International Institute of Information Technology
Bangalore

LastMile Project Report

Team Members:

Abhinav Kumar (IMT2022079)
Siddhesh Deshpande (IMT2022080)

December 12, 2025

Contents

Introduction

LastMile is a microservice-based ride-matching application designed to provide seamless last-mile connectivity for metro commuters. The system allows riders to hire short-distance drop services from metro stations to nearby destinations, while drivers publish their routes, available seats, and the stations they pass through. As drivers periodically update their location, the system detects when they approach a metro station and triggers the rider–driver matching process. A match is made only when the rider and driver share the same destination area and seats are available.

The application is decomposed into multiple independent microservices, each responsible for a specific function in the workflow:

- **User Service** – Manages user profiles for riders and drivers, and handles authentication.
- **Driver Service** – Stores driver routes, metro stations covered, seat availability, and real-time updates.
- **Rider Service** – Registers rider arrival times and destinations, and tracks ride status.
- **Matching Service** – Performs rider–driver matching based on time, proximity, and destination.
- **Trip Service** – Manages the lifecycle of trips from scheduled to completed.
- **Notification Service** – Sends real-time alerts and updates to riders and drivers.
- **Location Service** – Determines whether a driver is close to any metro station and provides proximity results to the Matching Service
- **Station Service** – Maintains metro station metadata and maps stations to nearby areas.

These microservices communicate using Google RPC (gRPC) for direct requests and use Kafka for asynchronous events streaming (such as notification updates and match triggers). The entire system is deployed on Kubernetes to support scaling, resilience, and fault tolerance.

System Architecture Diagram

The following figure presents the overall system architecture of the LastMile application, illustrating the interaction between microservices, the message broker (Kafka), and external clients:

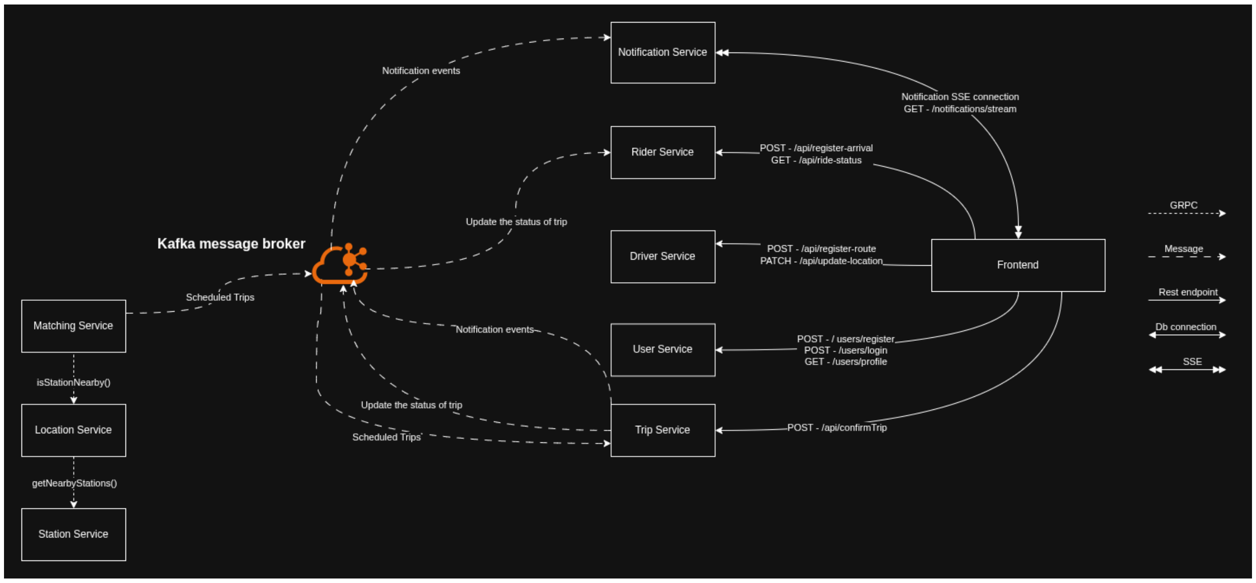


Figure 1: Lastmile application architecture

The above diagram[1] represents the complete architecture of the LastMile application. It shows all microservices, their communication patterns, and the central Kafka message broker used for asynchronous event streaming. The diagram highlights:

- **REST** communication between the frontend and core services (User, Driver, Rider, Trip).
- **SSE** channel from the Notification Service for real-time updates.
- **gRPC** communication for internal service-to-service requests (e.g., Matching → Location → Station).
- **Kafka** topics used for trip lifecycle, notification delivery and trip scheduling.
- How each microservice contributes to the rider–driver matching workflow

Microservice

Microservices architecture is a design approach where an application is broken into small, independent services, each responsible for a specific function. These services can be developed, deployed, and scaled separately, often using different technologies, and they communicate through lightweight APIs or messaging. This modular structure enables faster development, better fault isolation, and easier maintenance compared to monolithic systems, though it also introduces challenges such as increased operational complexity and the need for robust service coordination.

User Service

Overview

The User Service is responsible for managing all user identities within the LastMile system, including both riders and drivers. It centralizes authentication, authorization, and profile management. The service issues JWT tokens that are used by other microservices for identity verification and access control.

Responsibilities

- Manages rider and driver profile records.
- Performs authentication using username and password.
- Handles role-based authorization for multi-role users.
- Issues JWT tokens containing identity and role information.
- Validates credentials and enforces role selection during login.

