International Institute of Information Technology
Bangalore

Software Production Engineering and Cloud Computing
Report

CSE-816 and CSE-838

Project Report

Team Members:

Valmik Belgaonkar (IMT2022020)
Abhinav Deshpande (IMT2022580)
Vaibhav Bajoriya (IMT2022574)
Krish Dave (IMT2022043)
December 12, 2025

Software Production Engineering and Cloud Computing Project Report

Contents

The link to the GitHub repository is: Click here.

1 FreeRide: A Metro Car Pooling Application

1.1 Introduction

Metro Carpooling is a microservices-based ride-sharing platform designed to connect commuters traveling along similar routes in urban metropolitan areas. The system enables riders to find and share rides with drivers, optimizing transportation costs while reducing traffic congestion and carbon emissions.

Urban transportation faces significant challenges: increasing traffic congestion, rising fuel costs, and growing environmental concerns. Traditional ride-hailing services often result in inefficient single-occupancy vehicle trips. Metro Carpooling addresses these issues by facilitating shared rides among commuters with overlapping routes and schedules.

1.2 System Architecture Design Flow

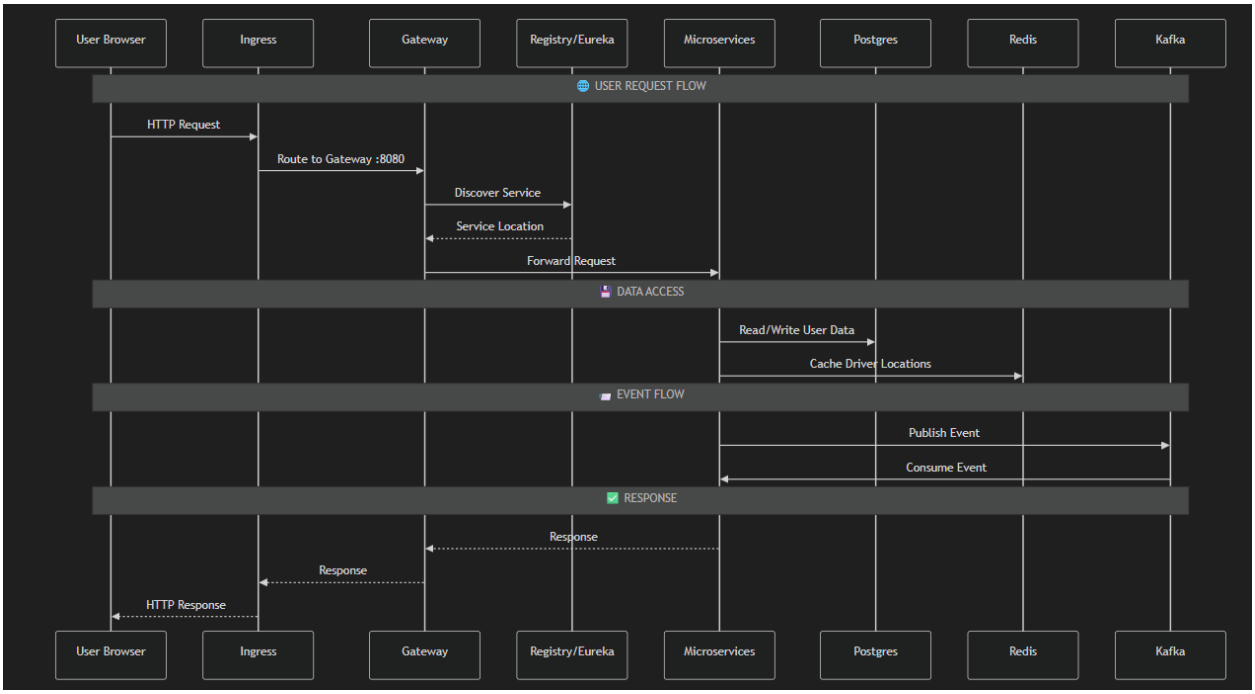


Figure 1: Metro Carpooling System Architecture Design

2 Overview of the design of the backend

The backend uses a API gateway that is central point of receiving the REST requests from the frontend. Then it uses gRPC to communicate with the backend microservices. The API gateway is the gRPC client and the microservices are the gRPC servers. We have used the expected Spring Boot layered architecture with the controller, service and repository layers. The API gateway has all the controllers and the all the microservices have the service and repository layers.

2.1 Reason:

The controller layer communicates with the frontend. The service layer handles the business logic. The repository layer communicates with the database. We can view a microservice architecture as a parallel monolithic architecture, and hence we must fit in the controller, service, and repository layers in our code base. Thus, in our design, as the API gateway communicates with the frontend, the controllers should be present in the API gateway. As microservices handle business logic and communicate with the DB, they must have service and repository layers.

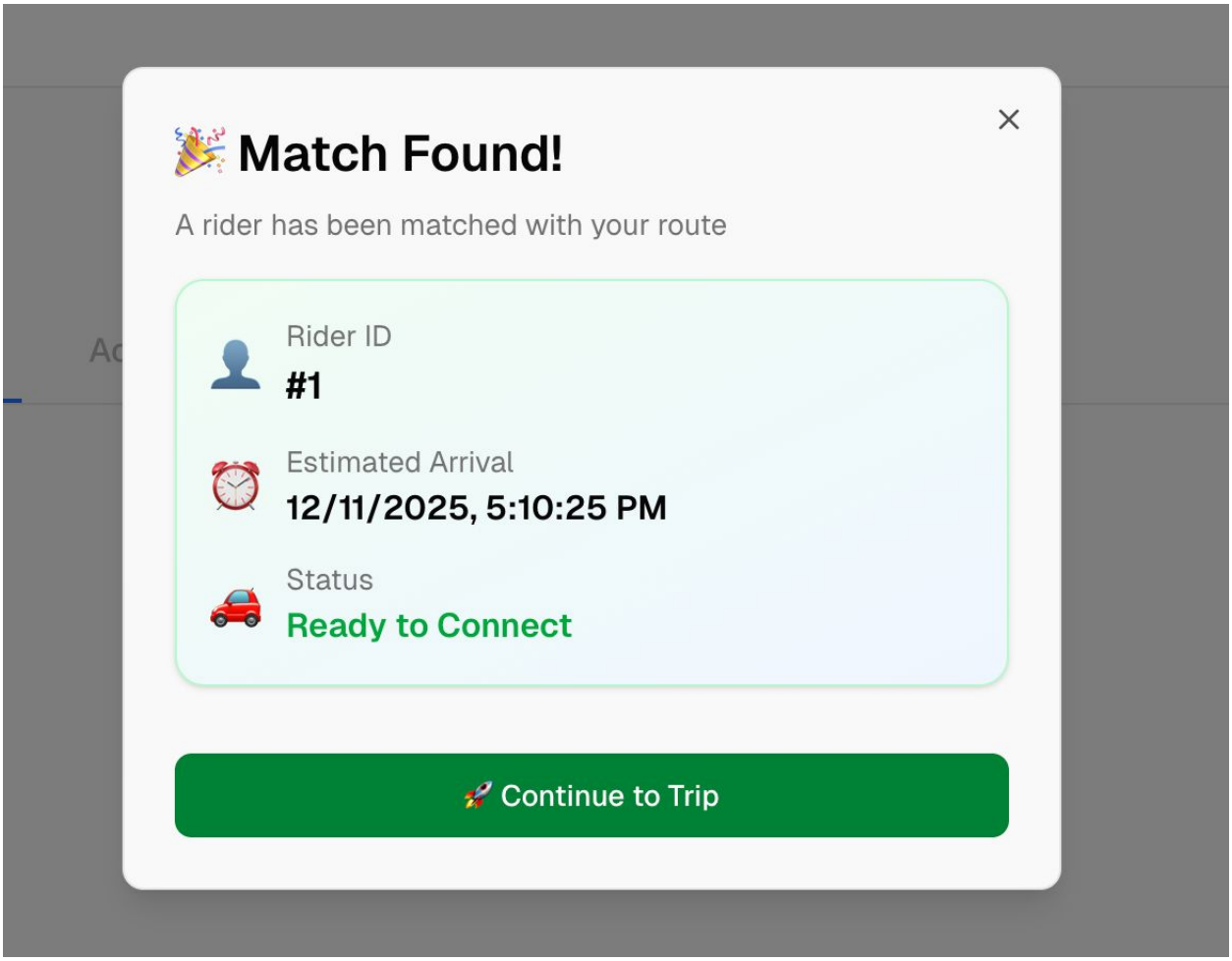


Figure 2: Screenshot of the rider driver match shown on the frontend

3 Matching Service: Matching logic (MatchingService.java)

3.1 Introduction

The `MatchingService` is a core component responsible for pairing riders with suitable drivers. It operates in an event-driven manner using Kafka for asynchronous communication and Redis for caching state (driver locations, rider queues, and distances). The service handles real-time driver updates, rider requests, and periodic checks for queued riders.

3.2 Data Structures and State Management

The service relies on several Redis-backed data structures to maintain state distributed across instances.

3.3 Driver Cache Model

The driver availability is stored in a nested Hash Map structure in Redis, identified by the key `driver-cache`.

- **Structure**: `HashMap<String, HashMap<String, List<MatchingDriverCache>>>`
- **Hierarchy**:

$$\text{Station} \rightarrow \text{Destination} \rightarrow \text{List of Drivers}$$

- **Purpose**: Allows efficient lookup of drivers currently at (or approaching) a specific station, grouped by their final destination.

3.3.1 Rider Waiting Queue

Riders who cannot be matched immediately are stored in a queue.

- **Key**: `rider-waiting-queue`
- **Type**: `Queue<RiderWaitingQueueCache>`
- **Behavior**: First-In-First-Out (FIFO) processing via a cron job.

3.4 Distance Matrix

A lookup map for distances between locations to enable spatial filtering.

- **Key**: `distance`
- **Structure**: `HashMap<String, HashMap<String, Integer>>`

3.5 Event Handling Architecture

3.5.1 Driver Location Updates

Trigger: `DriverLocationEvent` via Kafka topic `driver-location-topic`. **Logic**:

1. **De-duplication**: Checks if the message ID has already been processed using a Redis key with a 24-hour TTL.
2. **Locking**: Acquires a distributed lock (`lock:drivers`) to ensure thread safety during cache updates.
3. **Remove Old State**: If the driver was previously at an `oldStation` (and had a `finalDestination`), they are removed from the specific list in the `allMatchingCache`.
4. **Update New State**: The driver is added to the `nextStation` map under their `finalDestination`. The entry includes vital metrics like `timeToReachStation` and `availableSeats`.

3.5.2 Rider Request Processing

Trigger: `RiderRequestDriverEvent` via Kafka topic `rider-requests`. **Logic**:

1. **Locking**: Acquires locks for both drivers and distances.
2. **Matching Algorithm**: Executes the matching logic (detailed in Section 3.6).
3. **Outcome**:
    - **match**: Publishes `DriverRiderMatchEvent` and removes the driver from the cache.
    - **no-match**: Adds the rider to the Redis waiting queue.

3.6 Matching Algorithm

The matching process selects the best driver for a rider based on spatial and temporal proximity.

3.6.1 Parameters

- D_{thresh}: Distance Threshold (5 units).
- T_{thresh}: Time Threshold (10 minutes = 600,000 ms).

3.6.2 Algorithm Steps

Algorithm 1 Driver-Rider Matching Logic

Input: Rider R at Station S_{pickup} going to D_{rider} with Arrival Time T_{rider}
Output: Matched Driver or Enqueue Rider
$PQ \leftarrow$ PriorityQueue (ordered by $timeToReachStation$, ascending)
$StationMap \leftarrow$ Cache.get(S_{pickup})
if $StationMap$ is empty **then**
    return Enqueue(R)
end if
for all DriverDestination D_{driver} in $StationMap$.keys **do**
    $dist \leftarrow$ Distance(D_{rider}, D_{driver})
                                                                ▷ Spatial Filter
    if $dist \leq D_{thresh}$ **then**
        $Drivers \leftarrow StationMap$.get($D_{driver}$)
        for all Driver d in $Drivers$ **do**
            $T_{driver\_arrival} \leftarrow CurrentTime + d.timeToReachStation$
            $\Delta t \leftarrow |T_{rider} - T_{driver\_arrival}|$
                                                                ▷ Temporal Filter
            if $\Delta t \leq T_{thresh}$ **then**
                PQ.add(d)
            end if
        end for
    end if
end for
if PQ is not empty **then**
    $d_{best} \leftarrow PQ$.poll()   ▷ Best driver (min time to reach)
    Remove d_{best} from Cache
    Publish MatchEvent(R, d_{best})
else
    Enqueue(R)
    Update Redis Waiting Queue
end if

3.7 Cron Job Processing

A scheduled task runs every second (`cron = "* * * * * *"`) to retry matching for waiting riders.
Procedure:

1. Acquires distributed locks for Drivers, Distances, and the Waiting Queue.
2. Polls the head of the `rider-waiting-queue`.
3. Executes the same logic as the standard Matching Algorithm (Section 3.6).
4. **Result Handling**:
    - If a match is found: The match event is published, and the driver is removed from availability.
    - If no match is found: The rider's arrival time is updated to `System.currentTimeMillis()` (resetting their "freshness" for time delta calculations) and they are pushed back to the end of the queue.

3.7.1 Concurrency Control

To handle race conditions in a distributed environment (multiple service instances), `RedisDistributedLock` is utilized.

- **Retry Policy**: Attempts to acquire a lock up to 10 times with a 200 ms back-off.
- **Key Locks**:
    - `lock:drivers`: Protects the driver availability cache.
    - `lock:distance`: Protects reads/writes to the distance matrix.
    - `lock:rider-waiting-queue`: Protects the FIFO queue during cron processing.

4 Driver Service: Driver Location Simulation Logic (DriverService.java)

4.1 Introduction

The `DriverService` mimics real-time driver movement along a predefined route. Unlike a real GPS system, it simulates movement in discrete time steps (ticks) triggered by a scheduled Cron job. It manages driver state updates, route traversal, and event emission (location updates, ride completion) via Kafka.

4.2 Simulation Model

The simulation proceeds in discrete ticks.

4.2.1 Constants and Parameters

- D_{tick}: Distance moved per tick (10.0 units).
- T_{tick}: Duration of a tick (120 seconds or 2 minutes).
- v_{sim}: Simulated speed derived as:

$$v_{sim} = \frac{D_{tick}}{T_{tick}} = \frac{10.0}{120} \approx 0.0833 \text{ units/sec}$$

4.3 Movement Logic

During each cron tick, every active driver's position is updated.

4.4 State Representation

A driver's state is encapsulated in the `DriverCache` object:

- `NextPlace`: The immediate next node in the route graph.
- `DistToNext`: Remaining distance to reach `NextPlace`.
- `Route`: List of nodes $[N_0, N_1, ..., N_k]$.

4.4.1 Distance Update Equation

Let $d_{remain}(t)$ be the distance to the next node at time t. At time $t+1$ (next tick):

$$d_{remain}(t+1) = d_{remain}(t) - D_{tick}$$

4.4.2 Node Crossing Logic

If $d_{remain}(t+1) \leq 0$, the driver has reached or passed the node. The "overshoot" distance is:

$$\Delta_{excess} = |d_{remain}(t+1)|$$

The algorithm then advances the next target node iteratively until Δ_{excess} is consumed by upcoming segment lengths. Let S_i be the segment distance between node N_i and N_{i+1}.

1. While $\Delta_{excess} \geq S_{current}$:
   - $\Delta_{excess} \leftarrow \Delta_{excess} - S_{current}$
   - Advance current node pointer ($i \leftarrow i+1$)
   - If $N_{current}$ is the final destination, end ride.
2. The new specific location is effectively at distance $S_{new} - \Delta_{excess}$ from the new previous node towards the new next node.
3. **Update State**:

$$\texttt{DistToNext} \leftarrow S_{new\_segment} - \Delta_{excess}$$

4.5 Time Projection Math

The service calculates the "Expected Time of Arrival" (ETA) to the next node or a specific metro station.

4.5.1 Duration Calculation

Given a generic distance d, the time t_{sec} required to traverse it is calculated as:

$$t_{sec} = \lceil \frac{d}{D_{tick}} \times T_{tick} \rceil$$

Substituting constants:

$$t_{sec} = \lceil \frac{d}{10.0} \times 120 \rceil = \lceil 12 \times d \rceil \text{ seconds}$$

4.6 Station Search and ETA

To inform the rider, the system computes the time to the **Next Metro Station**.

4.6.1 Algorithm

1. **Find Target Node**: Scan the route forward from the current position to find the first node $N_{station}$ that maps to a valid Metro Station ID in the `nearby-stations` map.
2. **Calculate Total Distance**: Let the driver be between N_{k-1} and N_k, with distance d_k remaining to reach N_k. Let the target station be at node N_m (where $m \geq k$).

$$D_{total} = d_k + \sum_{j=k}^{m-1} \text{Distance}(N_j, N_{j+1})$$

3. **Calculate Time**: Apply the duration formula:

$$ETA_{station} = \text{computeDurationFromDistance}(D_{total})$$

4.7 Event Handling

4.7.1 Location Updates

If a valid next station is found and available seats > 0, a `DriverLocationEvent` is emitted via Kafka.

- **Topic**: `driver-location-topic`
- **Payload**: Driver ID, Old Station, Next Station, ETA, Available Seats.

4.8 Ride Completion

If the driver reaches the final destination node defined in the route:

1. A `DriverRideCompletionEvent` is published.
2. The driver is evicted from the Redis cache.

4.9 Concurrency and Data Integrity

The simulation ensures thread safety and data consistency using distributed locks.

- **Lock**: `lock:drivers` is acquired before any read/write to the driver cache.
- **Cache Normalization**: Since Redis data can be deserialized as generic Maps or specific Objects depending on the writer, a normalization step (`normalizeDriverCache`) ensures consistent `DriverCache` java, objects.
- **Safe Geometry Reads**: `safeReadLocationMap` robustly handles numeric types (Integer vs Double) from JSON deserialization to prevent `ClassCastException`.

