Template

2021-02-22

About arc42

arc42, the Template for documentation of software and system architecture.

By Dr. Gernot Starke, Dr. Peter Hruschka and contributors.

Template Revision: 7.0 EN (based on asciidoc), January 2017

This version of the template contains some help and explanations. It is used for familiarization with arc42 and the understanding of the concepts. For documentation of your own system you use better the *plain* version.

Introduction and Goals

The development of the software architecture for the ridesharing app "TripLink" requires consideration of several critical requirements and driving forces. Underlying business goals: The main objective of "TripLink" is to transform traditional transportation, reduce costs, and minimize environmental impacts. At the same time, the company aims for global presence and fair partnerships with drivers. Key challenges: The app must provide user-friendly interfaces, real-time GPS tracking, secure payment processing, route optimization, a reliable rating system, dynamic pricing, and environmentally friendly vehicle options. Essential functional requirements: Continuous connectivity, data privacy, efficient driver-passenger matching, and responsive customer support are fundamental. The app should also be usable on various platforms. Quality goals for the architecture: The architecture must ensure high availability to minimize downtime, be scalable to handle peak loads, and provide robust security measures to protect user data and transactions. Relevant stakeholders and their expectations: Different stakeholders have different expectations. Passengers expect convenience and safety, drivers seek fair compensation and job security, investors aim for profitable investments and long-term growth, regulatory authorities demand compliance with local laws and safety standards, and the development team strives for innovation and system stability. These requirements and expectations form the foundation for the success and evolution of "TripLink."

Requirements Overview

- Efficient and secure user registration and profile creation.
- Capability for users to search and book rides based on location, destination, and time.
- Real-time tracking of rides for passengers and drivers.
- Integrated payment system for fare handling.
- System for verifying and approving drivers.
- Mechanisms for users to rate and review each other.
- Support system for user queries and feedback.
- Analytics for ride data and user behavior.

Quality Goals

Priority	Quality	Motivation
1	User Experience	The app must provide an intuitive, user-friendly interface to ensure user satisfaction and retention. It should have a fast and straightforward registration and ride-booking process.
2	Performance	The app should be able to handle high data traffic and ensure that real-time updates on ride availability and estimated arrival times are reliable and accurate. Latency should be minimized, and uptime should be at least 99.9%.
3	Security	High-quality security measures must be implemented to protect user data.

Stakeholders

Identification and understanding of stakeholders are crucial to avoid later surprises and conflicts in the development process. Stakeholders not only influence the direction of the project but also significantly determine the success and acceptance of the ridesharing system.

The following table provides insight into the various stakeholders and their specific expectations and needs for the system architecture of the ridesharing project:

Role/Name	Expectations
Investor (Maryam Patel)	Concerned with the financial success of the app and its scalability. Will be interested in key performance indicators (KPIs) and return on investment (ROI).
Business Development Manager at Local Transportation Authority (Raj Gupta)	Interested in how the app integrates with existing transportation infrastructure and adheres to regulations.
User Experience Designer (Megan Chen)	Focused on ensuring the app's design is user-centric and intuitive, ultimately aiming for high user satisfaction and retention rates.
RideShare Driver and Representative of RideShare Driver Association (Amirah Rahman)	Concerned with driver welfare, fair pay, and the overall operation process from the driver's perspective.
Environmental Activist and Representative of Sustainable Transportation NGO (Javier Gomez)	Interested in the app's potential to reduce emissions and traffic congestion, and how it contributes to sustainable transportation goals.

Architecture Constraints

In the development of the TripLink a microservices architecture was chosen, because of the dynamic and scalable nature of the platform. This decision was driven by the need for a highly flexible, resilient, and independently scalable system. Microservices architecture allows us to rapidly deploy new features, scale specific functions of the app in response to varying load and maintain different components of the app with minimal impact on the overall system. It also supports our goal of creating a robust and efficient ride-sharing platform by enabling continuous integration and deployment, facilitating agile development practices, and ensuring that each part of the application can evolve at its own pace.

1. Network Dependency:

- **Explanation**: Microservices communicate over a network, which introduces latency and the potential for network failure.
- **Impact**: The design must include strategies for dealing with partial network failures and ensuring robust communication between services.

2. Data Consistency:

- **Explanation**: Each microservice can have its own database, leading to challenges in maintaining data consistency across services.
- **Impact**: Implementing eventual consistency, transactional patterns like SAGA, or using a distributed database system can become necessary.

