logs the attempt, and sends an alert to the security team within 5 minutes.

Usability
A first-time user opens the app and successfully completes profile setup without external guidance, within 10 minutes, aided by intuitive navigation and contextual help prompts.

Risks and Technical Debts

Risk / Technical Debt	Description
Outdated Dependencies	Using outdated libraries or frameworks may lead to security vulnerabilities and compatibility issues.
Service Coupling Risk	Some services may become tightly coupled due to shared logic or data dependencies, making refactoring or scaling more difficult.
Lack of Automated testing	Initial development may skip full unit/integration test coverage due to time pressure, increasing the risk of undetected bugs and regressions.

Glossary

Term	Definition
Ride	A confirmed trip where a driver transports a rider to a destination.
JWT	JSON Web Token, used for secure stateless authentication.
RBAC	Role-Based Access Control; access permissions assigned by user role.
Microservice	An independently deployable component that handles one domain function.
Event-Driven	System design where components communicate via events rather than direct calls.
Matching	The process of finding a nearby available driver for a ride request.
ETA	Estimated Time of Arrival; used to inform riders of driver arrival time.
Pricing Engine	Component responsible for fare calculation based on distance and demand.
Payment Gateway	External service used to process in-app transactions securely.
PostGIS	A spatial database extension for PostgreSQL used for geolocation queries.