5 Notification Service

5.1 Introduction

The `NotificationService` acts as a real-time bridge between the asynchronous backend Event Bus (Kafka) and the synchronous or streaming frontend clients (gRPC/Gateway). Its primary purpose is to ingest events from Kafka topics, deduplicate them, and push them to connected clients via Reactive Streams.

5.2 Architectural Design

5.2.1 Reactive Streams with Sinks

The core of the service design uses Project Reactor's `Sinks`.

```
private final Sinks.Many<RiderDriverMatch> riderDriverSink =
    Sinks.many().replay().latest();
```

- **Type**: `Sinks.Many` allows broadcasting multiple events to multiple subscribers (Multicast).
- **Replay Strategy**: `.replay().latest()` ensures that a new subscriber immediately receives the **last emitted item**. This is crucial for handling reconnections or clients connecting slightly after an event occurred.
- **Purpose**: It decouples the Kafka Consumer thread from the gRPC Server thread. Kafka pushes to the Sink; gRPC streams from the Sink.

5.3 Event Processing Logic

The service listens to multiple Kafka topics. The pattern for each listener is identical:

5.3.1 1. Idempotency Check

Before processing, the service checks if the message has already been handled to prevent double notifications (e.g., due to Kafka rebalancing).

$$\text{Processed}(M_{id}) = \exists Key(\texttt{PREFIX} + M_{id}) \text{ in Redis} \tag{1}$$

If `Processed` is true, the message is acknowledged and skipped.

5.3.2 2. Data Transformation

The incoming byte array (Kafka message) is parsed into a Protobuf Event object (e.g., `DriverRiderMatchEvent`). It is then converted into a notification-specific Protobuf message (e.g., `RiderDriverMatch`) suitable for the frontend.

5.4 3. Emission

The event is emitted to the Sink:

```
riderDriverSink.tryEmitNext(match);
```

This makes the event available to all active gRPC streams.

5.5 4. Acknowledgment

Finally, the message ID is marked as processed in Redis (with a 24-hour TTL), and the Kafka offset is manually acknowledged.

5.6 Streaming API

The service exposes methods that return `Flux<T>` for gRPC integration.

```
public Flux<RiderDriverMatch> streamRiderDriverMatches() {
    return riderDriverSink.asFlux();
}
```

Unlike a standard REST API that returns a list, this returns a **continuous stream** of data. The gRPC server calls this method and subscribes to the Flux, keeping the connection open and pushing new events to the client as they arrive in real-time.

5.7 Summary of Channels

The service manages four distinct notification channels:

1. **Rider-Driver Match**: Notifies both parties when a match is found.
2. **Driver Ride Completion**: Notifies the driver (and system) that they reached the final destination.
3. **Rider Ride Completion**: Notifies the rider that they have been dropped off.
4. **Driver Location for Rider**: Provides real-time ETA updates to the rider about their approaching driver.

6 Rider Service

6.1 Overview

The `RiderService` is a core component of the **Rider Microservice** responsible for handling direct rider interactions. Its primary function is to ingest rider requests (such as booking a ride) and publish them to the event-driven architecture via Kafka, acting as an event producer.

6.2 Design Implementation

File: `rider/src/main/java/com/metrocarpool/rider/service/RiderService.java`

6.2.1 Dependencies

The service leverages the following key components:

- `KafkaTemplate`: Spring Boot's abstraction for sending messages to Kafka.
- `Protobuf`: The `RiderRequestDriverEvent` class is a generated Protobuf class from the `contracts` shared library, ensuring type-safe event definitions.

6.3 Logic Workflow

6.3.1 Method: `processRiderInfo`

This method transforms a synchronous service call (typically from a REST Controller or gRPC Server) into an asynchronous event.

6.3.2 1. Payload Construction

It constructs a `RiderRequestDriverEvent` utilizing the Builder pattern.

```
RiderRequestDriverEvent event = RiderRequestDriverEvent.newBuilder()
    .setMessageId(UUID.randomUUID().toString()) // Unique Event ID
    .setRiderId(riderId)
    .setPickUpStation(pickUpStation)
    .setArrivalTime(arrivalTime)
    .setDestinationPlace(destinationPlace)
    .build();
```

Purpose: To encapsulate all necessary routing information (User ID, Location, Time) into a standardized binary format.

6.3.3 2. Asynchronous Publication

The event is published to the configured Kafka topic (e.g., `rider-requests`).

```
CompletableFuture<SendResult<...>> future =
    kafkaTemplate.send(topic, key, value);
```

- **Key**: `riderId.toString()` is used as the partition key. This guarantees that all requests from the same rider land on the same Kafka partition, preserving order (though order is less critical for single bookings).
- **Value**: The byte array representation of the Protobuf object.

6.3.4 3. Error Handling

The publication is non-blocking but resilient:

- **Success**: Logs the metadata (topic, partition) for debugging.
- **Failure**: Logs the error. Ideally, this would trigger a fallback (e.g., storing in a database or Redis Dead Letter Queue) to ensure the user's request isn't lost during network blips.

6.4 Purpose

The purpose of this service is to **decouple** the Rider Service from the Matching Service.

- The Rider Service does not call the Matching Service directly (HTTP/gRPC).
- Instead, it "fires and forgets" the request into the Event Bus.
- This ensures high availability: if the Matching Service is down, the Rider Service can still accept requests (buffered in Kafka).

7 Trip Service

7.1 Overview

The `TripService` is responsible for managing the lifecycle of an active ride. It bridges the gap between the Match, Driver, and Notification services by maintaining the state of "Who is in whose car?" and orchestrating real-time updates.

7.2 Core Responsibilities

1. **State Management**: Tracks which riders are currently in a specific driver's vehicle using Redis.
2. **Trip Association**: Reacts to match events to associate riders with drivers.
3. **Real-Time Updates**: Forwards driver location updates (e.g., "5 mins to station") to the specific riders in that driver's car.
4. **Trip Completion**: Handles the end-of-trip logic, notifying all parties and cleaning up the cache.

7.3 Data Structure Design

7.3.1 Redis Cache Schema

The service maintains a mapping of **Driver ID** to a **List of Riders**.

- **Key**: `trip-cache`
- **Value**: `Map<String, List<TripCache>>`
- **Mapping**:

$$\text{DriverID} \rightarrow [\text{Rider } 1, \text{Rider } 2, \dots]$$

This structure allows $O(1)$ lookup to find all riders affected by a specific driver's event.

7.4 Event Logic

7.4.1 1. Trip Start (Match Found)

Trigger: `rider-driver-match` Kafka topic. **Logic**:

1. A distributed lock (`lock:trip`) is acquired.
2. The event payload contains `DriverID` and `RiderID`.
3. The service fetches the current trip cache from Redis.
4. It adds the `RiderID` to the list associated with the `DriverID`.
5. The updated map is persisted back to Redis.

7.4.2 2. In-Trip Updates (Driver Location)

Trigger: `driver-location-topic` (from Driver Service). **Logic**:

1. Driver moves (simulated or real).
2. Trip Service looks up all riders associated with this driver.
3. For each rider, it emits a specific `DriverLocationForRiderEvent` to the `driver-location-rider` topic.
4. The Notification Service picks this up and pushes it to the specific rider's UI.

This ensures that Rider A only gets updates about Driver X, not Driver Y.

7.4.3 3. Trip Completion

Trigger: `ride-completion-topic` (from Driver Service). **Logic**:

1. Detects that the driver has reached the final destination.
2. Retrieves the list of all riders in the car.
3. Emits a "Ride Completed" event for the Driver.
4. Emits individual "Ride Completed" events for every Rider.
5. **Cleanup**: Removes the `DriverID` entry from the Redis cache, effectively ending the session.

7.5 Concurrency Control

To prevent race conditions (e.g., a trip ending while a new rider is being added), the service uses a **Redis Distributed Lock** (`RedisDistributedLock`).

- **Acquire**: Uses `SETNX` with a 5000ms timeout.
- **Retry**: Retries up to 10 times with 200ms backoff.
- **Release**: Uses a Lua script to safely release the lock only if owned by the current thread.

8 Notification Service: Pipeline followed for real time notifications

8.1 Introduction

The Metro Car Pooling project implements a real-time notification pipeline that bridges backend microservices with the frontend using a combination of **Kafka**, **gRPC Server Streaming**, and **Server-Sent Events (SSE)**.

8.2 Pipeline Architecture

The pipeline allows asynchronous events (e.g., Rider Matched, Ride Completed) to be pushed to the user's browser in real-time.

1. **Event Producers**: Services like `MatchingService` or `TripService` generate events.
2. **Message Broker**: **Apache Kafka** acts as the decoupling layer.
3. **Notification Hub**: The `NotificationService` consumes Kafka events and buffers them.
4. **Streaming Layer**: gRPC streams data from the Notification Service to the API Gateway.
5. **Client Edge**: The API Gateway exposes SSE endpoints to the Frontend.

8.3 Detailed Component Flow

8.3.1 1. Event Production (Matching/Trip Services)

When business logic executes (e.g., a driver is matched), the service publishes an event to a Kafka topic.

- **Source**: `MatchingService.java`
- **Topic**: `rider-driver-match`
- **Payload**: Protocol Buffers serialized message (e.g., `DriverRiderMatchEvent`).

8.3.2 2. Consumption and "Cold SSE" Logic (Notification Service)

The `NotificationService` consumes these events and prepares them for streaming.

- **Listener**: `@KafkaListener` receives the byte array and parses it.
- **Buffering (The "Cold" Logic)**: The service uses Project Reactor's `Sinks` to multicast events.
  - **Implementation**: `Sinks.many().replay().latest()`
  - **Behavior**: This configures the stream as a "Cold" (or Replaying Hot) source. When a new client connects, it **immediately replays the latest buffered event**. This ensures that if a user reconnects or connects slightly after an event, they still receive the current state (e.g., "Match Found") instantly without waiting for a new event.

8.3.3 3. gRPC Streaming (Notification → Gateway)

The service exposes a **Server Streaming RPC** defined in `trip_to_notification.proto`.

- **Server**: `NotificationGrpcServer.java`
- **Method**: `matchNotificationInitiationPost`
- **Logic**: Subscribes to the internal `Sink` Flux and pushes items to the `StreamObserver` as they arrive.

8.3.4 4. Server-Sent Events (Gateway → Frontend)

The API Gateway acts as the bridge to the web client.

- **Controller**: `NotificationController.java`
- **Endpoint**: `GET /api/notification/matches`
- **Protocol**: `text/event-stream` (SSE)
- **Process**:
  1. Calls the `NotificationGrpcClient` to start the gRPC stream.
  2. Wraps the response in a `Flux<ServerSentEvent<String>>`.
  3. Serializes the Protobuf objects to JSON for the browser.

8.4 Connection Lifecycle and Termination

8.4.1 Infinite Streaming

The pipeline is designed for **infinite streaming**. There is no logic in the `NotificationController` or `NotificationGrpcServer` to close the connection "after some time" (e.g., no `timeout` or `take(N)` operators are used). The stream remains open to push real-time updates indefinitely.

8.4.2 Cancellation propagation

The connections are closed based on client behavior (Backpressure/Cancellation):

1. **Browser Disconnect**: If the user closes the tab, the SSE connection drops.
2. **Gateway Detection**: The Spring WebFlux `NotificationController` detects the cancellation signal.
3. **Upstream Cancellation**: The Gateway cancels the subscription to the gRPC client.
4. **gRPC Cancellation**: The gRPC channel propagates the cancellation to the `NotificationService`.
5. **Resource Cleanup**: The `NotificationGrpcServer` stops sending updates to that specific `responseObserver`.

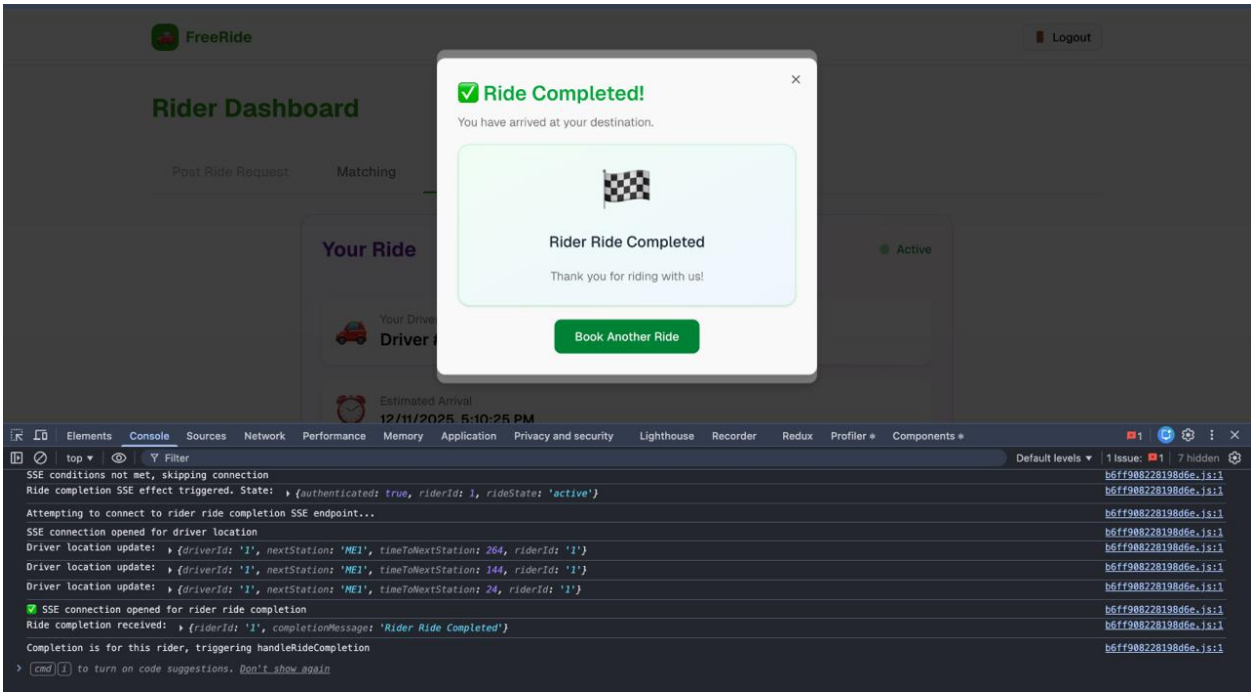


Figure 3: Notification of the completion of the ride of the rider

9 User Service

9.1 Overview

The `User Service` is the foundational Identity and Access Management (IAM) microservice in the Metro Car Pooling ecosystem. It acts as the single source of truth for all user entities, managing the lifecycle and authentication of the system's two primary actors: **Drivers** and **Riders**.

9.2 Architectural Design

The service follows a robust **Layered Architecture** typical of Spring Boot applications, designed for separation of concerns and testability.

9.2.1 Layers

1. **Controller / gRPC Layer**: (External Interface) Handles incoming network requests, deserializes parameters, and delegates to the Service Layer.

2. **Service Layer** (`UserService.java`): Encapsulates the core business logic. It allows for transaction management and validation before interacting with the database.

3. **Repository Layer** (`DriverRepository`, `RiderRepository`): Interfaces with the persistent storage (Database) using Spring Data JPA. It abstracts the SQL queries.

4. **Entity Layer** (`DriverEntity`, `RiderEntity`): Java POJOs mapped directly to database tables (ORM).

9.3 Logic Implementation

File: `user/src/main/java/com/metrocarpool/user/service/UserService.java`

9.3.1 1. User Registration (Sign Up)

The service provides distinct pathways for registering Drivers and Riders, accommodating their specific data requirements (e.g., License IDs for drivers). **Logic Flow:**

1. **Input**: Receives raw data (Username, Password, License ID).
2. **Construction**: Uses the `Builder Pattern` (via Project Lombok) to create a new Entity instance. This code is clean and readable.
3. **Persistence**: Calls `repository.save(entity)` to insert the record into the database.
4. **Output**: Returns the persisted entity, now populated with a generated Database ID (Primary Key).

```
public DriverEntity driverSignUp(String username, String password, Long licenseId){
        try {
            log.info("Reached UserService.driverSignUp.");

            DriverEntity driverEntity = driverRepository.save(
                    DriverEntity.builder()
                    .username(username)
                    .password(password)
                    .licenseId(licenseId)
                    .build()
            );
            return driverEntity;
        }
        catch (Exception e){
            log.error("Error in UserService.driverSignUp = {}", e.getMessage());
            return null;
        }
    }
```

```
public RiderEntity riderSignUp(String username, String password){
        try {
            log.info("Reached UserService.riderSignUp.");

            RiderEntity riderEntity = riderRepository.save(
                    RiderEntity.builder()
                            .username(username)
                            .password(password)
                            .build()
            );
            return riderEntity;
        }
        catch (Exception e){
            log.error("Error in UserService.riderSignUp = {}", e.getMessage());
            return null;
        }
    }
```

9.3.2 2. Authentication (Login)

The service provides a mechanism to verify credentials. **Logic Flow:**

1. **Retrieval**: Fetches the user record by Username.
2. **Existence Check**: If `repository.findByUsername` returns empty, the user does not exist (Return `null`).
3. **Credential Verification**: Compares the provided plain-text password with the stored password.
4. **Result**: Returns the User Entity on success (Login Successful) or `null` on failure (Invalid Credentials).

```
public DriverEntity driverLogin(String username, String password) {
    DriverEntity driver = driverRepository.findByUsername(username).orElse(null);
    if (driver != null && password.equals(driver.getPassword())) {
        return driver;
    }
    return null;
}
```

9.4 Purpose and Impact

- **Identity Provider**: It generates the `driverId` and `riderId` that uniquely identify users across all other services (Matching, Trip, Notification).
- **Security Boundary**: Acts as the gatekeeper, ensuring only registered users can access the system's core features.
- **Flexibility**: By separating Drivers and Riders into different entities and repositories, the system allows for independent evolution of their profiles (e.g., adding "Car Model" to Drivers without affecting Riders).

10 API Gateway

10.1 Overview

The **API Gateway** serves as the single entry point for all external client traffic (Web/Mobile Apps). It implements the **BFF (Backend for Frontend)** pattern, abstracting the complexity of the underlying microservices environment. Instead of clients speaking directly to Driver, Rider, or User microservices, they communicate with this unified REST interface.