3. Service Discovery:

- **Explanation**: In a dynamic environment with services scaling up and down, keeping track of service instances is crucial.
- **Impact**: Requires implementing a service discovery mechanism to allow services to find and communicate with each other.

4. Deployment Complexity:

- **Explanation**: Deploying multiple independent services can be more complex than deploying a single monolithic application.
- **Impact**: Need for automated deployment processes, using CI/CD pipelines and container orchestration tools like Kubernetes.

5. Performance Overhead:

- **Explanation**: The use of multiple databases and communication over the network can introduce performance overhead.
- **Impact**: Optimization of service calls and database queries becomes crucial, and caching strategies may be needed to improve performance.

6. Operational Overhead:

- **Explanation**: Monitoring and logging for many services can be more complex than for a monolithic application.

- **Impact**: Implementation of centralized monitoring and logging solutions to track the health and performance of all services.

7. Skillset Requirements:

- **Explanation**: Microservices require a team with a diverse set of skills, including DevOps, cloud computing, and various programming languages.
- **Impact**: The need for skilled professionals and potentially more training or hiring to cover the range of required expertise.

System Scope and Context

Contents: The RideShare system is designed to connect passengers with drivers for shared rides. It integrates with mapping services for route planning and navigation, and payment gateways for transaction processing. The business context involves ride-sharing and efficient urban mobility, while the technical context covers APIs, protocols, and infrastructure necessary for location tracking, payment processing, and data management.

Motivation: Understanding the system's interaction with mapping and payment services is crucial, as these are key components that directly impact user experience, ride efficiency, and the financial transactions within the RideShare app.

Business Context

Contents: RideShare interacts with various entities for its operations. Users (both drivers and passengers) provide ride requests and accept rides through the app. Payment services handle fare transactions. Mapping and GPS services are used for route navigation and ride tracking. A customer support system handles user queries and issues. Data storage is managed by cloud service providers.

Communication Partners:

Partner	Inputs from RideShare	Outputs to RideShare
User	Ride Requests, location	Ride matches, navigation
Payment Service	Fare details	Payment confirmations
Mapping Service	Destination, current location	Route data, ETA
Support System	User queries, feedback	Support responses, updates
Hosting Partner	Data processing requests	Access to data storage, processing capabilities

Motivation: To clarify the information flow and system integration with external services, which is essential for the app's operational effectiveness and user satisfaction.

Technical Context

Contents: TripLink, leveraging a microservices architecture, employs both Google Cloud Pub/Sub for inter-service communication via Remote Procedure Calls (RPC) and HTTPS for external API interactions. This setup ensures efficient, scalable communication within the system and secure, reliable data exchange with external services.

Domain Interface Technical Channel Protocol:

Domain	Interface	Technical Channel	Protocol
User Management	User Service-API	HTTPS	RESTful API
Ride Coordination	Ride Service-API	GCP Pub/Sub	RPC
Payment Processing	Payment Service-API	HTTPS	RESTful API, OAuth
Driver Management	Driver Service-API	GCP Pub/Sub	RPC
Navigation	Notifaction Service-API	GCP Pub/Sub	RPC
Data Storage	Database Server	SQL/NoSQL over JDBC/ODCB	QSL, NoSQL

Motivation: In a microservices architecture, choosing the right communication protocol is crucial for performance and reliability. HTTPS is used for secure, stateless communication with external APIs, while Google Cloud Pub/Sub provides a highly scalable and reliable messaging service for inter-service communication, supporting asynchronous messaging patterns that are vital for decoupling services in microservices architectures. These technical interfaces are integral to the TripLink system, ensuring it operates securely, efficiently, and is resilient to changes and load variations.

Solution Strategy

Contents

Technology Decisions:

- Adoption of a microservices architecture to enable scalability, flexibility, and independent service development.
- Use of Google Cloud Pub/Sub for asynchronous inter-service communication and RESTful APIs for external integrations.

Top-level Decomposition:

- The system is decomposed into several key microservices, such as User Management, Ride Management, Payment Processing, Driver Management, and Navigation, each interacting through well-defined APIs.

Key Quality Goals Achievement:

- Strong emphasis on performance and reliability to handle high traffic and provide real-time updates.
- Focus on usability with an intuitive and responsive user interface.
- Rigorous security measures, including secure API gateways and data encryption, to protect user data.