10.2 Core Responsibilities

1. **Protocol Translation**: Converts external HTTP/REST requests into internal gRPC calls.
2. **Service Discovery**: Dynamically resolves the locations of backend services using Eureka.
3. **Security**: Handles JWT generation and user authentication.
4. **Real-Time Streaming**: Bridges gRPC Server-Side Streaming to HTTP Server-Sent Events (SSE).

10.3 Design Implementation

10.3.1 1. Communication Stack

The gateway uses a hybrid communication model:

- **Frontend ↔ Gateway**: JSON over HTTP (REST) and Server-Sent Events (SSE).
- **Gateway ↔ Microservices**: Protobuf over gRPC (Netty).

10.3.2 2. Workflow Logic

Example: User Signup (`UserController.java`)

1. **Receive**: `@PostMapping("/add-driver")` accepts a JSON DTO.
2. **Discover**: Queries Eureka for the active instance of the `USER-SERVICE`.
3. **Connect**: Establishes a gRPC channel to the resolved IP and Port.
4. **Delegate**: calls `stub.driverSignUpRequest()` using the auto-generated BlockingStub.
5. **Response**: Maps the Protobuf response back to a JSON DTO and returns standard HTTP 200/400 codes.

10.3.3 3. Real-Time Notification Logic (`NotificationController.java`)

This is a critical flow for updating the frontend about live events (Matching, Location Updates).

- **Endpoint**: `/matches`, `/driver-location-for-rider`.
- **Technology**: Uses **Project Reactor (Flux)** to handle reactive streams.
- **Mechanism**:
  - The frontend subscribes to an SSE endpoint.
  - The Gateway invokes a gRPC streaming method (`stub.matchNotification()`).
  - It wraps the gRPC `StreamObserver` in a Reactor `Flux.create()` sink.
  - Messages flow **Asynchronously**: Backend $\xrightarrow{gRPC}$ Gateway $\xrightarrow{SSE}$ Browser.

10.4 Key Components

10.4.1 gRPC Clients

The gateway manually constructs gRPC channels based on service discovery data.

```
public class UserGrpcClient {
    public SignUpResponse signUp(DTO request) {
        // 1. Discovery
        ServiceInstance instance = discoveryClient.getInstances("user")
            .findFirst().orElseThrow();

        // 2. Channel Creation
        ManagedChannel channel = ManagedChannelBuilder
            .forAddress(instance.getHost(), getGrpcPort(instance))
            .usePlaintext().build();

        // 3. Invocation
        var stub = UserServiceGrpc.newBlockingStub(channel);
        return stub.driverSignUpRequest(protoRequest);
    }
}
```

10.4.2 Benefits of this Design

- **Security**: Internal microservices are hidden behind a private network; only the Gateway is exposed.
- **Performance**: Internal chatter uses efficient binary Protobufs.
- **Simplicity**: Clients deal with standard REST/JSON, unaware of the complex gRPC implementation details.

11 Registry Service: Netflix Eureka Service Registry

11.1 Overview

The **Registry** component (located in `registry/`) is a dedicated microservice acting as the **Service Discovery Server**. It uses **Netflix Eureka**, a REST-based service primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. In the Metro Car Pooling architecture, it serves as the central phonebook where all other microservices (User, Rider, Driver, Notification, Gateway) register themselves so they can find each other without hardcoded hostnames or ports.

11.2 Implementation Logic

11.2.1 Main Application Wrapper

File: `RegistryApplication.java` The Application class is minimal but critical. It uses the `@EnableEurekaServer` annotation to stand up a registry that other applications can talk to.

- `@SpringBootApplication`: Standard Spring Boot entry point.
- `@EnableEurekaServer`: Activates the Eureka Server implementation. This marker annotation configures the registry business logic, including the peer-to-peer replication nodes and the dashboard UI.

11.3 Configuration and Design

File: `application.yaml` The registry is configured to run as a standalone server (in this development setups) rather than a clustered peer.

11.3.1 Network Configuration

```
server:
  port: 8761
```

It runs on port **8761**, the standard default port for Eureka servers.

11.3.2 Client Behavior Disabled

Since this instance IS the server, it should not try to register itself as a client.

```
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

- `register-with-eureka: false`: Prevents the server from registering itself as a client application.
- `fetch-registry: false`: Since it holds the data, it doesn't need to cache it from another peer (in a single-node setup).

11.3.3 Eviction and Self Preservation

```
eureka:
  server:
    enable-self-preservation: false
    eviction-interval-timer-in-ms: 5000
```

- **Self Preservation Mode**: Disabled (`false`). Normally, Eureka enters a protection mode if it misses too many heartbeats (assuming network partition). For local development, this is disabled to allow fast detection of down services.
- **Eviction Interval**: **5000ms** (5 seconds). The server checks for dead instances every 5 seconds and removes them from the registry if they haven't renewed their lease (heartbeat).

11.4 Dependencies

File: `pom.xml` The project uses the BOM (Bill of Materials) pattern via the parent `spring-cloud-dependencies`. The critical dependency is:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

This starter pulls in the Netflix Eureka Server libraries, Jersey (JAX-RS implementation), and other required transport layers.

11.5 Purpose in Architecture

The registry enables:

1. **Dynamic Scaling**: New instances of the Driver or Rider services can spin up, register, and immediately receive traffic without reconfiguration.
2. **High Availability**: If a service instance dies, it stops sending heartbeats. The registry evicts it (after 5s), preventing the Gateway from routing traffic to a dead node.
3. **Abstraction**: The Gateway calls `http://user-service` instead of `http://192.168.1.5:9090`.

12 Contracts directory

12.1 Module Overview

The `contracts` folder is a shared library module that defines the **schema** for asynchronous events exchanged via Kafka. It uses **Protocol Buffers (Protobuf)** as the Interface Definition Language (IDL).

12.2 Purpose

In a microservices architecture, services must agree on the structure of the data they exchange. The `contracts` module serves as the single source of truth for these definitions.

- **Type Safety**: Generates Java classes automatically, preventing runtime type errors.
- **Efficiency**: Protobuf messages are serialized into binary, which is significantly smaller and faster to parse than JSON.
- **Decoupling**: Producers and Consumers rely on the generated JAR, not on each other's code.

12.3 Message Definitions

12.3.1 Driver Location Event

File: `driver_location_to_matching.proto`
Topic: `driver-location-topic`
Emitted by the Driver Service to update the Matching Service on a driver's movement.

```
message DriverLocationEvent {
  int64 driverId = 1; // Unique Driver ID
  string oldStation = 2; // Previous Metro Station
  string nextStation = 3; // Upcoming Metro Station
  int32 timeToNextStation = 4; // ETA in seconds
  int32 availableSeats = 5; // Capacity
  string finalDestination = 6; // Trip End Point
  string messageId = 7; // Deduplication ID
}
```

12.3.2 Rider Request Event

File: `rider_to_matching.proto`
Topic: `rider-requests`
Emitted when a Rider initiates a ride search.

```
message RiderRequestDriverEvent {
  int64 riderId = 1;
  string pickUpStation = 2;
  google.protobuf.Timestamp arrivalTime = 3; // Pickup Time
  string destinationPlace = 4;
  string messageId = 5;
}
```

12.3.3 Driver-Rider Match Event

File: `driver_rider_matching.proto`
Topic: `rider-driver-match`
Produced by the Matching Service when a successful pair is found. This event triggers notifications to both parties.

```
message DriverRiderMatchEvent {
  int64 riderId = 1;
  int64 driverId = 2;
  string pickUpStation = 3;
  google.protobuf.Timestamp driverArrivalTime = 4;
  string messageId = 5;
}
```

12.3.4 Trip Completion Events

File: `trip_to_notification.proto`
Used to notify users when trips end or status changes.

- **DriverRideCompletionKafka**: Signal that the driver reached the final destination.
- **RiderRideCompletionKafka**: Signal that the rider was dropped off.
- **DriverLocationForRiderEvent**: Targeted update sent to a specific rider about their matched driver's location.

12.4 Build Process

The module uses the `xolstice-maven-plugin` (or similar Protobuf compiler plugin) in `pom.xml`. During the `mvn clean install` phase:

1. The .proto files are compiled.
2. Java source code is generated in `target/generated-sources`.
3. The code is packaged into a JAR file.
4. Other services (Matching, Driver, etc.) import this JAR as a dependency to use the classes `DriverLocationEvent`, `RiderRequestDriverEvent`, etc.

13 Kafka

13.1 Overview

Apache Kafka acts as the central nervous system of the Metro Car Pooling project, facilitating asynchronous communication, decoupling services, and ensuring high availability. All state changes—from driver movements to trip completions—are propagated as events.

14 Kafka Events Specification

The system defines six primary events. The flow of data is described below:

Topic Name	Event Message (Protobuf)	Producer	Consumer
driver-location-topic	DriverLocationEvent	DriverService	MatchingService, TripService
rider-requests	RiderRequestDriverEvent	RiderService	MatchingService
rider-driver-match	DriverRiderMatchEvent	MatchingService	NotificationService, TripService, DriverService
ride-completion	DriverRideCompletionEvent	DriverService	TripService
driver-ride-completion	DriverRideCompletionKafka	TripService	NotificationService
rider-ride-completion	RiderRideCompletionKafka	TripService	NotificationService
driver-location-rider	DriverLocationForRider	TripService	NotificationService

Table 1: Kafka Event Catalogue

14.1 Detailed Event Logic

14.1.1 1. Driver Location Updates

Topic: `driver-location-topic`
Trigger: The `DriverService` internal simulation moves a driver.
Payload: Current location, Next Station, ETA, Available Seats.
Purpose:

- **MatchingService**: Updates the spatial index (Redis Cache) to make the driver available for new riders.
- **TripService**: Forwards these updates to riders currently in that driver's car.

14.1.2 2. Matching Flow

Topics: `rider-requests` → `rider-driver-match`
Trigger: A rider requests a ride.
Logic:

- `RiderService` pushes the request to the queue.
- `MatchingService` processes the queue, finds a driver, and emits a **Match Event**.
- `NotificationService` alerts both users.
- `TripService` locks the state and initiates the trip.
- `DriverService` updates its cache (decrements seat count).

14.1.3 3. Trip Completion

Topics: `ride-completion` → `driver/rider-ride-completion`
Trigger: Driver reaches the final destination.
Logic:

- `DriverService` detects the end of the route and emits the raw completion signal.

- `TripService` consumes this, looks up all associated riders, and splits the signal into individual notifications (one for the driver, one per rider).

14.2 Idempotent Design Pattern

Given Kafka's "at-least-once" delivery guarantee, consumers may receive duplicate messages (e.g., due to network rebalancing). To prevent state corruption (like booking a driver twice), an **Idempotent Consumer** pattern is implemented using Redis.

14.3 Mechanism

Every Protobuf message includes a UUID `messageId`.

1. **Check**: Before processing, the consumer checks Redis for the key: `prefix + messageId`.
2. **Logic**:
   - **Exists**: Log "Duplicate detected", acknowledge message, and return immediately.
   - **Absent**: Process the business logic.
3. **Mark**: After successful processing, set the key in Redis with a 24-hour TTL.

```
private boolean alreadyProcessed(String messageId) {
    if (redisTemplate.hasKey("processed:" + messageId)) {
        return true;
    }
    // Process logic...
    redisTemplate.opsForValue().set("processed:" + messageId, "1", 24, TimeUnit.HOURS);
    return false;
}
```

14.4 Configuration

- **Partitioning**: Events are keyed by entity ID (e.g., `driverId`) to ensure that all updates for a specific entity land on the same partition, guaranteeing order.
- **Serialization**: Protobuf is used for the value (byte array), and String is used for the key.

14.5 Code Pattern Analysis

The pattern typically looks like this:

```
CompletableFuture<SendResult<String, byte[]>> future =
    kafkaTemplate.send(TOPIC, KEY, VALUE);
future.thenAccept(result -> {
    // 1. Success Callback
    log.debug("Delivered to partition: {}",
        result.getRecordMetadata().partition());
}).exceptionally(ex -> {
    // 2. Failure Callback (Dead Letter Queue logic goes here)
    log.error("Failed to send: {}", ex.getMessage());
    return null;
});
```

14.6 Logic Breakdown

14.6.1 1. The `send()` Method

The `kafkaTemplate.send(topic, key, value)` helper method is called.

- **Behavior**: It serializes the key and value and places the record into an internal buffer. The Kafka Producer client (running in a background I/O thread) batches these records and sends them to the broker.
- **Immediate Return**: The method returns a `CompletableFuture` immediately, **before** the network request has actually completed. This is crucial for high throughput.

14.6.2 2. `thenAccept` (Success Handler)

This lambda function is executed asynchronously only after the Kafka Broker (and replicas, based on `acks` config) has acknowledged persistence of the message.

- **Usage**: primarily used for metrics and debug logging (e.g., verifying which partition the data landed on).

14.6.3 3. `exceptionally` (Failure Handler)

This lambda is executed if the delivery failed (e.g., Broker down, Timeout, Serialization error).

- **Retry Logic**: This is the critical place to implement reliability. In a production system, this block would push the failed payload into a fallback storage (DB/Redis) or a "Dead Letter Queue" for manual retry.
- **Current Implementation**: Currently logs the error, effectively dropping the message (At-Most-Once delivery behavior in failure scenarios).

14.7 Instances in Codebase

This pattern is ubiquitous in the project, found in:

1. **RiderService.java**: When a rider requests a driver.

2. **DriverService.java**:
    - Updating location (Every simulation tick).
    - Reporting ride completion.
3. **MatchingService.java**: Publishing the match event.
4. **TripService.java**:
    - Forwarding driver location to riders.
    - Notifying rider of ride completion.
    - Notifying driver of ride completion.

14.8 Purpose and Benefits

- **Latency Hiding**: The service processing time is strictly `Business Logic + Serialization Time`. It does not include `Network RTT + Broker Disk I/O`, which is handled in the background.
- **Scalability**: Frees up the HTTP request threads (e.g., from the Tomcat pool) immediately, allowing the server to handle more incoming API requests per second.

15 Persistent Caching using Redis

15.1 Overview

Redis is utilized as the **Central Nervous System** of the distributed architecture. reliability. It is not merely a cache but a primary data store for ephemeral state, synchronization, and deduplication across microservices.

15.2 Usage Patterns

15.2.1 1. Dynamic State Caching

Used to store rapidly changing entities that do not require permanent persistence in PostgreSQL.

- **Key**: `drivers`
- **Type**: `Map<Long, DriverCache>` (Serialized JSON)
- **Producer**: `DriverService` (Updated every 2 minutes via Cron)
- **Consumer**: `MatchingService` (Read frequently to find matches)
- **Schema**:

```
{
  "101": {
    "driverId": 101,
    "routePlaces": ["StationA", "StationB", "StationC"],
    "nextPlace": "StationB",
    "timeLeftToNext": 120,
    "availableSeats": 3
  }
}
```

15.2.2 2. Read-Optimized Graph Data

To avoid expensive database joins or graph traversals during matching, pre-computed graph data is loaded into Redis.

- **Key**: `location-location-map`
- **Type**: `Map<String, Map<String, Double>>`
- **Purpose**: O(1) Lookup for distance between any two city nodes.
- **Key**: `nearby-stations`
- **Type**: `Map<String, String>`
- **Purpose**: Maps a city node (lat/long) to its nearest Metro Station.

15.2.3 3. Distributed Queues

Used to buffer riders when no drivers are immediately available.

- **Key**: `rider-waiting-queue`
- **Type**: `List<RiderWaitingQueueCache>`
- **Logic**: FIFO Queue processed by `MatchingService` Cron Job.

15.2.4 4. Event Deduplication (Idempotency)

To guarantee **Exactly-Once Semantics** for Kafka consumers, processed message IDs are stored with a TTL.

- **Key Prefix**: `match_found_processed_kafka_msg:`
- **Value**: `"1"`
- **TTL**: 24 Hours
- **Logic**:

```
if (redisTemplate.hasKey(key + messageId)) {
    return; // Duplicate detected, skip processing
}
redisTemplate.opsForValue().set(key + messageId, "1", 24, HOURS);
process(event);
```

15.2.5 5. Distributed Locking

To prevent race conditions when multiple instances of a service try to update the same cache (e.g., Driver updating location while Matching Service reserves a seat).

- **Mechanism**: Redis `SETNX` with expiration.
- **Keys**: `lock:drivers`, `lock:trip`, `lock:rider-waiting-queue`.
- **Implementation**: Custom Retry Policy (10 retries, 200ms backoff).

```
String lock = redisDistributedLock.acquireLock("lock:drivers", 5000);
if (lock != null) {
    try {
        // Critical Section: Read -> Modify -> Write Cache
    } finally {
        redisDistributedLock.releaseLock("lock:drivers", lock);
    }
}
```

15.3 Summary of Keys

Redis Key	Service	Description
`drivers`	Driver	Real-time state of all active drivers.
`trip-cache`	Trip	Active Rider-Driver pairings.
`rider-waiting-queue`	Matching	Buffer for unmatched riders.
`location-location-map`	Infra	Distance matrix for city graph.
`nearby-stations`	Infra	Node-to-Station mapping.

16 Redis Distributed Locking

16.1 Introduction

The `RedisDistributedLock` component provides a mechanism to ensure mutual exclusion across multiple instances of the application. It utilizes Redis as a central coordinator to manage lock states, ensuring that critical sections (such as matching algorithms or cache updates) are executed by only one process at a time.

16.2 Lock Acquisition Logic

16.2.1 Mechanism

The `acquireLock` method attempts to obtain a lock for a specific key.

- **Input**: `lockKey` (Resource identifier), `timeoutMs` (Lock expiration time).
- **Output**: A unique `lockValue` (UUID) if successful, or `null` if the lock is already held.

16.2.2 Algorithm

The implementation relies on the Redis atomic command `SET ... NX PX`.

1. Generate a unique identifier V_{lock} = UUID.randomUUID().
2. Execute the atomic operation:

$$\text{Success} \leftarrow \text{SET}(K_{lock}, V_{lock}, \text{NX}, \text{PX } T_{expiry})$$

- **NX (Not Exists)**: Ensuring the write only happens if the key does not exist.
- **PX (Partition Expiry)**: Setting the Time-To-Live (TTL) to T_{expiry} milliseconds to prevent deadlocks if the application crashes.

$$\text{Result} = \begin{cases} V_{lock} & \text{if Redis returns OK (True)} \\ \text{null} & \text{otherwise} \end{cases} \tag{2}$$

16.3 Lock Release Logic

16.3.1 Mechanism

The `releaseLock` method ensures that a lock is only released by the client that originally acquired it. This prevents accidental release of a lock that may have expired and been re-acquired by another process.

16.3.2 The "Check-Then-Delete" Problem

A naive implementation might Check if the value matches, then Delete. However, this introduces a race condition:

1. Client A reads Key, sees Value A.
2. (Context Switch) Key expires. Client B acquires Key with Value B.
3. Client A sends Delete command, wrongly removing Client B's lock.

16.3.3 Atomic Lua Solution

To solve the race condition, the release logic is executed via a **Lua script**, which Redis guarantees to execute atomically. **Lua Script**:

```
if redis.call('get', KEYS[1]) == ARGV[1] then
    return redis.call('del', KEYS[1])
else
    return 0
end
```

16.3.4 Mathematical Logic

Let K be the lock key, V_{stored} be the current value in Redis, and V_{owner} be the local UUID held by the requester. The operation $\Phi(K, V_{owner})$ is defined as:

$$\Phi(K, V_{owner}) = \begin{cases} \text{DEL}(K) \to 1 & \text{if GET}(K) = V_{owner} \\ 0 & \text{if GET}(K) \neq V_{owner} \vee \text{GET}(K) \text{ is null} \end{cases} \tag{3}$$

This ensures strong consistency where the lock is released if and only if:

1. The lock exists.
2. The lock value matches the owner's unique ID.

17 gRPC

17.1 Overview

The Metro Car Pooling system utilizes **gRPC** (Google Remote Procedure Call) for high-performance, synchronous, and asynchronous inter-service communication through Kafka, using .proto files to define the kafka message structure. The architecture follows a strict pattern where the **API Gateway** acts as a gRPC Client, interacting with the backend microservices (User, Rider, Driver, Notification), which act as gRPC Servers.

17.2 Architectural Pattern

The codebase follows a consistent Separation of Concerns pattern:

1. **Controller / Gateway Layer**: Uses a `GrpcClient` wrapper. It does not know about Protobuf internals; it deals with DTOs and delegates the call to the client wrapper.

2. **gRPC Client**: Converts DTOs to Protobuf messages, handles Service Discovery (Eureka), manages the gRPC Channel, and invokes the Stub.
3. **gRPC Server**: Receives the Protobuf message, extracts data, and calls the respective **Business Service** (Spring Service).
4. **Business Service**: Executes the core logic (DB access, calculations) and returns domain entities.
5. **gRPC Server**: Wraps the domain entity results back into a Protobuf response and sends it to the client.

17.3 User Service gRPC Logic

17.3.1 User gRPC Server

File: `UserGrpcServer.java`
This class extends the auto-generated `UserServiceImplBase`.

- **Integration**: It injects the `UserService`.
- **Methods**: Implements `driverSignUpRequest`, `riderSignUpRequest`, etc.
- **Logic**:
  1. Receives a request (e.g., `DriverSignUp`).
  2. Extracts fields (username, password, license).
  3. Delegated to `userService.driverSignUp(...)`.
  4. Maps the resulting `DriverEntity` to a `SignUpOrLoginResponse` Protobuf object containing the user ID and status code.
  5. Uses `responseObserver.onNext(...)` to send the single response.

17.3.2 User gRPC Client

File: `UserGrpcClient.java`
Located in the Gateway, this client handles communication with the User Service.

- **Service Discovery**: Uses `DiscoveryClient` to find the "user" service instance and its gRPC port (metadata `grpc.port`).
- **Channel Management**: Creates a `ManagedChannel` for the request.
- **Blocking Stub**: Uses `UserServiceBlockingStub` for synchronous request-response calls.
- **Logic**: Wraps DTOs (e.g., `DriverSignUpRequestDTO`) into Protobuf objects, makes the remote call, and handles exceptions by returning a default failure response (Status 401).

17.4 Rider Service gRPC Logic

17.4.1 Rider gRPC Server

File: `RiderGrpcServer.java`

- **Service Integration**: Injects `RiderService`.
- **Method**: `postRiderInfo`.
- **Logic**: Calls `riderService.processRiderInfo` with pickup, destination, and arrival time. Returns a boolean status wrapped in `RiderStatusResponse`.

17.4.2 Rider gRPC Client

File: `RiderGrpcClient.java`

- **Optimization**: Unlike the User client, this client attempts to cache and reuse the `ManagedChannel` using an `AtomicReference`, improving performance by reducing connection overhead.
- **Logic**: Converts arguments to `PostRider` proto message and executes the call via `RiderServiceBlockingStu`

17.5 Driver Service gRPC Logic

17.5.1 Driver gRPC Server

File: `DriverGrpcServer.java`

- **Service Integration**: Injects `DriverService`.
- **Method**: `postDriverInfo`.
- **Logic**: Accepts calls containing route stations and capacity. Delegates simulation initiation to `DriverService`.

17.5.2 Driver gRPC Client

File: `DriverGrpcClient.java`

- Similar to the User client, it connects to the "driver" service via Eureka lookup.
- Builds `PostDriver` messages containing complex repeated fields (lists of route stations).

17.6 Notification Service gRPC Logic (Streaming)

The Notification service logic differs significantly as it uses **Server-Side Streaming** to push real-time updates to the Gateway.

17.6.1 Notification gRPC Server

File: `NotificationGrpcServer.java`

- **Streaming Logic**: Instead of a simple return, it subscribes to reactive streams (Flux) provided by `NotificationService`.
- **Backpressure**:

```
notificationService.streamRiderDriverMatches()
    .doOnNext(data -> responseObserver.onNext(data))
    .subscribe();
```

  It pushes events to the `responseObserver` as they occur in real-time.

17.6.2 Notification gRPC Client

File: `NotificationGrpcClient.java`
This client bridges the gap between gRPC streams and Spring WebFlux `Flux`.

- **Stub**: Uses 'newStub' (Async Stub) instead of Blocking Stub.
- **Flux Adaptation**:
  - Creates a `Flux` using `Flux.create(sink -> ...)`.
  - Passes a `StreamObserver` to the gRPC call.
  - When the server pushes data (`onNext`), the client observer emits it to the Flux sink (`sink.next(value)`).
- This enables the Gateway to stream these events directly to frontend clients (e.g., via Server-Sent Events).

18 gRPC Service Discovery via Eureka

18.1 The Challenge

Standard Spring Cloud integration automatically handles service discovery for REST (HTTP) traffic (e.g., via `RestTemplate` or `Feign`). However, **gRPC** runs on a separate port (typically 9090+) and uses HTTP/2, which the standard load balancers do not natively route.

18.2 The Solution: Metadata Maps

To bridge this gap, the project utilizes the **metadata-map** feature of the Eureka instance registry. Each microservice explicitly registers its auxiliary gRPC port alongside its standard HTTP port.

18.3 Implementation Details

18.3.1 1. Service Registration (Microservice Side)

In the `application.yaml` of every microservice (e.g., `user`, `driver`), the configuration explicitly adds the gRPC port to the instance metadata. **File**: `user/src/main/resources/application.yaml`

```yaml
grpc:
  server:
    port: 9095 # The actual port gRPC server binds to
eureka:
  instance:
    prefer-ip-address: true
    metadata-map:
      grpc.port: 9095 # <--- CRITICAL: Advertising the port
```

18.3.2 2. Service Discovery (Gateway Side)

The Gateway client code acts as a **Smart Client**. Instead of connecting to a hardcoded URL, it queries Eureka. **Logic Flow**:

1. **Lookup**: `discoveryClient.getInstances("service-name")` returns a list of active `ServiceInstance` objects.
2. **Extraction**: The code inspects the `metadata` map of the instance to retrieve the custom `grpc.port` value.
3. **Connection**: A `ManagedChannel` is built using the resolved IP address and the custom gRPC port.

Code Example: `UserGrpcClient.java`

```java
// 1. Get Instance from Eureka
ServiceInstance instance = discoveryClient.getInstances("user")
    .stream().findFirst().orElseThrow();
// 2. Extract Metadata using Helper Method
int port = getGrpcPort(instance);
// 3. Connect directly to that IP:Port
ManagedChannel channel = ManagedChannelBuilder
    .forAddress(instance.getHost(), port)
    .usePlaintext()
    .build();
// Helper Method Logic
private int getGrpcPort(ServiceInstance instance) {
    // Reads the key "grpc.port" originating from the yaml above
    String port = instance.getMetadata().get("grpc.port");
    return Integer.parseInt(port);
}
```

18.4 Architecture Diagram

1. **Boot Up**: User Service starts on HTTP 8081 / gRPC 9095.
2. **Register**: Sends heartbeat to Eureka with metadata {`grpc.port: 9095`}.
3. **Request**: Gateway receives sign-up request.
4. **Query**: Gateway asks Eureka: "Where is 'user' service?"
5. **Resolve**: Eureka returns IP + Metadata.
6. **Call**: Gateway opens socket directly to IP:9095.

19 Frontend

19.1 Overview

The Frontend is a modern Single Page Application (SPA) built with **Next.js 16 (App Router)** and **React 19**. It delivers a responsive, role-based interface for Drivers and Riders, synchronizing state in real-time with the backend via Server-Sent Events (SSE).

19.2 Tech Stack

- **Framework**: Next.js 16 (React 19)
- **Language**: TypeScript
- **Styling**: Tailwind CSS 4 + Radix UI Primitives (Headless UI)
- **Icons**: Lucide React
- **Forms**: React Hook Form + Zod Validation
- **Real-Time**: Native `EventSource` API (SSE)

19.3 Architecture Design

19.3.1 1. Routing Strategy

The application uses file-system based routing under the `app/` directory:

- `/auth`: A unified entry point for authentication. It features a toggle Switch to select the persona (**Driver** vs **Rider**) and mode (**Login** vs **Signup**).
- `/driver`: The protected dashboard for drivers.
- `/rider`: The protected dashboard for riders.

Note: Route protection is implemented via client-side checks on `localStorage` tokens in `useEffect`.

19.3.2 2. State Management

Instead of a heavy global store (Redux), the app uses a **State Machine Pattern** local to each dashboard page. **States**:

1. `IDLE`: User is filling out a request/offer form.
2. `WAITING`: Request submitted; listening for matches via SSE.
3. `MATCHED`: A match is found; showing confirmation modal.
4. `ACTIVE`: Trip is in progress; listening for location updates.

19.4 Implementation Logic

19.4.1 1. Authentication Flow (`auth/page.tsx`)

- **Logic**: Submits form data to the API Gateway (`/api/user/login-driver` or `/api/user/add-driver`).
- **Persistence**: On success, the JWT token, User ID, and Role are stored in `localStorage`. This enables session persistence across page reloads.

19.4.2 2. Real-Time Matching (`Waiting State`)

When a user submits a request, the component enters the `WAITING` state and opens an SSE connection. **Code Pattern (Rider Example)**:

```
useEffect(() => {
  if (rideState !== 'waiting') return;
  // Connect to Gateway SSE Endpoint
  const eventSource = new EventSource(
    `${API_URL}/api/notification/matches?status=true`,
    { withCredentials: true }
  );
  eventSource.onmessage = (event) => {
    const match = JSON.parse(event.data);

    // Client-side filtering: Check if this match belongs to ME
    if (match.riderId === currentUserId) {
      setCurrentMatch(match);
      setRideState('matched'); // Trigger Modal UI
    }
  };
  return () => eventSource.close();
}, [rideState]);
```

19.4.3 3. Active Trip Monitoring (`Active State`)

Once the trip begins, the Dashboard switches tabs to the "Active Trip" view and opens two new SSE channels:

1. **Location Updates**: Listens to `/driver-location-for-rider`.
   - **Rider UI**: Updates the "Next Station" and "Time to Arrival" display live.
2. **Completion Events**: Listens to `/ride-completion`.
   - **Trigger**: When received, shows a "Ride Completed" modal and resets the state machine to `IDLE`.

19.5 Component Design

The application separates **Smart Pages** (Logic) from **Dumb Components** (Presentation).

- `page.tsx`: Handles all API calls, SSE connections, and state transitions.

- `components/rider-trip-view.tsx`: Purely visual. Renders the progress based on props passed from the page.
- `components/matching-modal.tsx`: Reusable dialog for accepting/rejecting matches.

20 CSV generator codes

20.1 Overview

The infrastructure module generates a synthetic city topology and simulates the placement of metro stations. This process is divided into two Python scripts:

1. `generate_city_graph.py`: Creates a fully connected graph of city nodes with random distance weights.
2. `location_nearby.py`: Selects optimal metro station locations to achieve a specific coverage target (e.g., 30% of the city).

20.2 City Graph Generation

File: `infra/csv_generation/generate_city_graph.py`

20.2.1 Logic

This script constructs a synthetic dataset representing distances between all possible pairs of locations in the city.

1. **Node Generation**: Nodes are labeled using a combination of a letter ($a - z$) and a number ($1 - k$).
$$\text{Nodes} = \{L_i \mid L \in \{'a', \ldots, 'z'\}, i \in \{1, \ldots, k\}\}$$
Where $k = 10$, resulting in $26 \times 10 = 260$ total nodes.
2. **Edge Weighting**: For every pair of nodes (u, v), a distance weight $w_{u,v}$ is assigned.
   - If $u = v$, $w_{u,v} = 0$.
   - If $u \neq v$, $w_{u,v} \sim U(1, \text{max\_distance})$, where U is a uniform integer distribution.
   - Implementation uses `random.randint(1, 20)`.
3. **Output**: Writes a compressed CSV (`.csv.gz`) containing $N \times N$ rows (including self-loops) representing the complete distance matrix.

20.3 Metro Station Selection & Mapping

File: `infra/csv_generation/location_nearby.py`

20.3.1 Objective

The goal is to select a subset of nodes $\mathcal{S}$ (Metro Stations) and a distance threshold t such that approximately 30% of all city nodes are within distance t of at least one station $s \in \mathcal{S}$.

20.3.2 Mathematical Model

Let V be the set of all nodes, $|V| = N$. Let $D(u, v)$ be the distance between nodes u and v. Defined parameters:

- $P_{target} = 0.30$ (Target Coverage Percentage).
- $N_{target} = \text{round}(P_{target} \times N)$.

20.3.3 Algorithm: Greedy Set Cover Approximation

The script iterates through possible thresholds $t \in [1, \max(D)]$ and station counts $S \in [1, 30]$ to find the optimal configuration.

Algorithm 2 Metro Station Selection Logic

Input: Graph $G(V, E)$, Target Count mapping target N_{target}
$Results \leftarrow []$
for $t \leftarrow 1$ to $\max(distance)$ **do** ▷ Iterate Distance Thresholds
    Compute Neighborhoods $N_t(u) = \{v \in V \mid D(u, v) \leq t\}$
    for $S \leftarrow 1$ to $MAX\_STATIONS$ **do** ▷ Iterate Station Counts
        Greedy Selection:
        Sort nodes by degree $|N_t(u)|$ descending.
        Select top S nodes as stations: $\mathcal{S}_{cand} = \{s_1, \ldots, s_S\}$.
        Calculate Coverage:
        $Covered = \bigcup_{s \in \mathcal{S}_{cand}} N_t(s)$
        $Count = |Covered|$
        Store $(t, S, Count)$ in $Results$.
    end for
end for
Optimization:
Find tuple (t^*, S^*) minimizing $|Count - N_{target}|$.
If multiple match, favor smaller difference then smaller S.

20.3.4 Mapping Logic

Once the optimal threshold t^* and stations $\mathcal{S}^*$ are chosen, every node $u \in V$ is mapped to a station using the following logic:

$$M(u) = \begin{cases} s_k & \text{if } u = s_k \in \mathcal{S}^* \\ \arg\min_{s \in \mathcal{S}^*} \{D(u,s) \mid D(u,s) \leq t^*\} & \text{if } \exists s, D(u,s) \leq t^* \\ \text{None (Empty)} & \text{otherwise} \end{cases} \tag{4}$$

Details:

1. If a node is a station itself, it maps to itself.
2. If a node is within threshold t^* of multiple stations, it maps to the **nearest** one.
3. Nodes farther than t^* from any station remain unmapped.

20.3.5 Outputs

- **location_nearby.csv**: Mapping of City Point → Metro Station (e.g., $a1 \rightarrow ME3$).
- **stations.json**: List of selected station nodes and configuration parameters.
- **mapping.json**: JSON representation of the CSV mapping.
- **metadata.json**: Statistics including actual coverage percentage achieved.

21 Logstash

21.1 Overview

The Logstash pipeline acts as the **Processing Layer** of the ELK stack (Elasticsearch, Logstash, Kibana). It is responsible for ingesting raw log data from all microservices, structuring it, enriching it with metadata, and shipping it to the central search index.

21.2 Pipeline Structure

The configuration file (`infra/logstash-pipeline/logstash.conf`) consists of three standard sections:

1. **Input**: How data enters the pipeline.
2. **Filter**: How data is transformed and enriched.
3. **Output**: Where the data is sent.

21.3 Detailed Logic

21.3.1 1. Input Stage

The pipeline listens on a TCP socket, expecting new-line delimited JSON objects.

```
input {
  tcp {
    port => 5044
    codec => json_lines
  }
}
```

Note: All microservices are configured to ship logs to `logstash:5044`.

21.3.2 2. Filter Stage (The Logic Core)

This section applies a series of transformations:

21.3.3 A. Standardization

- **Timestamp**: Parses the ISO8601 string from the log into Logstash's native `@timestamp` field.
- **Field Renaming**: Renames `level` → `log_level` and `thread` → `thread_name` for consistency.

21.3.4 B. Event Categorization

The pipeline uses **Regex Matching** on the `message` body to tag generic logs with specific event types. This enables powerful filtering in Kibana later (e.g., "Show me all matching events").