Organizational Decisions:

- An agile development approach with cross-functional teams responsible for individual microservices.
- Continuous integration and deployment (CI/CD) practices to enable rapid and reliable software release cycles.

Motivation These decisions form the cornerstone of the TripLink architecture, ensuring it is robust, user-centric, and capable of adapting to changing market demands and user expectations. The chosen technology stack and development practices are aligned with the business goals of providing an efficient, secure, and highly available ride-sharing service. These foundational decisions guide subsequent architectural choices, ensuring consistency and alignment with the overall vision for the TripLink app.

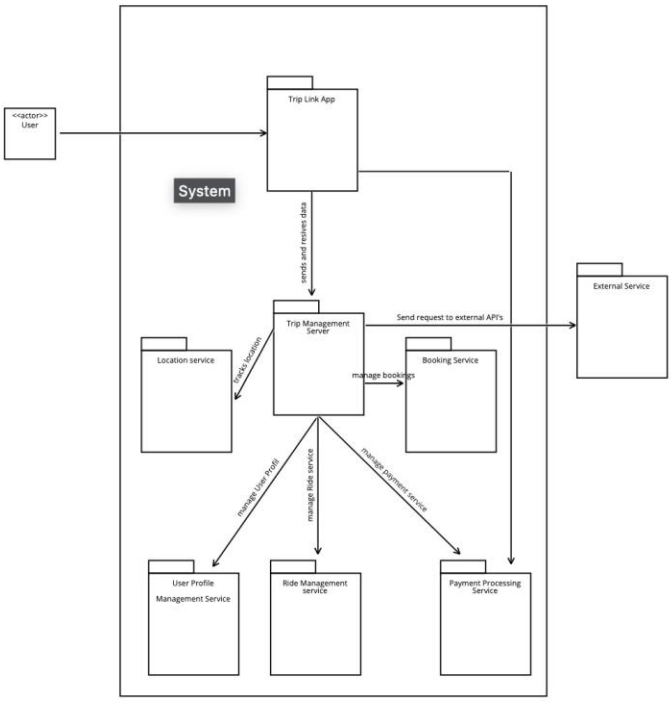
Building Block View

Content

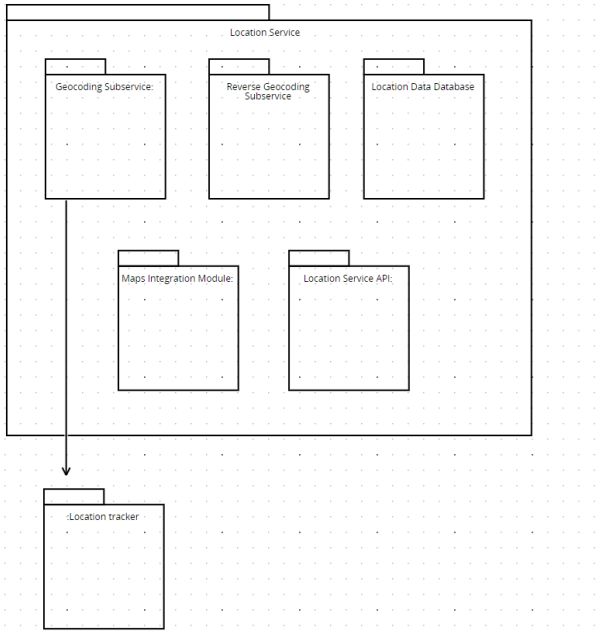
The TripLink system is designed as a comprehensive rideshare platform, architecturally segmented into distinct, interacting modules to provide seamless trip planning and management services to users

- **User**: This is the actor who interacts with the TripLink system. The User can be a traveler looking to plan a trip, book a ride, or access trip-related services.
- **TripLink App**: The primary interface for the User. It's the front-end through which Users initiate trip planning, ride booking, and other interactions with the system.
- **System**: Encompasses the entire back-end of the TripLink platform, which includes various services that handle different business logic and data processing tasks.
- **Trip Management Server**: Acts as the central processing unit within the system. It handles trip-related requests from the TripLink App, coordinates with other services, and communicates with external APIs.
- **Booking Service**: Manages all booking operations
- **Location Service**: Provides location-based functionality, accommodations, or providing navigational assistance.
- **User Profile Management Service**: Handles user data, including profiles, preferences, and security. This service ensures that user information is up-to-date and secure.
- **Ride Management Service**: Specific to managing the ride-booking aspect of the trip, including matching Users with transport options and handling ride lifecycle.
- **Payment Processing Service**: Responsible for handling financial transactions, including ride payments, booking fees, and other financial operations related to the trip.
- **External Service**: Represents third-party services and APIs that the TripLink system interacts with. These could include payment gateways, external booking systems, map services, and other data providers.

Level 1:



Level 2



Runtime View

Important Use Cases/Features and Execution:

- **Book a Ride**:
    - The user (Traveler) initiates a ride request using the TripLink app.
    - The app sends the request to the RideService.
    - RideService contacts the LocationService to find nearby drivers.
    - RideService sends ride requests to potential drivers until one accepts.
    - The chosen driver's details are sent back to the Traveler via the app.
- **Calculate Fees**:
    - Upon ride completion, the driver triggers the fare calculation through the app.
    - The app requests the RideService to calculate the fare.
    - RideService processes the fare and sends the amount to the app.
    - The Traveler confirms the payment, which is processed by the PaymentService.
    - PaymentService interfaces with an external Payment Provider to handle the transaction.
    - The app displays the receipt or an error message if the transaction fails.

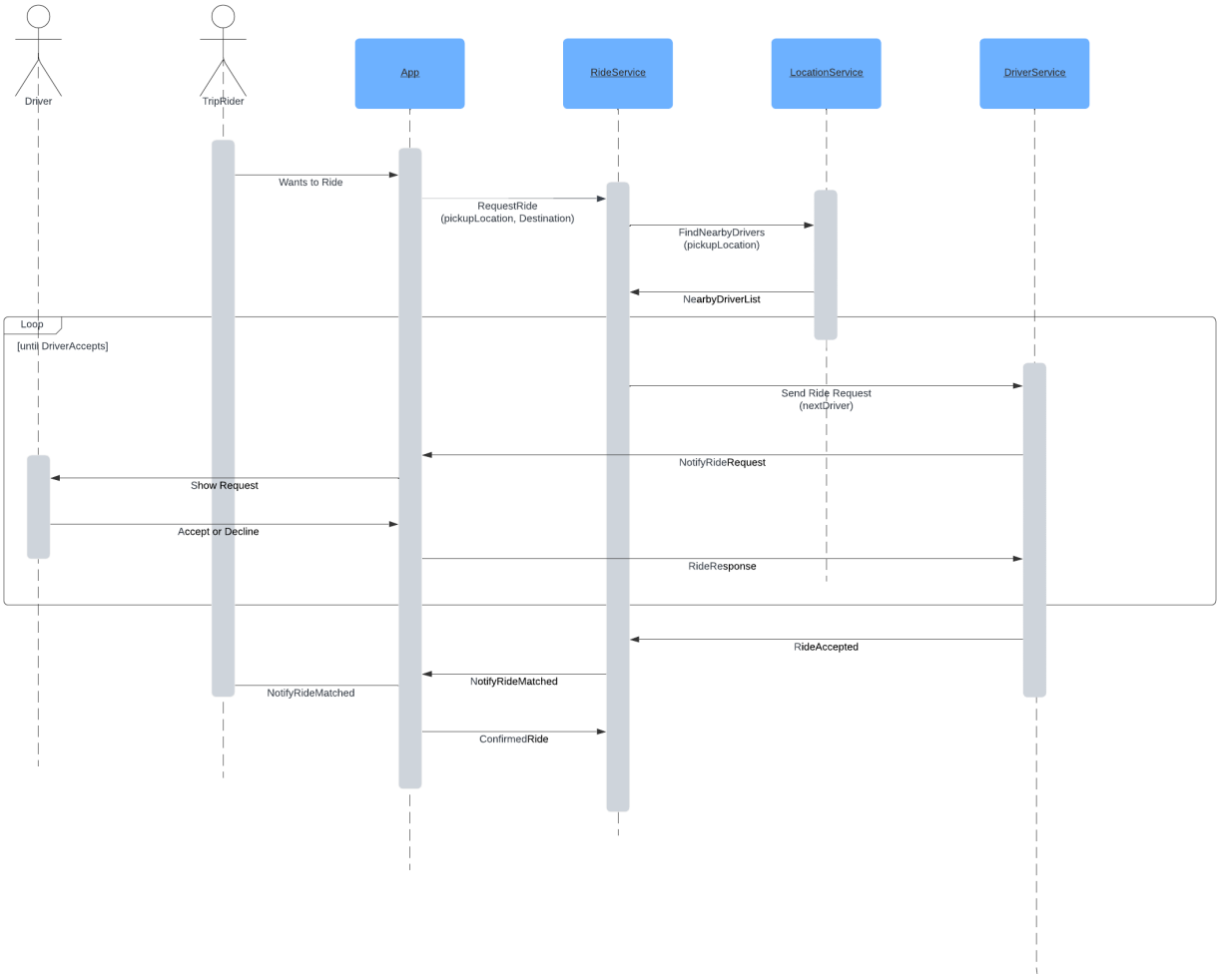
Error and Exception Scenarios:

- **Booking Errors**:
    - Handles scenarios where no drivers are available, the driver does not accept the ride, or there is a communication failure between the app and RideService.
- **Payment Errors**:
    - Manages payment processing failures due to issues like insufficient funds, external service downtime, or connectivity problems with the Payment Provider.

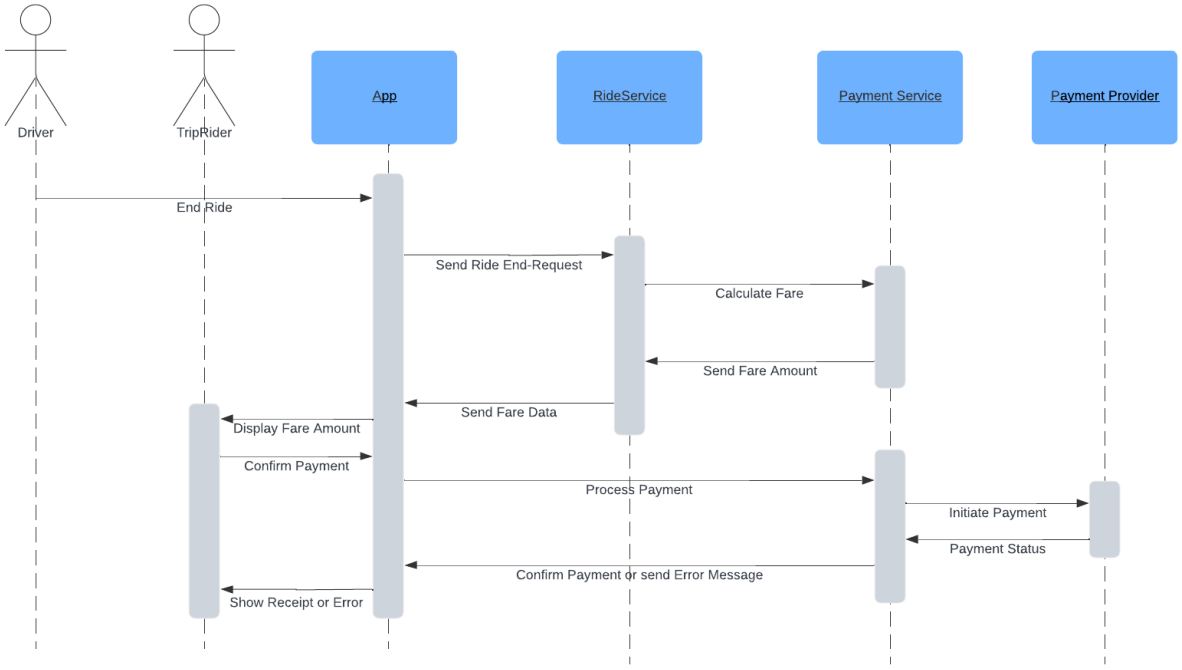
Motivation:

Understanding the detailed interactions of these critical features for the TripLink app is essential for several reasons:

Book a Ride


Driver
TripRider
App
RideService
LocationService
DriverService
Wants to Ride
RequestRide
(pickupLocation, Destination)
FindNearbyDrivers
(pickupLocation)
NearbyDriverList
Loop
[until DriverAccepts]
Send Ride Request
(nextDriver)
NotifyRideRequest
Show Request
Accept or Decline
RideResponse
RideAccepted
NotifyRideMatched
NotifyRideMatched
ConfirmedRide

Calculate Fees



Deployment View

Kubernetes Cluster Deployment for RideShare:

Infrastructure Elements:

- **Control Plane**: Manages the cluster, including the Scheduler, API Server, Controller Manager, and etc for cluster state.
- **Nodes**: Worker machines in the cluster that run the application workloads. Each node hosts multiple pods.
- **Pods**: The smallest deployable units created and managed by Kubernetes, each pod represents a running process in the cluster.