```
if [message] =~ /^Matches found:/ {
    mutate { add_field => {"log_type" => "rider_driver_matching_event"} }
} else if [message] =~ /^Driver location:/ {
    mutate { add_field => {"log_type" => "driver_location_event"} }
}
# ... similarly for signups, logins, etc.
```

21.3.5 C. Severity scoring

Maps text-based log levels to numeric values to allow for "greater than" queries (e.g., *severity ¿ 2*).

- **ERROR** = 4
- **WARN** = 3
- **DEBUG** = 2
- **INFO** = 1

21.3.6 3. Output Stage

The enriched data is sent to two destinations:

1. **Elasticsearch**: The primary storage. Data is partitioned into daily indices (`microservices-2024.12.08`).
2. **Stdout**: For debugging the pipeline itself (printed to the Logstash container logs).

```
output {
  elasticsearch {
    hosts => ["http://elasticsearch:9200"]
    index => "microservices-%{+YYYY.MM.dd}"
  }
}
```

21.4 Purpose

This configuration transforms unstructured text logs into **Structured Data**. Instead of just searching for text, developers can now query: *"Count 'rider_login_events' where 'severity' >= 3 in the last 15 minutes"*.

22 Centralized Logging and Monitoring

22.1 Overview

The Metro Car Pooling system implements a robust, centralized logging and monitoring infrastructure using the **ELK Stack** (Elasticsearch, Logstash, Kibana). This enables real-time observability, log aggregation from all microservices, and powerful analytics capabilities for debugging, performance monitoring, and business intelligence.

22.2 Architecture Overview

The logging pipeline follows a standard three-tier architecture:

1. **Collection Layer**: Each microservice uses Logback with Logstash encoder to generate structured JSON logs.
2. **Processing Layer**: Logstash receives logs via TCP, enriches them with metadata, categorizes events, and indexes them.
3. **Storage & Visualization Layer**: Elasticsearch stores indexed logs, and Kibana provides query and visualization capabilities.

22.3 Logback Configuration

22.3.1 TCP Socket Appender

Each microservice (User, Driver, Rider, Matching, Trip, Notification) is configured with `logback-spring.xml` to send logs to Logstash over TCP using the **LogstashTcpSocketAppender**.

File: `<service>/src/main/resources/logback-spring.xml`

22.3.2 Key Configuration Parameters

- **Destination**: `${LOGSTASH_HOST}:${LOGSTASH_PORT}` (Default: `logstash:5044`)
- **KeepAliveDuration**: 5 minutes - TCP connection persistence
- **ReconnectionDelay**: 5000ms - Automatic retry on connection failure
- **WriteBufferSize**: 8192 bytes - Buffering for batch transmission
- **ImmediateFlush**: true - Critical logs are sent immediately

22.3.3 JSON Serialization

The encoder uses `LogstashEncoder` which automatically converts log events to structured JSON with the following fields:

```
{
  "@timestamp": "2024-12-08T12:34:56.789Z",
  "message": "Driver signup: success",
  "logger_name": "com.metrocarpool.user.service.UserService",
  "thread_name": "http-nio-8081-exec-5",
  "level": "INFO",
  "application": "user-service",
  "stack_trace": null
}
```

22.3.4 Custom Fields

Each service adds its application name via custom fields:

```
<customFields>{"application":"${APP_NAME}"}</customFields>
```

This allows filtering logs by service in Kibana (e.g., `application:"driver-service"`).

22.3.5 Dual Appender Strategy

Logs are written to two destinations simultaneously:

1. **CONSOLE**: Human-readable format for local development and Docker logs.
2. **LOGSTASH**: Structured JSON format for centralized aggregation.

22.4 Dependency Management

The Logstash encoder is added as a managed dependency in the project's parent `pom.xml`:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>7.4</version>
</dependency>
```

All child services (User, Driver, Rider, etc.) inherit this dependency automatically.

22.5 ELK Stack Deployment

22.5.1 Docker Compose Configuration

The entire ELK stack is deployed via Docker Compose in `infra/docker-compose.yml`:
Elasticsearch (Port 9200):

- Single-node cluster for development.
- Security disabled for simplified local access.
- JVM heap: 512MB (configured via `ES_JAVA_OPTS`).
- Persistent volume: `es-data:/usr/share/elasticsearch/data`.

Logstash (Port 5044):

- Listens on TCP 5044 for incoming log streams.
- Pipeline configuration mounted from `./logstash-pipeline/logstash.conf`.
- Depends on Elasticsearch for output.

Kibana (Port 5601):

- Web UI for querying and visualizing log data.
- Connected to Elasticsearch at `http://elasticsearch:9200`.
- Accessible at `http://localhost:5601`.

22.6 Log Field Structure

After processing through Logstash, each log entry contains:

22.6.1 Standard Fields

22.6.2 Event-Type Enrichment

Logstash adds a `log_type` field based on message pattern matching:

22.7 Kibana Data View Configuration

To visualize logs in Kibana, a Data View is created with:

- **Index Pattern**: `microservices-*`
- **Timestamp Field**: `@timestamp`

This allows querying logs across all daily indices (e.g., `microservices-2024.12.08`).

Field Name	Type	Description
`@timestamp`	DateTime	ISO8601 formatted event time
`application`	String	Service name (user, driver, rider, etc.)
`log_level`	String	Log severity (INFO, WARN, ERROR, DEBUG)
`severity`	Integer	Numeric severity (1-4 for filtering)
`message`	String	Log message content
`thread_name`	String	Execution thread identifier
`logger_name`	String	Fully qualified class name
`environment`	String	Deployment environment (development)

Table 2: Standard Log Fields

Log Type	Message Pattern	Use Case
`driver_signup_event`	`^Driver signup:`	Track driver onboarding
`rider_signup_event`	`^Rider signup:`	Track rider onboarding
`driver_login_event`	`^Driver login:`	Authentication monitoring
`rider_login_event`	`^Rider login:`	Authentication monitoring
`rider_driver_matching_event`	`^Matching:`	Match success rate analysis
`rider_added_to_waiting_queue_event`	`^Rider waiting queue:`	Queue depth tracking
`user_reached_destination_event`	`^Trip completion:`	Trip completion analytics
`driver_location_event`	`^Driver location:`	Real-time tracking logs

Table 3: Event Type Categories

22.8 Benefits of the Logging System

1. **Centralization**: All microservice logs aggregated in one searchable index.
2. **Structured Querying**: Filter by service, severity, event type, and time range.
3. **Real-Time Monitoring**: Logs are searchable within seconds of generation.
4. **Fault Tolerance**: Logback's reconnection mechanism ensures logs are not lost during Logstash restarts.
5. **Scalability**: Daily indices prevent index bloat and allow for time-based retention policies.

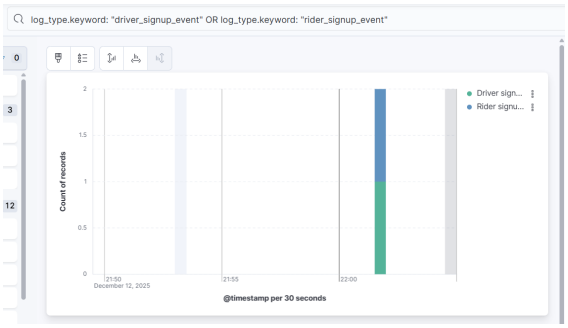
22.9 Some Kibana Visualizations

Based on the enriched log fields and event types, the following charts and visualizations are recommended for the Kibana dashboard:

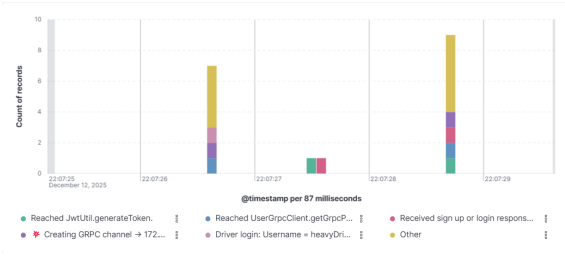
22.9.1 1. Event Rate Over Time (Line Chart)

Metric: Count of documents
X-Axis: `@timestamp` (Time buckets: 5-minute intervals)
Split Series: `log_type.keyword`
Purpose: Visualize the rate of different events (signups, logins, matches, etc.) over time to identify traffic patterns and peak usage periods.

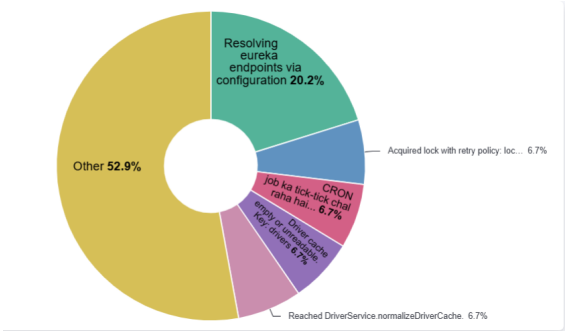
22.9.2 Driver and Rider signup event captured



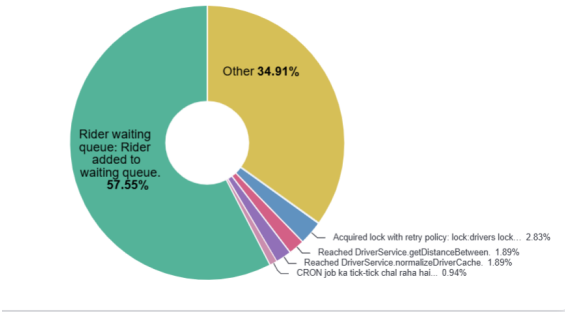
(a) Driver and rider signup events



(b) Driver and rider login events



(c) Initial log distribution at startup

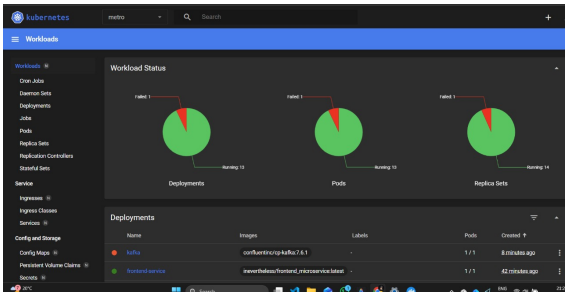


(d) Ride request posted by rider and driver

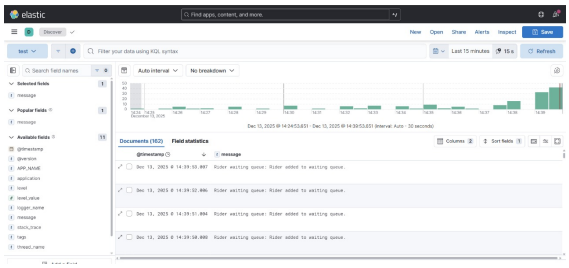
(e) Live driver movement observed with various fields

(f) Kibana logs discovery

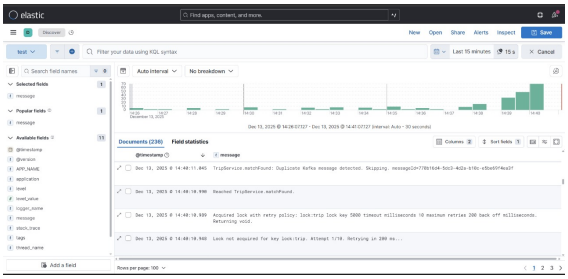
Figure 4: Kibana visualization grid for system events (Part 1)



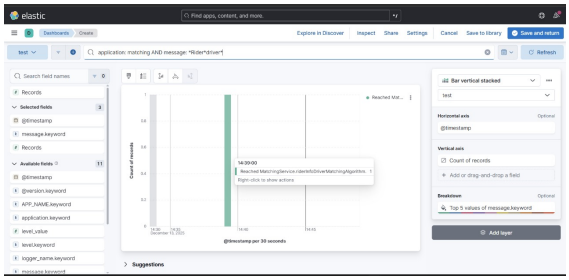
(a) Monitoring using metrics server facility in-built in Minikube



(b) Event "rider added to waiting queue" as seen on Kibana dashboard



(c) Event "rider driver match found" as seen on Kibana dashboard



(d) Rider driver matching log filtered using KQL query

Figure 5: Kibana visualization grid for system events (Part 2)

22.9.3 2. Service-Wise Log Distribution (Pie Chart)

Metric: Count of documents
Slice By: `application.keyword`
Purpose: Show the proportion of logs generated by each microservice. Helps identify chatty services and potential logging issues.

22.9.4 3. Severity Level Breakdown (Bar Chart)

Metric: Count of documents
X-Axis: `log_level.keyword`
Color: `severity` (Gradient: Green for INFO, Yellow for WARN, Red for ERROR)
Purpose: Monitor error rates and system health. Sudden spikes in ERROR logs indicate system issues.

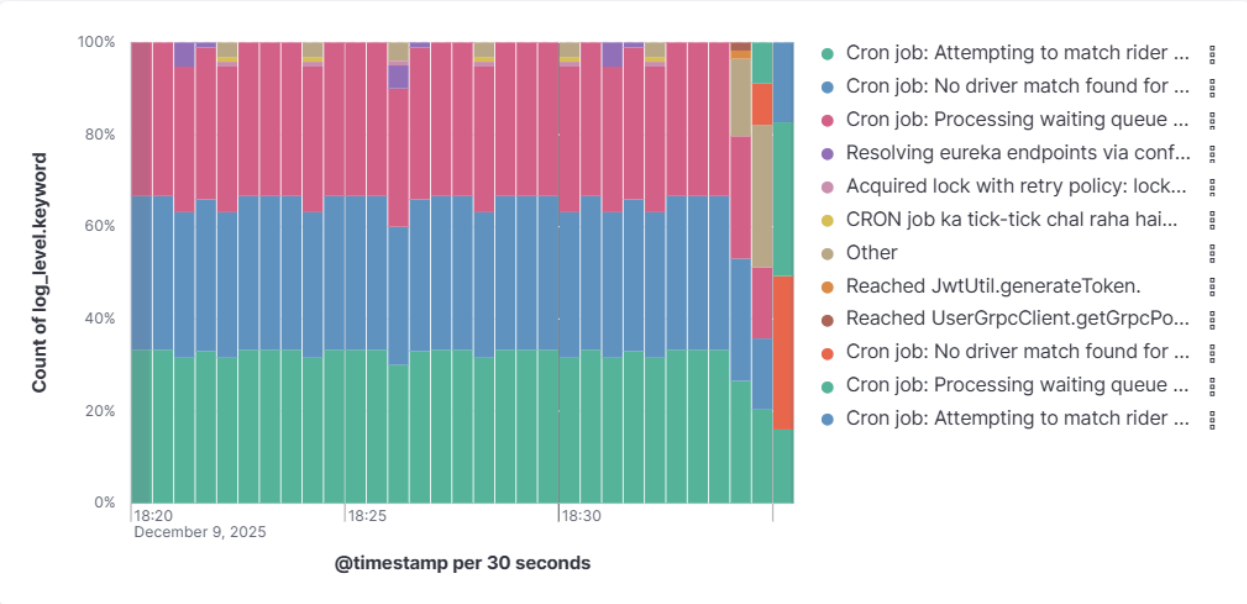


Figure 6: Bar Chart showing security level breakdown

22.9.5 4. Matching Success Rate (Metric Visualization)

Filter: `log_type:"rider_driver_matching_event"`
Metric: Count
Time Range: Last 24 hours
Purpose: Display total successful matches to track system effectiveness. Can be paired with waiting queue metrics.

22.9.6 5. Rider Waiting Queue Depth (Area Chart)

Filter: `log_type:"rider_added_to_waiting_queue_event"`
Metric: Count
X-Axis: `@timestamp` (Buckets: 1-minute intervals)
Purpose: Visualize how many riders are added to the waiting queue over time. Helps identify periods of high demand vs. low driver availability.

22.9.7 6. Authentication Events Heatmap (Heat Map)

Filter: `log_type:(driver_login_event OR rider_login_event)`
Y-Axis: Hour of day
X-Axis: Day of week
Color Intensity: Document count
Purpose: Identify peak login times to optimize server capacity and understand user behavior patterns.

22.9.8 7. Error Logs Table (Data Table)

Filter: `severity >= 3`
Columns: `@timestamp`, `application`, `log_level`, `message`, `logger_name`

Sort: `@timestamp` descending
Purpose: Real-time error monitoring dashboard for immediate incident response.

22.9.9 8. Top Active Services (Top Values)

Metric: Count
Field: `logger_name.keyword`
Size: Top 10
Purpose: Identify which specific classes/services are generating the most logs, useful for debugging verbose components.

22.9.10 9. Trip Completion Rate (Gauge)

Filter: `log_type:"user_reached_destination_event"`
Metric: Count
Time Comparison: Current hour vs. previous hour
Purpose: Monitor successful trip completions as a key business metric.

22.9.11 10. Driver Location Updates Frequency (Metric)

Filter: `log_type:"driver_location_event"`
Metric: Count per second
Purpose: Monitor the real-time location update stream to ensure the driver tracking simulation is functioning correctly.

22.10 Index Lifecycle Management

Logs are indexed with daily rotation:

- Pattern: `microservices-%{+YYYY.MM.dd}`
- Example: `microservices-2024.12.08`

This allows for efficient deletion of old data (e.g., retain logs for 30 days) without affecting current indices.

23 Docker

23.1 Overview

The project adopts a **Cloud-Native** approach, where every component—from the databases to the frontend—is containerized using Docker. The orchestration is modularized, with a central infrastructure stack and decoupled service descriptions.

23.2 Dockerfiles: Multi-Stage Build Strategy

To optimize image size and security, all services utilize **Multi-Stage Builds**. This separates the heavy build tools (Maven/Node) from the lean runtime environment (JRE/Alpine).

23.2.1 1. Java Microservices (User, Rider, Driver, etc.)

Pattern: `maven:3.9.9-eclipse-temurin-21` → `eclipse-temurin:21-jre-jammy`

```
# STAGE 1: Build
FROM maven:3.9.9-eclipse-temurin-21 AS builder
WORKDIR /build
COPY . .
# -pl user: Build only the specific module
# -am: Also make dependencies (contracts)
RUN mvn -pl user -am clean package -DskipTests
# STAGE 2: Runtime
FROM eclipse-temurin:21-jre-jammy
WORKDIR /app
# Copy only the compiled JAR from the builder stage
COPY --from=builder /build/user/target/*.jar user_service.jar
EXPOSE 8081 9095
ENTRYPOINT ["java", "-jar", "user_service.jar"]
```

Benefits:

1. **Size**: The final image is ∼200MB (JRE only) vs ∼800MB (JDK + Maven Cache).
2. **Security**: Source code and build tools are discarded, leaving a minimal attack surface.

23.2.2 2. Frontend (Next.js)

Pattern: `node:20-bookworm-slim` (Base → Deps → Builder → Runner) The frontend uses a 4-stage pipelines to cache 'node_modules' efficiently and utilize Next.js's 'standalone' output mode for minimal production artifacts.

23.3 Orchestration: Docker Compose

The system uses a **Federated Compose Architecture**. Instead of one giant monolithic file, the configuration is split by domain.

23.3.1 1. Infrastructure Stack (`infra/docker-compose.yml`)

This file defines the shared backbone network `infra_default` and hosts stateful services.

- **PostgreSQL** (5432): User persistence.
- **Redis** (6379): Caching and Distributed Locks.
- **Kafka + Zookeeper** (9092): Event Broker(enables event transmission and receival).
- **Eureka** (8761): Service Registry.
- **ELK Stack** (Elasticsearch, Logstash, Kibana): Centralized Logging.
- **Cache Loader**: One-off Python container to seed city graph data into Redis.

23.3.2 2. Service Stacks

Each microservice (e.g., `user/docker-compose.yml`) defines its own deployment, attaching to the external `infra_default` network.

```
services:
  user-service:
    build:
      context: ..
      dockerfile: user/Dockerfile
    networks:
      - infra_default # Connects to Postgres/Kafka/Eureka
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/metrocarpool
      EUREKA_CLIENT_SERVICEURL_DEFAULTZONE: http://registry:8761/eureka
```

23.4 Networking

All containers communicate over the user-defined bridge network `infra_default`. Service discovery is performed via DNS (e.g., `jdbc:postgresql://postgres...`) or Eureka (for inter-service gRPC).

24 Jenkins

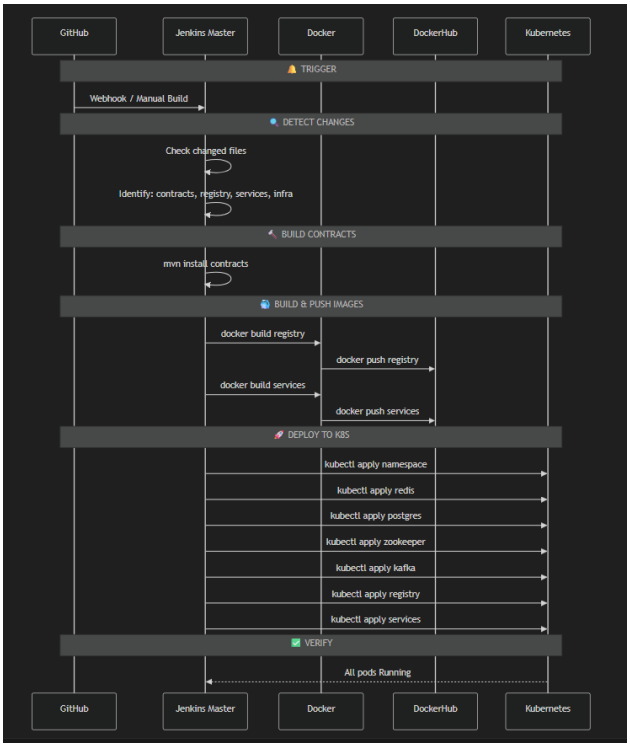


Figure 7: Jenkins CI/CD Pipeline Flow

24.1 Introduction

This document provides a detailed analysis of the Continuous Integration and Continuous Deployment (CI/CD) pipelines implemented for the Metro Car Pooling project. The system employs a "Master-Slave" (or Orchestrator-Worker) architecture using Jenkins pipelines to efficiently manage the build and deployment of multiple microservices.

24.2 Master-Slave Architecture Design

The CI/CD strategy is designed to handle a monorepo structure containing multiple microservices. Instead of triggering all builds on every commit, the system uses a central Orchestrator pipeline Jenkinsfile.master to intelligently detect changes and trigger specific downstream pipelines for individual microservices.

24.2.1 Architectural Components

- **Master Orchestrator (Jenkinsfile.master)**: The entry point for all webhook triggers. It analyzes the git diff to identify modified directories and maps them to specific services. It handles shared concerns like Infrastructure updates, Contract builds, and Registry updates.
- **Microservice Pipelines (Slave Pipelines)**: Individual pipelines residing in each microservice directory (e.g., driver/Jenkinsfile); these pipelines are responsible for the specific build, test, dockerization, and deployment lifecycle of a single service.

24.2.2 Workflow Overview

1. **Trigger**: A push to the GitHub repository triggers the Master pipeline.
2. **Change Detection**: The Master pipeline calculates the diff between the current commit and the previous one.
3. **Decision Matrix**:
    - If ‘infra/‘ or ‘k8s/‘ changes: Trigger Infrastructure Deployment.
    - If ‘contracts/‘ changes: Trigger Contracts Build.
    - If ‘registry/‘ changes: Trigger Registry Build and Deploy.
    - If ‘service-folder/‘ changes: Mark that specific service for triggering.
4. **Execution**: The Master pipeline triggers the identified downstream jobs in parallel.

24.3 Detailed Analysis: Master Pipeline

The Jenkinsfile.master is the brain of the operation.

24.4 Key Stages

24.4.1 1. Detect Changes

This stage uses ‘git diff‘ to identify changed files.

```
script {
    def changedFiles = sh(script: "git diff --name-only HEAD~1 HEAD", returnStdout: true)
        .trim()
    // Logic to map folders to services
    if (file.startsWith('driver/')) env.CHANGED_SERVICES += 'driver-service'
    // ...
}
```

It sets environment variables like ‘CHANGED_SERVICES‘, ‘INFRA_CHANGED‘, etc., to control subsequent stages.

24.4.2 2. Build Contracts & Registry

These are shared dependencies.

- **Contracts**: If changed, builds Maven artifacts for shared data contracts used by other services.
- **Registry**: If changed, builds and pushes the Service Registry (Eureka) Docker image.

24.4.3 3. Deploy Infrastructure

Uses ‘kubectl‘ to deploy shared infrastructure components to Minikube.

- **Order of Operations**: Namespace ->Redis/Postgres/Zookeeper ->Wait for Zookeeper -¿ Kafka ->Elasticsearch ->Ingress.
- **Verification**: Uses ‘kubectl rollout status‘ to ensure pods are ready before proceeding.

24.4.4 4. Trigger Service Pipelines

This is the core ”Master” functionality. It iterates through the detected ‘CHANGED_SERVICES‘ and triggers them using the ‘build‘ step.

```
parallelBuilds[service] = {
    build job: service,
          parameters: [
              string(name: 'GIT_COMMIT', value: env.GIT_COMMIT),
              string(name: 'TRIGGERED_BY', value: 'master-orchestrator')
          ],
          wait: true
}
parallel parallelBuilds
```

This allows multiple microservices to be built and deployed simultaneously.

24.5 Detailed Analysis: Microservice Pipelines

Each microservice (Driver, Rider, User, Trip, Notification, Gateway, Matching, Frontend) has its own Jenkinsfile. They follow a standardized template.

24.6 Standard Pipeline Stages

24.6.1 1. Checkout

Checks out the specific commit passed from the Master pipeline (or HEAD if manual).

24.6.2 2. Build (Backend Services)

For Java/Spring Boot services (Driver, User, etc.):

```
sh "mvn -f service-name/pom.xml clean package -DskipTests"
```

This compiles the code and packages the JAR file.

24.6.3 3. Test

Runs unit tests to ensure code quality.

```
sh "mvn -f service-name/pom.xml test"
```

24.6.4 4. Docker Build & Push

Builds the Docker image using the service's ‘Dockerfile‘ and pushes it to DockerHub.

```
sh "docker build -f service-name/Dockerfile -t repo/image:tag ."
sh "docker push repo/image:tag"
```

24.6.5 5. Deploy to K8s

Updates the Kubernetes deployment with the new image.

- Uses ‘sed‘ to inject the specific Docker image tag into the YAML file.
- Applies the configuration using ‘kubectl apply -f‘.

24.6.6 6. Verify Deployment

Ensures the deployment rollout is successful.

```
sh "kubectl rollout status deployment/service-name -n metro --timeout=180s"
```

24.7 Service-Specific Differences

- **Frontend**: The Frontend pipeline skips the local ‘npm install‘ and ‘npm build‘ steps (commented out) and relies on the multi-stage Docker build process defined in its content to handle the application build.
- **Gateway**: Identical to backend services but deploys the API Gateway component.

24.8 Conclusion

The implemented Pipeline architecture provides a robust, scalable, and efficient CI/CD process. By decoupling the orchestration (Master) from the execution (Slave microservices), the system minimizes redundant builds and ensures that only changed components are redeployed, creating a rapid feedback loop for developers.

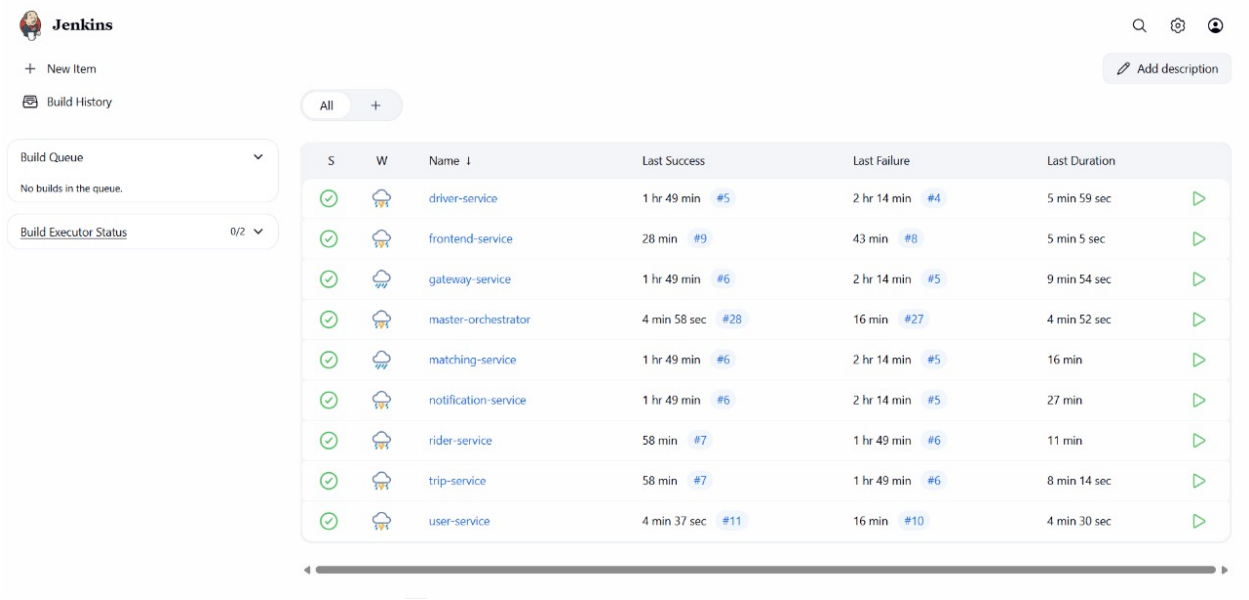


Figure 8: Builds of all components of the project

25 Jenkins Job Queue in Master Slave Architecture

25.1 Overview

In the Jenkins Master-Slave (Controller-Agent) architecture, the **Job Queue** is a fundamental component that manages the scheduling of build tasks. It acts as a buffer between incoming build requests and the available execution resources (agents/nodes).

25.2 Mechanism in Current Project

Based on the analysis of ‘Jenkinsfile.master‘, the Job Queue plays a critical role during the `Trigger Service Pipelines` stage.

1. **Orchestration**: The Master pipeline acts as an orchestrator.
2. **Parallel Triggers**: The code block below demonstrates how multiple service builds are scheduled simultaneously:

```
// Extract from Jenkinsfile.master
parallelBuilds[trimmedService] = {
    echo "Triggering ${trimmedService}..."
    build job: trimmedService,
          parameters: [...],
          wait: true,
          propagate: true
}
parallel parallelBuilds
```

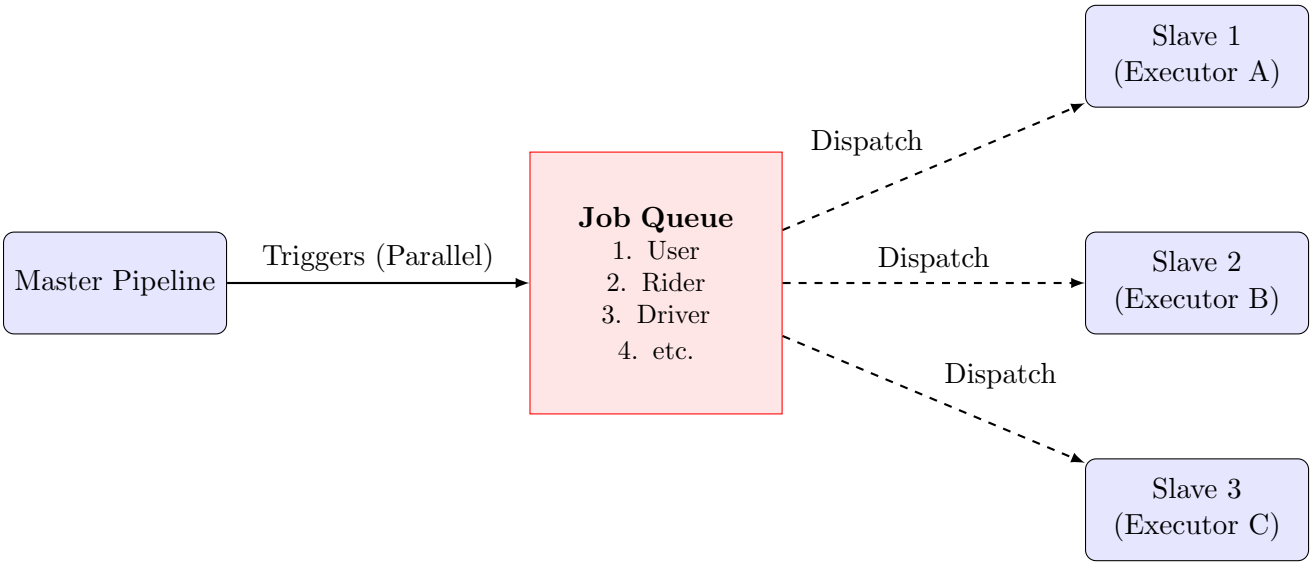
3. **Queueing**: When ‘parallel parallelBuilds‘ executes, Jenkins places a `Schedule` request for each service (User, Rider, etc.) into the Job Queue.

25.3 Technical Architecture

The scheduling process follows these technical steps:

- **Submission**: The ‘build‘ step invokes the ‘Queue.schedule()‘ method internally.
- **Queue Item**: A ‘Queue.WaitingItem‘ is created. It waits for the ‘quiet period‘ (if configured) to expire.
- **Blocked State**: The item may enter a ‘BlockedItem‘ state if the target job is already running and concurrent builds are disabled, or if dependencies are missing.
- **Buildable State**: Once ready, it becomes a ‘BuildableItem‘.
- **Dispatcher**: The ‘Queue.TaskDispatcher‘ runs strictly on the Master node. It iterates through ‘BuildableItems‘ and matches them against available **Executors** on Slave nodes.
- **Execution**: The Dispatcher assigns the task to a Slave’s workspace, creating a ‘Executable‘ (the actual build).

25.4 Visual Representation



25.5 Conclusion

The Job Queue ensures efficient resource utilization. In your project, even if you define ‘agent any‘, the generic agents (slaves) will pull tasks from this central queue as they become idle, allowing your microservices to build in parallel up to the limit of your available executors.