Mapping of Software Building Blocks:

- **Pod 1**: User Management Service - Manages user profiles, authentication, and registration.
- **Pod 2**: Ride Management Service - Handles ride requests, matching algorithms, and ride status.
- **Pod 3**: Driver Management Service - Manages driver profiles and availability.
- **Pod 4**: Payment Service - Processes payments and manages billing.
- **Pod 5**: Location Service - Tracks and updates the location of rides and drivers.
- **Pod 6**: Notification Service - Sends notifications and alerts to users.

Replicas: The table on the right side of the diagram indicates the number of replicas for each pod. This ensures high availability and load distribution. For example, the Ride Management Service has 3-4 replicas, likely due to its critical role in the system and the high demand for ride matchmaking.

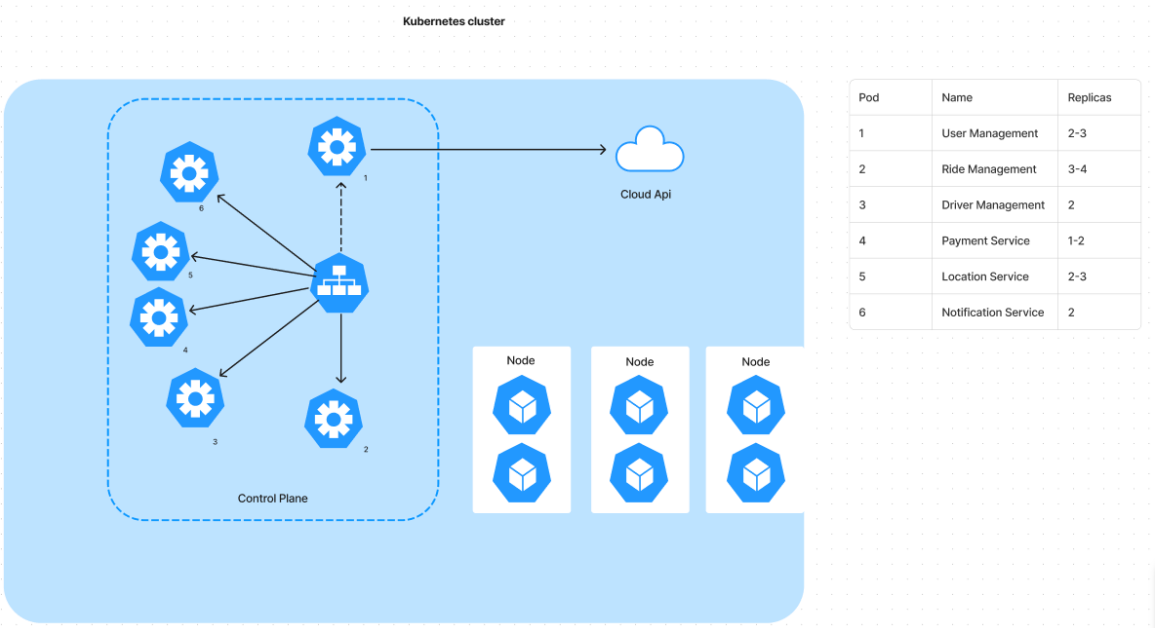
Channels:

- **Cloud API**: Represents the external interface for the cluster to interact with other cloud services, like payment gateways or external mapping services.

Environments: While not detailed in the diagram, the RideShare application would typically be deployed in several environments:

- **Development Environment**: For daily development work with frequent deployments and rapid changes.
- **Testing Environment**: To rigorously test new features and bug fixes in a controlled setting that simulates the production environment.
- **Production Environment**: The live environment where the application is available to end-users.

Motivation: Understanding the deployment of the RideShare application is crucial as the infrastructure affects performance, reliability, and scalability. The deployment view provides a clear mapping of the software components to the hardware that will execute them, ensuring that the technical team can manage and scale the system effectively. Additionally, it allows stakeholders to visualize and understand the complexity of the system, ensuring that the hardware and network setup can support the software requirements and traffic load



Pod	Name	Replicas
1	User Management	2-3
2	Ride Management	3-4
3	Driver Management	2
4	Payment Service	1-2
5	Location Service	2-3
6	Notification Service	2

Cross-cutting Concepts

Domain Concepts: The domain model of RideShare revolves around efficient urban mobility, connecting passengers with drivers. It encompasses user profile management, ride booking and tracking, payment processing, and driver-passenger interactions. This includes the core functionalities of searching for rides, managing ride details, handling payments, and facilitating driver and passenger ratings and feedback.