26 Kubernetes

26.1 Infrastructure Deployment Flow

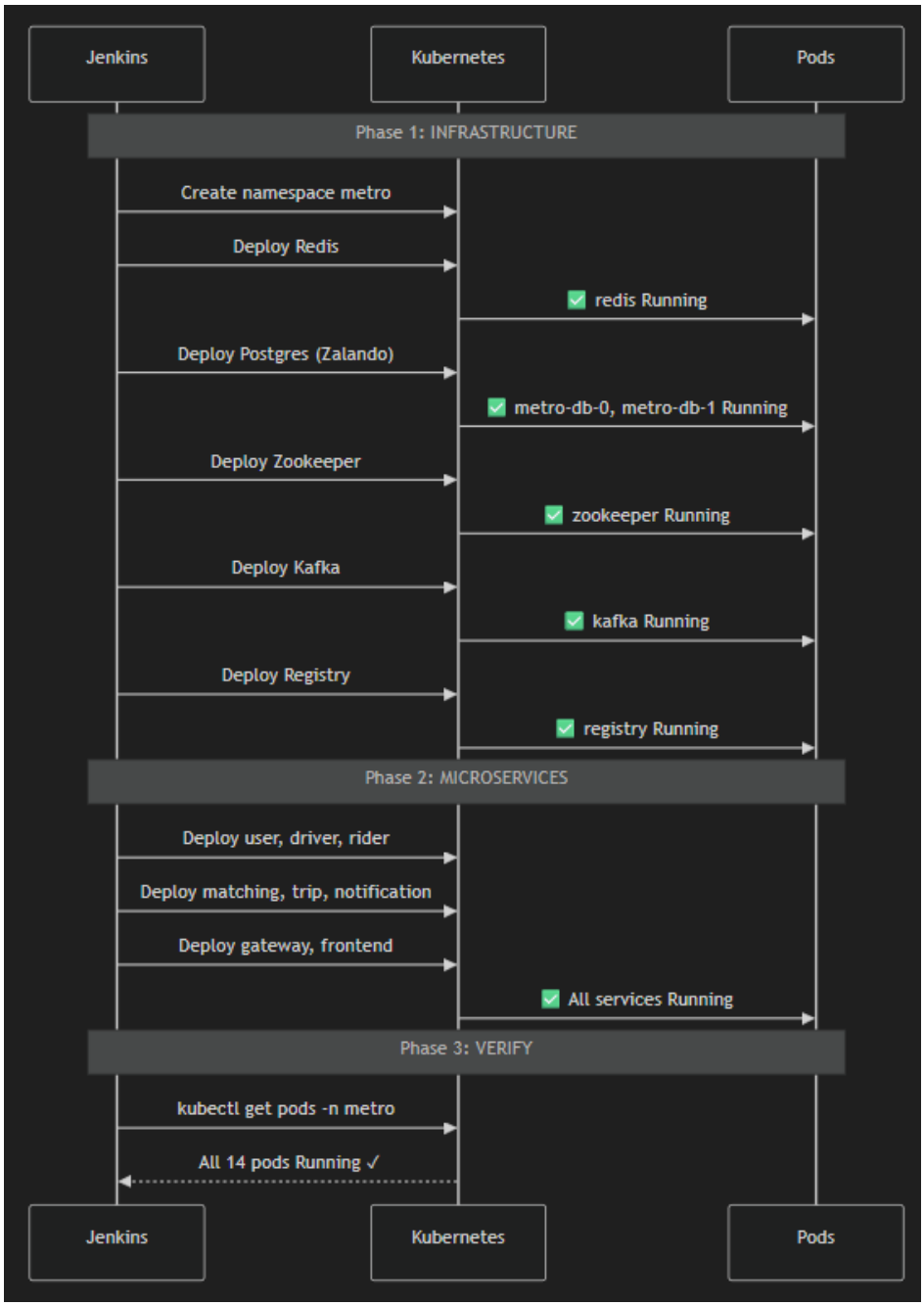


Figure 9: Infrastructure Deployment Design Using Jenkins and Kubernetes

26.2 Introduction

This document details the Kubernetes infrastructure used in the Metro Car Pooling project. The architecture leverages Minikube for local orchestration and integrates advanced operators for state-

ful database and event streaming services.

26.3 PostgreSQL Master-Slave Architecture

The project employs a robust high-availability architecture for the PostgreSQL database, managed by the **Zalando Postgres Operator**.

26.3.1 Architecture

The `postgres-operator` manages a PostgreSQL cluster named `acid-metro-cluster`.

- **Type**: Master-Slave Replication.
- **Instances**: 2 Pods (1 Leader, 1 Replica).
- **Failover**: Automatic failover managed by the operator via Patroni.

26.3.2 Configuration Details

The cluster is defined via a Custom Resource Definition (CRD) of kind `postgresql`.

- **API Version**: `acid.zalan.do/v1`
- **Volume Size**: 5Gi
- **Resources**:
  - Requests: 100m CPU, 100Mi Memory
  - Limits: 500m CPU, 500Mi Memory
- **Users**: `metrouser` (Superuser, Createdb rights)

26.4 Horizontal Pod Autoscaling (HPA)

Microservices are configured with Horizontal Pod Autoscalers to handle varying loads dynamically.

26.4.1 Policy

- **Metric**: CPU Utilization.
- **Target**: 70% Average utilization.
- **Scaling Range**: Minimum 1 replica, Maximum 5 replicas.

26.4.2 Covered Services

The following services have HPA enabled:

- `gateway-service`
- `frontend-service`
- `matching-service`
- `rider-service`

- `driver-service`
- `trip-service`

26.5 Service Deployments

The system architecture comprises several microservices and infrastructure components deployed as Kubernetes `Deployments` and exposed via `Services`.

26.5.1 Infrastructure Services

1. **Elasticsearch**:
   - **Image**: `docker.elastic.co/elasticsearch/elasticsearch:8.15.0`
   - **Configuration**: Single-node discovery type.
   - **Resources**: Limits set to 1000m CPU / 3Gi Memory.
2. **Kafka (Event Streaming)**:
   - **Operator**: **Strimzi Kafka Operator**.
   - **Mode**: KRaft (ZooKeeper-less/Raft) mode enabled via annotation `strimzi.io/kraft: enabled`.
   - **Version**: Kafka 3.7.0.
   - **Resources**: 1Gi-1.5Gi Memory for Kafka brokers.
3. **Redis**:
   - **Image**: `redis:7-alpine`.
   - **Role**: Caching and session management.
   - **Resources**: Lightweight footprint (128Mi Memory limit).
4. **Service Registry**:
   - **App**: Netflix Eureka (‘registry-service‘).
   - **Function**: Service discovery for microservices.
5. **Logging**:
   - **Logstash**: Deployed to aggregate logs and ship them to Elasticsearch.
   - **Config**: Uses a `ConfigMap` for pipeline definition.

26.5.2 Microservices

Core business services (User, Driver, Trip, Matching, etc.) follow a standard deployment pattern:

- **Replica Count**: Defaults to 1 (scaled by HPA).
- **Environment Variables**:
  - `SPRING_DATASOURCE_URL`: Points to the Postgres cluster service (`acid-metro-cluster`).
  - `SPRING_KAFKA_BOOTSTRAP_SERVERS`: Points to the Kafka bootstrap service (`metro-kafka-kafka-bootst`
  - `EUREKA_CLIENT_SERVICEURL_DEFAULTZONE`: Registry URL.
- **Secrets**: Database credentials injected via `secretKeyRef` created by the Postgres Operator.

26.6 Operators and Custom Resource Definitions (CRDs)

The project heavily utilizes the Operator pattern to manage complex stateful applications.

- **Strimzi**: Manages the Kafka cluster lifecycle. Defines CRDs such as `Kafka`, `KafkaTopic`, and `KafkaUser`.
- **Zalando Postgres Operator**: Manages High Availability (HA) PostgreSQL clusters. Defines CRDs such as `postgresql`.

```
➜  Metro-Car-Pooling git:(main) ✗ kubectl -n metro get pods
NAME                                      READY   STATUS      RESTARTS      AGE
acid-metro-cluster-0                      1/1     Running     1 (8h ago)    25h
acid-metro-cluster-1                      1/1     Running     1 (8h ago)    25h
driver-service-8879475d6-kkn48            1/1     Running     1 (8h ago)    25h
elasticsearch-79f8dd5dc4-hw4xz            1/1     Running     1 (8h ago)    25h
frontend-service-6b788676b-pxm8b          1/1     Running     1 (8h ago)    22h
gateway-service-6c64b6bf77-wjgjm          1/1     Running     0             8h
logstash-648787958c-kzlvr                 1/1     Running     1 (8h ago)    25h
matching-service-596c4987d5-29wl6         1/1     Running     1 (8h ago)    25h
metro-kafka-entity-operator-6b76d576d5-x98mk   1/1     Running     2 (8h ago)    25h
metro-kafka-metro-kafka-pool-0            1/1     Running     1 (8h ago)    25h
notification-service-56d6b9f86b-kwgqb     1/1     Running     1 (8h ago)    25h
postgres-operator-f4f4f789d-dxwkb         1/1     Running     2 (8h ago)    25h
redis-6c6fcd64b8-2kph8                    1/1     Running     1 (8h ago)    25h
redis-loader-qkk7z                        0/1     Completed   0             7h42m
registry-6fb87b6465-gbb88                 1/1     Running     1 (8h ago)    25h
rider-service-76fd55b4cb-fmqxv            1/1     Running     1 (8h ago)    25h
strimzi-cluster-operator-8457d7566-xvgx6  1/1     Running     1 (8h ago)    26h
trip-service-5f6d966d4f-wqjps             1/1     Running     1 (8h ago)    25h
user-79955dc64-c7slz                      1/1     Running     4 (8h ago)    25h
➜  Metro-Car-Pooling git:(main) ✗
```

Figure 10: Kubernetes All Pods

```
➜  Metro-Car-Pooling git:(main) ✗ kubectl -n metro get hpa
NAME           REFERENCE                      TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
driver-hpa     Deployment/driver-service      cpu: 7%/70%      1         5         1          6h19m
frontend-hpa   Deployment/frontend-service    cpu: 5%/70%      1         5         1          6h19m
gateway-hpa    Deployment/gateway-service     cpu: 5%/70%      1         5         1          6h19m
matching-hpa   Deployment/matching-service    cpu: 12%/70%     1         5         1          6h19m
rider-hpa      Deployment/rider-service       cpu: 5%/70%      1         5         1          6h19m
trip-hpa       Deployment/trip-service        cpu: 9%/70%      1         5         1          6h19m
➜  Metro-Car-Pooling git:(main) ✗
```

Figure 11: Horizontal Scaling

```
➜  Metro-Car-Pooling git:(main) ✗ minikube tunnel
✅  Tunnel successfully started

📌  NOTE: Please do not close this terminal as this process must stay alive for the tunnel to be accessible ...

🏃  Starting tunnel for service frontend-service.
🏃  Starting tunnel for service gateway-service.
❗  The service/ingress metro-ingress requires privileged ports to be exposed: [80 443]
🔑  sudo permission will be asked for it.
🏃  Starting tunnel for service metro-ingress.-
```

Figure 12: The tunneling which we did using Ingress for frontend and gateway service

```
jenkins@Vaibhav:/mnt/d/Desktop/SPE_Project$ kubectl get deployments -n  metro
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
driver-service        1/1     1            1           114m
elasticsearch         1/1     1            1           146m
frontend-service      1/1     1            1           35m
gateway-service       1/1     1            1           110m
kafka                 0/1     1            0           90s
matching-service      1/1     1            1           103m
notification-service  1/1     1            1           92m
postgres              1/1     1            1           147m
redis                 1/1     1            1           147m
registry              1/1     1            1           146m
rider-service         1/1     1            1           100m
trip-service          1/1     1            1           107m
user                  1/1     1            1           50m
zookeeper             1/1     1            1           147m
jenkins@Vaibhav:/mnt/d/Desktop/SPE_Project$ kubectl get pods -n metro
NAME                                  READY   STATUS             RESTARTS      AGE
driver-service-59d459bc8c-5hdwq       1/1     Running            0             114m
elasticsearch-56fbd9dfd4-gk8jw        1/1     Running            0             146m
frontend-service-7c8dd7b787-w5k6p     1/1     Running            0             35m
gateway-service-5cf888fbb8-zkrcv      1/1     Running            0             110m
kafka-5b6c4fbd58-gd4mq                0/1     CrashLoopBackOff   3 (34s ago)   97s
matching-service-56fc7c9df7-jnmkz     1/1     Running            0             103m
notification-service-76759db84b-bgm2p 1/1     Running            0             92m
postgres-56857c9bc5-lm5s8             1/1     Running            0             147m
redis-6c6fcd64b8-zff55                1/1     Running            0             147m
registry-5bf995bbdc-4flkr             1/1     Running            0             146m
rider-service-fdcb6c8c9-wh8hb         1/1     Running            0             100m
trip-service-7d7b5b4657-lz9g5         1/1     Running            0             107m
user-cbbfc6f66-v6jqb                  1/1     Running            0             11m
zookeeper-66b566b7dd-bjbt4            1/1     Running            0             147m
jenkins@Vaibhav:/mnt/d/Desktop/SPE_Project$ kubectl get svc -n metro
NAME                  TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)             AGE
driver-service        ClusterIP      10.99.157.126    <none>        8085/TCP,9090/TCP   114m
elasticsearch         ClusterIP      10.97.177.32     <none>        9200/TCP            147m
frontend-service      LoadBalancer   10.100.163.213   <pending>     3000:32028/TCP      35m
gateway-service       LoadBalancer   10.110.244.29    <pending>     8080:32074/TCP      110m
kafka                 ClusterIP      10.102.32.191    <none>        9092/TCP            147m
matching-service      ClusterIP      10.109.195.76    <none>        8082/TCP            104m
notification-service  ClusterIP      10.103.69.188    <none>        8087/TCP            93m
postgres              ClusterIP      10.96.198.222    <none>        5432/TCP            147m
redis                 ClusterIP      10.105.170.112   <none>        6379/TCP            147m
registry              ClusterIP      10.106.219.9     <none>        8761/TCP            147m
rider-service         ClusterIP      10.106.178.12    <none>        8083/TCP            100m
trip-service          ClusterIP      10.106.8.174     <none>        8086/TCP            107m
user                  ClusterIP      10.107.143.142   <none>        8081/TCP,9095/TCP   50m
zookeeper             ClusterIP      10.98.92.228     <none>        2181/TCP            147m
jenkins@Vaibhav:/mnt/d/Desktop/SPE_Project$
```

Figure 13: Output on the command prompt of the Kubernetes deployment

27 Leader Election in PostreSQL master slave architecture

27.1 Introduction

This document details the Leader Election mechanism employed by the **Zalando Postgres Operator** in the Metro Car Pooling project. The High Availability (HA) logic is delegated to **Patroni**, a template for PostgreSQL HA, which runs alongside PostgreSQL in each pod.

27.2 Core Components

27.2.1 1. Patroni

Patroni is the "brain" of the operation. It runs as a sidecar process (or entrypoint) inside every PostgreSQL Pod. It manages the PostgreSQL service state (start, stop, promote, demote).

27.2.2 2. Distributed Configuration Store (DCS)

Patroni relies on a DCS to maintain the cluster state and the "Leader Key". In a Kubernetes environment, the **Kubernetes API** itself acts as the DCS. Patroni uses Kubernetes `Endpoints` or `ConfigMaps` to store the cluster state.

- **Leader Key**: A specific resource (e.g., an Endpoint named `acid-metro-cluster-leader`) used to indicate acts as the Distributed Lock.

27.2.3 3. The Operator

The Zalando Operator is the controller. It deploys the StatefulSets and configures Patroni, but it does *not* participate in the second-by-second election loop. It watches the K8s resources to ensure the Service (`acid-metro-cluster`) always points to the current leader.

27.3 The Election Process (Internals)

27.3.1 Step 1: The Race for Leadership

When the cluster starts (or after a crash), all Patroni instances (running in pods) act as candidates.

1. Each Patroni instance attempts to create/update the **Leader Key** in the Kubernetes API.
2. This operation uses a Compare-And-Swap (CAS) atomic operation.
3. **Winner**: The instance that successfully writes its identity to the Leader Key becomes the **Master**.
4. **Losers**: All other instances detect that the key is already held. They become **Slaves** (Replicas) and start streaming data from the Master.

27.3.2 Step 2: Maintaining Leadership (Heartbeats)

- The Master must periodically renew the Leader Key (send a heartbeat) to indicate it is still alive.
- This is done by updating the `TTL` (Time-To-Live) on the Leader Key record.
- **TTL Duration**: Typically configured to a few seconds (e.g., 30s).

27.4 Failure Detection and Failover

27.4.1 How Slaves Detect Master Death

The Slave nodes (Patroni instances) continuously monitor the Leader Key in the DCS.

1. **Missing Heartbeat**: If the Master crashes or loses network connectivity, it fails to update the TTL.
2. **TTL Expiry**: The TTL on the Leader Key expires.
3. **Detection**: The Slaves observe that the Leader Key has expired and is effectively "unlocked".

27.4.2 The New Election

Once the lock is released:

1. All healthy Slaves immediately initiate a race to acquire the Leader Key.
2. The Slave with the most up-to-date transaction log (highest LSN - Log Sequence Number) is prioritized to prevent data loss.
3. The winner updates the key with its own identity.
4. **Promotion**: The winner works internally to promote its local PostgreSQL instance from Read-Only to Read-Write.
5. **Service Update**: Patroni (or the Operator) updates the Kubernetes Service (`acid-metro-cluster`) to route traffic to the IP of the new Master pod.

27.5 Summary flow

```
[Pod A (Master)] ---- updates TTL ---> [K8s API (Leader Key)]
                                                ^
                                                | monitors
[Pod B (Slave)] -----------------------------+
... CRASH (Pod A dies) ...
[K8s API] : TTL Expires -> Key Released
[Pod B] : "Key is free!" -> Acquires Key -> Promotes Self -> becomes Master
```

28 Concurrency Control and Database Locking

28.1 Introduction

This section explains the concurrency control mechanisms employed in the Metro Car Pooling project, specifically focusing on the interaction between the User microservice and the PostgreSQL database under high-concurrency write scenarios.

28.2 Architecture Overview

The system utilizes a Master-Slave architecture for PostgreSQL:

- **Write Operations**: Directed exclusively to the **Master** node.
- **Read Operations**: Load balanced between the Master and Slave replicas.
- **Scaling**: The User microservice is horizontally autoscaled, meaning multiple instances may attempt simultaneous writes.

28.3 Handling Simultaneous Writes

When multiple User microservice instances attempt to write to the database simultaneously, concurrency is managed primarily at the **Database Level**, as the application code does not implement explicit Optimistic Locking (no `@Version` annotation found) or Pessimistic Locking (no `@Lock` annotation found).

28.3.1 Mechanism: Write Serialization on Master

Since all write operations must go to the Master node, strict consistency is maintained by the database engine itself.

1. **Single Point of Truth**: Regardless of how many microservice instances are running, all `INSERT`, `UPDATE`, and `DELETE` commands are executed by the single Master PostgreSQL instance.
2. **ACID Properties**: PostgreSQL ensures Atomicity, Consistency, Isolation, and Durability for each transaction.

28.4 Locking Mechanism

28.4.1 Isolation Level

PostgreSQL uses **Read Committed** isolation level by default. This ensures that:

- A query sees only data committed before the query began.
- It prevents dirty reads but allows non-repeatable reads (which is generally acceptable for this use case).

28.4.2 Row-Level Locking (Pessimistic Locking)

PostgreSQL handles concurrent writes to the *same row* using **Row-Level Locks**. This is implicit behavior and does not require application-level annotations.

- **Scenario**: Two instances try to update the same user profile (e.g., `id=101`).
- **Process**:
    1. Transaction A acquires a Row-Level Lock on user `101`.
    2. Transaction B attempts to update user `101` but blocks (waits) because the lock is held by A.

3. Once Transaction A commits or rolls back, the lock is released.
4. Transaction B proceeds with the update.

- **Result**: Data integrity is preserved; the last committed transaction wins (Last-Write-Wins semantics).

28.4.3 Insert Concurrency

For new user registrations (`INSERT` operations):

- Concurrency is handled via **Unique Constraints** (e.g., on the `username` column).
- If two instances try to insert the same username simultaneously:
  - One transaction succeeds.
  - The other fails immediately with a uniqueness violation error (e.g., `DataIntegrityViolationException` in Spring Boot), effectively prevented by the unique index lock mechanism.

28.5 Conclusion

The system relies on PostgreSQL's robust internal locking mechanisms to handle concurrency.

- **Level of Locking**: **Row-Level** (writes to specific rows) and **Index-Level** (unique constraints).
- **Strategy**: Implicit Pessimistic Locking (database blocks conflicting writes).

29 Stress Testing: A way to test horizontal auto-scaling

29.1 Introduction to Autoscaling and Stress Testing

Autoscaling is a dynamic feature in cloud computing that automatically adjusts the allocation of computational resources (e.g., virtual machines, containers) based on current load requirements. Its primary purpose is to ensure performance and high availability during peak traffic while minimizing infrastructure costs during lulls.

Horizontal Pod Autoscaling (HPA), particularly within a Kubernetes environment, scales the application by changing the number of running pod replicas. Scaling decisions are typically driven by observed metrics, most commonly CPU utilization or memory consumption.

Stress Testing is a critical type of performance test. In the context of autoscaling, it is used to validate the resilience and responsiveness of the scaling mechanism under artificially induced, extreme load. The test confirms:

- **Scale Up:** The system can scale fast and far enough to handle a traffic surge.
- **Scale Down:** The system can correctly release unnecessary resources once the load subsides, optimizing costs.

29.2 The Stress Testing Procedure

The procedure outlined is a practical, command-line-driven method for observing the full scale cycle of a Kubernetes HPA.

29.2.1 Prerequisites

1. A functioning Kubernetes cluster.
2. An application (`frontend-service`) with an associated Horizontal Pod Autoscaler (`frontend-hpa`) deployed.
3. Access to two terminal windows for simultaneous execution and monitoring.

29.2.2 Step 1: Monitoring the Autoscaler (Terminal 1)

Monitoring is essential to observe the scaling logic in real-time.

```
kubectl -n metro get hpa -w
```

- **Command Function:** The `-w` (watch) flag keeps the output open, continuously displaying the status of HPA resources in the `metro` namespace.
- **Observed Metrics:** This output typically includes the current CPU utilization, the target utilization threshold, and the number of current and desired replicas.

29.2.3 Step 2: Generating Load (Terminal 2)

An aggressive load generator is deployed to create a resource-intensive traffic spike.

```
kubectl -n metro run -i --tty load-generator --rm --image=busybox:1.28 \
--restart=Never -- /bin/sh -c "while sleep 0.001; do \
wget -q -O- http://frontend-service:3000 > /dev/null; done"
```

- **Pod Creation:** A temporary pod named `load-generator` is created using the minimalist `busybox` image. The `--rm` flag ensures it is deleted after termination.
- **Load Mechanism:** The core is a `while` loop that executes a non-blocking `wget` command every 0.001 seconds. This floods the `frontend-service:3000` endpoint with requests, dramatically spiking the CPU usage of the target application's pods.

29.2.4 Expected Scaling Outcomes

The stress test is designed to demonstrate a complete scaling cycle:

1. **CPU Usage Spike (Scale Up Trigger):**
   - After $1 - 2$ minutes (allowing for metric aggregation delay), the HPA will report that the Current CPU usage has exceeded the Target (e.g., 250%/70%).
2. **Scale Up Execution (Replica Increase):**
   - The HPA controller calculates the required number of replicas N_{desired} to return to the target CPU utilization:

$$N_{\text{desired}} = \left\lceil \frac{\text{Current Metric Value}}{\text{Target Metric Value}} \times N_{\text{current}} \right\rceil$$

- The `REPLICAS` value in Terminal 1 will incrementally increase (e.g., $1 \to 2 \to \cdots \to 5$) until the load is stabilized or the maximum replica count is reached.

3. **Stop Load:**

   - The user terminates the load generation by pressing Ctrl + C in Terminal 2.

4. **Scale Down Execution (Replica Decrease):**

   - The CPU usage drops back to the baseline after the load stops.
   - After a mandatory "cooling down" or stabilization period (typically $\sim$ 5 minutes in default HPA settings), the HPA initiates a Scale Down event.
   - The REPLICAS will gradually drop back to the minimum configured replica count (e.g., 1).

29.3 Importance of Stress Testing Autoscaling

Stress testing the HPA is a critical step in a deployment lifecycle for several reasons:

Table 4: Key Benefits of Autoscaling Stress Testing

Aspect	**Rationale**
Validation of Scaling Logic	Confirms the HPA's ability to react promptly and correctly to fluctuating load, ensuring the system remains performant during unexpected spikes.
Identification of Bottlenecks	Reveals non-scaling limits, such as database connection pool exhaustion, file descriptor limits, or third-party service rate limits, which can be masked by sufficient replicas.
Cost Optimization	Proves the Scale Down mechanism's efficiency, preventing unnecessary cloud resource consumption from "zombie" or over-provisioned pods.
Tuning of Configuration	Provides empirical data to fine-tune critical HPA parameters, including the target metric threshold and the stabilization windows, for an optimal balance between responsiveness and cost.

30 Load Balancing over and Autoscaling of Microservices, Eureka Registry instances and API gateway instances put together + Synchronization required for the same

30.1 Introduction

The Metro Car Pooling architecture employs a multi-layered scaling and synchronization strategy. It uses **Kubernetes Ingress** for external load balancing, **Horizontal Pod Autoscaling (HPA)** for dynamic scaling of microservices and the API Gateway, and **Eureka Service Registry** for internal synchronization and discovery.

30.2 1. The Scaling Chain

30.2.1 Layer 1: Scaling the API Gateway

Traffic enters the cluster via the Ingress Controller (Nginx).

1. **Ingress Load Balancing**: The Ingress rule directs traffic to the `gateway-service`.
2. **Gateway Autoscaling**:
   - The API Gateway itself is stateless and autoscaled by HPA.
   - **Config**: Min 1, Max 5 replicas based on 70% CPU utilization.
3. **Synchronization**: When HPA scales up the Gateway (adds pods), Kubernetes automatically updates the `Endpoints` object for the `gateway-service`. The Nginx Ingress Controller watches these endpoints and reloads its configuration (or updates Lua tables) to round-robin traffic to the new Gateway instances immediately.

30.2.2 Layer 2: Scaling the Microservices

Behind the Gateway, microservices (User, Driver, etc.) are also autoscaled.

- **Trigger**: HPA monitors CPU usage. If load increases, new pods are spun up.
- **Registration**: When a new microservice pod starts, it registers itself with Eureka (The Registry) via a REST call, providing its IP and Port.
- **Discovery**: The Gateway (running a Eureka Client) periodically fetches the Registry (every 30s by default). It now knows the IPs of the new microservice instances.
- **Client-Side Load Balancing**: The Gateway uses Spring Cloud LoadBalancer to distribute requests among the available microservice instances.

30.3 2. Scaling the Registry

30.3.1 Current Architecture: Standalone

Currently, the Registry is deployed as a single replica (**Master/Slave is NOT used for simple Eureka setups**; Peer-to-Peer is the standard).

- **Config**: `register-with-eureka: false`, `fetch-registry: false`.
- **Limitation**: This is a Single Point of Failure (SPOF).

30.3.2 How to Scale (Peer-to-Peer)

To scale the Registry, one would use Eureka's Peer Awareness mode.

1. **Peer-to-Peer Mode**: Multiple Registry instances run.
2. **Sync**: Each registry instance registers with its "peers" (other registry nodes). They replicate the registration table among themselves.
3. **Client Config**: Microservices would be configured with a list of registry URLs (e.g., `http://registry-1:876`

30.4 3. Synchronization Flow Example

Scenario: Sudden traffic spike on the User Service.

1. **Traffic Ingress**: 1000 requests/sec hit Ingress → Distributed to 2 Gateway Pods.
2. **Gateway Limits**: Gateway CPU hits 75%. HPA detects this ¿ 70%.
3. **Gateway Scaling**: HPA scales Gateway to 3 Pods. Ingress detects the new IP and starts sending traffic to Gateway Pod #3.
4. **Service Overload**: User Service CPU hits 80
5. **Registration**: The 2 new User Pods start up. They send a `POST /eureka/apps/USER-SERVICE` to the Registry.
6. **Propagation**: The Registry adds these IPs to its list.
7. **Sync**: The 3 Gateway Pods (in background threads) fetch the updated registry delta.
8. **Routing**: The Gateways now round-robin traffic to all 4 User Pods.

All components remain synchronized because:

- **K8s Services** handle network reachability.
- **Eureka** handles application-level discovery.
- **HPA** handles capacity sizing.

31 Analysis: Ansible Overhead vs. Kubernetes Native Capabilities in the Metro Car Pooling Project

31.1 Introduction

This document analyzes the architectural decision to exclude Ansible from the *Metro Car Pooling* project tech stack. It details why introducing Ansible would result in unnecessary operational overhead and how Kubernetes (K8s) natively fulfills the roles traditionally assigned to Ansible in this specific context.

31.2 The Traditional Role of Ansible

Ansible is a powerful **Configuration Management** tool designed to:

- Provision infrastructure (servers, VMs).
- Configure operating systems (installing packages, patching, setting users).
- Deploy applications by copying artifacts and restarting services.
- Manage configuration files imperatively or declaratively on mutable infrastructure.

31.3 Why Ansible is Overhead in this Project

In the *Metro Car Pooling* project, the architecture relies on **Containerization (Docker)** and **Orchestration (Kubernetes)**. This shift in paradigm renders many of Ansible's core use cases redundant.

31.3.1 1. Mutable vs. Immutable Infrastructure

Ansible Approach: Ansible thrives on *mutable infrastructure*. It SSHs into existing servers to update libraries, change configurations, or deploy new code versions in place.
Project Reality: This project uses **immutable infrastructure**.

- Applications are packaged as Docker images.
- When a change is needed, a *new* image is built, and Kubernetes replaces the old container with a new one. All dependencies are baked into the image (via `Dockerfile`).
- **Result:** There is no "server state" for Ansible to manage. Using Ansible to patch a running container is an anti-pattern.

31.3.2 2. Declarative State Management

Ansible Approach: While Ansible can be declarative, it is often used procedurally (e.g., "apt-get install nginx", then "service nginx restart"). It runs as a "fire-and-forget" process. If a server drifts from the configuration *after* the playbook runs, Ansible won't know until it runs again.
Kubernetes Approach: Kubernetes uses a **continuous reconciliation loop**.

- You define the *desired state* (e.g., `replicas: 3`) in YAML manifests.
- The Kubernetes Control Plane constantly monitors the cluster. If a pod crashes, K8s restarts it immediately.
- **Result:** Kubernetes actively maintains the state 24/7. Introducing Ansible to apply YAMLs (via `kubectl apply`) adds an extra layer of abstraction without adding value, as Jenkins or ArgoCD can apply these manifests directly.

31.4 Kubernetes Replacements for Ansible Use Cases

The following table demonstrates how Kubernetes natively handles tasks that would otherwise require Ansible playbooks.

Task	Ansible (Traditional)	Kubernetes (Current)
App Deployment	Copying JARs/binaries to servers and restarting systemd services.	Updating the `Deployment` YAML with a new Docker image tag. K8s handles the rolling update.
Scaling	Provisioning new VMs and adding them to load balancer configs.	`HorizontalPodAutoscaler` (HPA) automatically scales pods based on CPU/Memory.
Self-Healing	Requires custom monit scripts or frequent playbook runs to check service health.	Native `livenessProbe` and `readinessProbe`. K8s automatically restarts failed instances.
Config Mgmt	Managing `/etc/config` files using templates (Jinja2).	`ConfigMaps` and `Secrets` injected directly into containers as environment variables.
Service Discovery	Updating HAProxy/Nginx config files when new backend servers are added.	`Services` (ClusterIP) and `Ingress` handle internal and external routing dynamically.

31.5 Specific Scenario: Why Ansible fails here

Consider the `gateway-service` autoscaling:

1. **Kubernetes:** The HPA detects high CPU usage and scales replicas from 2 to 5. The Service object automatically load balances traffic to these new pods immediately.
2. **Ansible:** Ansible is a "push-based" tool. It cannot react to real-time traffic spikes efficiently. To scale with Ansible, you would need an external trigger to run a playbook that provisions new VMs and updates LB configs, which is slow and brittle.

31.6 Conclusion

Integrating Ansible into the *Metro Car Pooling* project would introduce significant **technical debt** and **operational overhead**.

- **Redundancy:** Kubernetes already orchestrates the deployment, configuration, and lifecycle of the applications.
- **Complexity:** Managing both Ansible Playbooks (for potential VM setup) and Kubernetes Manifests (for app deployment) splits the source of truth.

Recommendation: Rely exclusively on **Kubernetes** for orchestration and application management, and use **Jenkins** pipelines to apply the Kubernetes manifests directly.

32 Why is autoscaling of PostgreSQL DB instances not recommended?

32.1 Introduction

In modern cloud-native environments, Horizontal Pod Autoscaling (HPA) is commonly used to dynamically scale stateless microservices based on load. However, applying the same autoscaling

principles to stateful databases such as PostgreSQL can lead to serious operational and consistency issues.

This report explains why autoscaling PostgreSQL read replicas using HPA is **not recommended**, particularly when managed by a Kubernetes operator such as the **Zalando PostgreSQL Operator**, and outlines safer alternatives.

32.2 Why Autoscaling PostgreSQL Replicas is Problematic

32.2.1 Operator Conflict

PostgreSQL clusters managed by the Zalando Operator rely on a **declarative desired state**. The number of database instances is defined explicitly using:

```
numberOfInstances: <N>
```

If an HPA attempts to scale the replica count dynamically:

- The HPA modifies the number of pods.
- The operator detects a deviation from the declared state.
- The operator reverts the replica count back to the configured value.

This creates a **control loop conflict**, causing:

- Pod churn
- Unstable clusters
- Repeated failovers or resync attempts

32.2.2 Data Replication Physics

Unlike stateless services, PostgreSQL replicas cannot become ready instantly.

When a new replica pod starts:

- It must clone the full database dataset (often multiple gigabytes).
- WAL synchronization must complete.
- Replication lag must reduce before it can serve reads.

For example, cloning a 5GB dataset can take **several minutes**. This means:

- Autoscaling does **not** react fast enough to traffic spikes.
- The new replica provides no immediate benefit.
- System load may worsen during clone operations.

32.2.3 Mismatch with HPA Design

HPA is designed for:

- Stateless workloads
- Fast startup times
- Horizontal scalability

PostgreSQL violates all three assumptions:

- Stateful data
- Slow startup and synchronization
- Strong consistency requirements

32.3 Recommended Alternatives

32.3.1 Static Scaling

If higher read throughput is anticipated, the safest approach is to **statically increase** the number of replicas:

```
numberOfInstances: 3
```

This ensures:

- Predictable cluster state
- No operator conflict
- Replicas are already warmed and available

32.3.2 Connection Pooling with PgBouncer

In many real-world cases, performance issues arise from excessive database connections rather than CPU or disk limits.

PgBouncer provides:

- Efficient connection reuse
- Reduced memory overhead on PostgreSQL
- Better handling of bursty traffic

This is often **more effective** than adding replicas.

32.3.3 Read Scaling Strategy

A recommended progression:

1. Enable PgBouncer
2. Tune application query patterns
3. Increase replica count manually if needed

32.4 Final Recommendation

- Do **not** use HPA to autoscale PostgreSQL replicas.
- Maintain **numberOfInstances: 2** (1 Primary, 1 Standby) for High Availability.
- Manually increase to 3 or more replicas only when sustained read load is expected.
- Prefer PgBouncer for handling connection spikes.

32.5 Conclusion

PostgreSQL scaling is fundamentally different from microservice scaling. Attempting to autoscale database replicas introduces instability without delivering real performance benefits.

A carefully planned, operator-aligned scaling strategy ensures:

- High availability
- Predictable performance
- Operational stability

33 Future Works and Shortcomings of the current design

33.1 Clean up of the buffer maintained by SSE

Currently, the cold SSE emits the latest event in the buffer so that a client, who might have accidentally lost connection, does not miss it. As we are storing the user ID in the local storage in the frontend and only filtering out the SSE notifications which are for that specific user ID, the moment the rider initializes the request for a ride, after the completion of their current ride, a match notification is shown but in reality, there is no match.

33.2 Streaming of notifications client wise

Currently, the stream between the backend and the frontend (through the API gateway) sends all the notifications to the frontend in the same stream. However, the design should be such that there should be separate streams for separate users. Or we could have one connection that will persist but we will have to make separate channels inside the connection for separate users. Basically, we need to use the several-lanes-in-a-highway analogy.

33.3 Clean up of rider waiting queue

Currently, the riders are being put back into the waiting queue cache if a driver is not found to match them. However, this is happening infinitely. Thus, we need to have a separate field for each rider in the waiting queue cache that stores when the rider was put into the queue. Then we need to decide the time period after which if we are not able to find a match with a driver, the rider must not be put back into the queue. The rider should be notified that a driver was not found. They may choose to search for a driver again. Thus, they will be put into the queue again and the process will continue.

33.4 Setting up more slave DB instances

Currently we have 1 master + 1 slave in the PostgreSQL DB architecture. We can add more slave DB instances. Basically, we can have 1:N master slave architecture. This will further help when the reads are load balanced over the N + 1 DB instances.