User Experience (UX) Concepts: RideShare prioritizes a user-centric design, focusing on ease of use, intuitive ride-booking processes, and clear navigation within the app. The design ensures seamless interaction from ride request to completion, with real-time updates for tracking and communication.

Security Concepts: Security within RideShare is paramount, encompassing secure authentication, encrypted data transmission, and rigorous driver verification processes. Privacy controls for user data and ride histories are integral, with additional measures like secure payment processing to protect financial information.

Architecture Patterns: RideShare adopts a microservices architecture, allowing for loosely coupled services that are independently deployable and scalable. This architecture is instrumental in facilitating rapid feature development, testing, and deployment, while also enhancing the system's resilience and scalability.

Operational Concepts: RideShare's cloud-based deployment strategy ensures scalability and high availability. The operational model includes continuous monitoring for system health, automated scaling to handle varying loads, and robust disaster recovery mechanisms to ensure uninterrupted service.

Development Concepts: The development approach for RideShare follows Agile methodologies, encouraging adaptability and fast-paced development cycles. Emphasis is placed on test-driven development (TDD) and a CI/CD pipeline, fostering high-quality code and enabling quick iterations and deployments.

Design Decisions

Contents

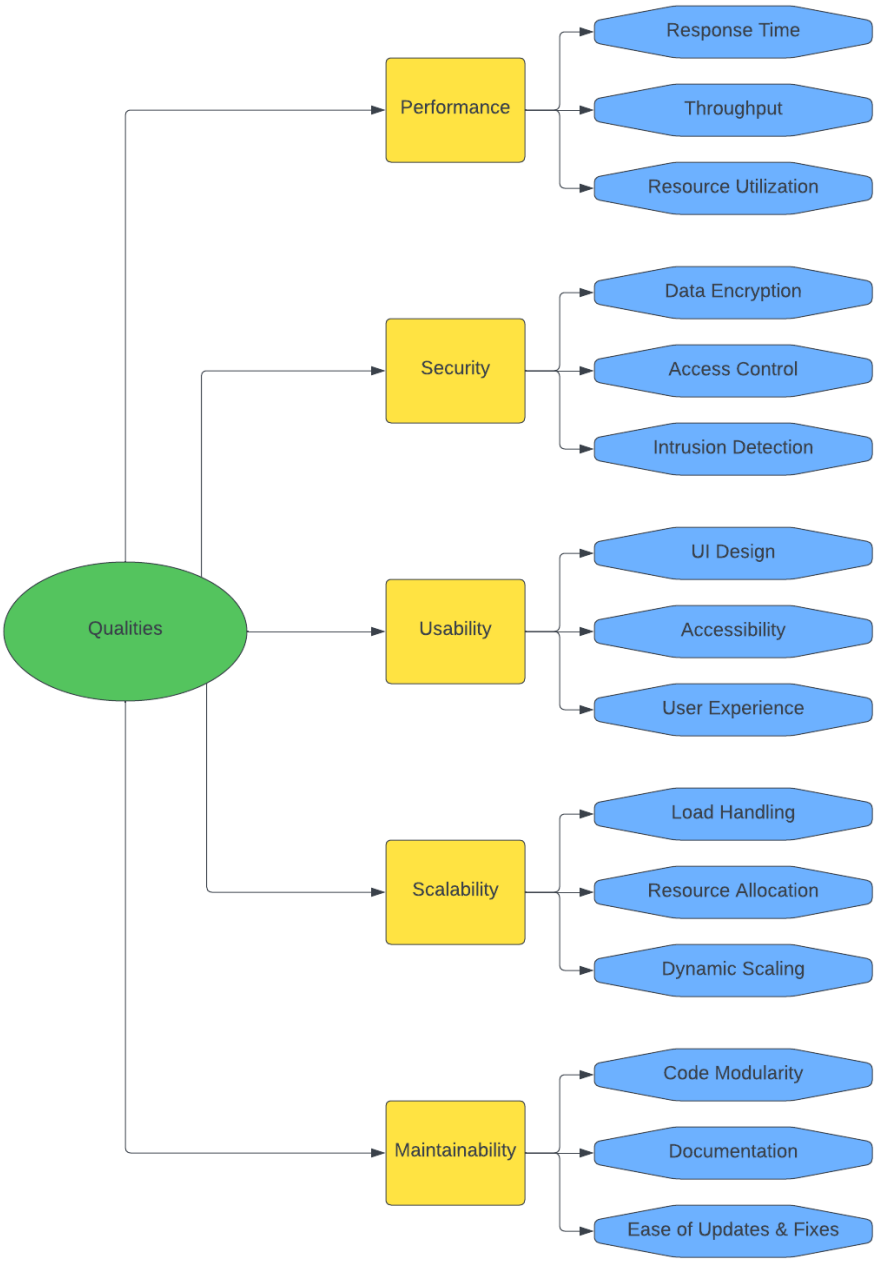
RideShare's architecture decisions include adopting a microservices architecture for flexible scaling, using cloud-based services for operational efficiency, and integrating advanced mapping and navigation tools for optimal route planning.

Problem	Considerer Alternatives	Decision
Managing Microservices Complexity	1. Continue with a simpler monolithic architecture. 2. Use a Service-Oriented Architecture (SOA).	Adopt a microservices architecture with robust service orchestration and monitoring tools to manage complexity effectively.
Real-Time Data Processing	1. Use traditional polling methods. 2. Implement a basic WebSocket solution.	Employ advanced WebSockets and event-driven architecture for real-time updates to ensure efficient and timely data processing.
Scalable Payment Processing	1. Develop an in-house payment solution. 2. Integrate with a single payment gateway.	Integrate with multiple payment gateways using a secure, scalable API to provide flexibility and reliability in payment processing.
Driver and Passenger Matching Efficiency	1. Basic first-come-first-serve algorithm. 2. Location-based matching without preferences.	Implement an intelligent matching algorithm considering location, driver availability, and user preferences to enhance efficiency.
Navigation and Route Optimization	1. Use basic GPS functionality. 2. Integrate with a single mapping service.	Integrate with multiple advanced mapping services for comprehensive route planning

		and optimization, improving ride efficiency.
Data Storage	Database Server	SQL/NoSQL over JDBC/ODCB

Quality Requirements

Quality Tree



Quality Scenarios

Quality Scenario	Description
Performance	• App loads within 2 seconds of launching • Ride requests processed within 15 seconds • Real time GPS tracking deviation within 100 meters
Reliability	• APP-Uptime of atleast 99,9% • Error-free and and secure payment processing • Cosistent in-app communication
Security	• User data is encrypted and protected from unauthorized access • Secure storage and processing of payment information • Driver and passenger verification for safety

Risks and Technical Debts

Risk	Description	Measures
Date Breach	Unauthorized access risk.	Encryption, audits, zero-trust model.
Microservice Interdependency	Potential issues in communication or data consistency across services.	Implement a robust service mesh, standardize inter-service communication protocols.
Infrastructure Overhead	As microservices scale, overhead might increase due to multiple instances.	Ensure efficient resource allocation, use orchestration tools like Kubernetes

Glossary

Centralized API Gateway

A unified platform that routes incoming API requests to the correct microservices, streamlining management and security protocols.

DevOps
Practices that integrate software development with IT operations to enhance agility and speed up the delivery lifecycle.

GCP Pub/Sub
Google Cloud's messaging and event ingestion service for asynchronous event-driven systems, enabling inter-service communication.

HTTPS
A secure communication protocol that encrypts data exchanged over the internet, protecting against interception and tampering.

Kubernetes
A container orchestration tool that automates the deployment, scaling, and management of containerized applications.

Load Balancing
The distribution of incoming network traffic across multiple servers to ensure even workload distribution and reliability.

Microservices
A design approach where an application is composed of small, independently deployable services, each focusing on single business functions.

Monolith Architecture
A single-tiered software architecture where the user interface and data access code are combined into a single program from a single platform.

OAuth
A protocol for authorization that allows third-party services to access user data without exposing user credentials.

Performance Bottlenecks
Points within a system that reduce throughput and efficiency, often identified for optimization to improve overall performance.

RESTful API
An API design that leverages HTTP protocols for creating, reading, updating, and deleting resources.

RPC (Remote Procedure Call)
A communication method that enables a computer program to execute a procedure on another machine without knowing the network details.