Architecture

The service is structured around a layered approach. A persistence layer stores user profiles, including usernames, encrypted passwords, and assigned roles. A security layer integrates with an authentication manager and a JWT provider to verify credentials and generate tokens. The controller layer exposes REST endpoints for registration, login, and authenticated profile retrieval. Role checks are performed during login to ensure the user selects a valid role assigned to them.

Data and Interactions

User records include a username, a hashed password, and an array of associated roles. The service communicates with a relational database for storage and retrieval of user data. Upon successful authentication, a JWT containing the user ID, selected role, and username is returned. Other microservices consume this token to validate permissions without calling the User Service again, enabling distributed authorization. The service interacts only through synchronous HTTP calls and does not publish or consume events.

Endpoints

- **POST /users/register** - Registers a new user, checks for existing usernames, encodes the password, and stores roles.
- **POST /users/login** - Validates the user, verifies the selected role, authenticates credentials, and returns a JWT.
- **GET /users/profile** - Returns authenticated user details based on the JWT provided in the request.

Driver Service

Overview

The Driver Service manages driver operations within the LastMile system. It allows drivers to register their routes, update real-time locations, and manage available seats. The service integrates with both a relational database for persistence and Redis for fast access to driver state.

Responsibilities

- Allows drivers to register routes with starting location, destination, and vehicle information.
- Tracks real-time driver location updates for active routes.
- Manages available seats for each route.
- Maintains driver-related data in both persistent storage and fast-access caches.
- Supports downstream services in locating nearby drivers for matching purposes.

Architecture

The service consists of a controller layer exposing REST endpoints, a persistence layer storing route data, and a caching layer using Redis for quick access to driver locations and route metadata. Authentication information is derived from the security context, ensuring only valid drivers can update their routes or locations.

Data and Interactions

Route data includes driver ID, starting location, destination, vehicle number, and available seats. Redis stores real-time location and route metadata for efficient retrieval by other services and these routes are also stored in database for storing route Details.

Endpoints

- **POST /api/register-route** - Registers a new driver route, persists it in the database, and updates Redis with route and seat information.
- **PATCH /api/update-location** - Updates the driver's current location for an active route in both Redis and the database.

Rider Service

Overview

The Rider Service manages all rider-side operations within the LastMile ecosystem. It handles arrival registrations at metro stations, stores user-selected destinations, and tracks ride status throughout the entire trip lifecycle. The service integrates with both persistent storage and Redis for matching active ride requests with respective drivers by matching service.

Responsibilities

- Allows riders to register their upcoming metro arrival time, station name, and destination.
- Stores rider arrival data in both the database and Redis to support fast lookups.
- Tracks ride status from the moment a rider registers until the ride is completed.
- Listens to Kafka events to update ride status (e.g., matched, en-route, completed).
- Provides APIs for riders to query current trip status.

Architecture

The Rider Service exposes REST endpoints for arrival registration and status retrieval. It uses a relational database to persist ride entries and Redis to cache arrival data keyed per rider for fast access during matching. Kafka consumer handlers update ride status asynchronously based on events published by other microservices. Authentication details are extracted from the security context to associate actions with the correct rider.

Data and Interactions

A ride entry contains information such as rider ID, arrival time, station, destination, and current status. Redis holds a compact representation of arrival metadata for efficient consumption by matching logic. Kafka topics provide asynchronous updates such as trip completion or status transitions. The service interacts primarily with the Trip Service with event-driven flows through Kafka to update the current status of the rides.

Endpoints

- **POST /api/register-arrival** - Registers rider arrival time and destination, stores the information in the database, and caches it in Redis for quick matching.
- **GET /api/ride-status** - Returns the current status of a rider's trip based on a ride identifier.

Matching Service

Overview

The Matching Service is responsible for pairing riders with suitable drivers based on proximity, timing, and destination compatibility. It operates as a periodic background scheduler that evaluates all active rider and driver entries from Redis and performs matching decisions. Once a successful match is found, the service triggers downstream workflows through event publication.

Responsibilities

- Continuously scans cached rider and driver data to identify compatible pairs.
- Ensures drivers have available seats and that their destinations align with rider requests.
- Verifies proximity between rider arrival stations and driver locations using a gRPC-based location service.
- Publishes match events to the trip management workflow for further processing.
- Removes matched riders from Redis and updates driver availability for future matches.
- Coordinates concurrent execution of multiple instances of this service (in case of scale up) by using redis locks so that there is no race condition for same keys by multiple instances of the service.

Architecture

The Matching Service runs as a scheduled component within the system. It retrieves real-time data from Redis, applies matching rules, and communicates with external systems through Kafka and gRPC. To maintain correctness in distributed deployments, it uses a Redis-backed lock manager that ensures only one instance executes the matching loop at a given time. The service integrates with the Location Service via a gRPC client to assess proximity between driver locations and metro stations.

Data and Interactions

Driver data stored in Redis includes current location, available seats, destination, and vehicle information. Rider data contains arrival times, station names, and trip destinations. The matching algorithm considers temporal windows, destination equality, seat availability, and spatial proximity. Upon a match, a unified event containing driver and rider identifiers is sent to downstream Kafka topics. Redis entries are updated accordingly to maintain consistency for subsequent matching rounds.

Endpoints

- The Matching Service does not expose HTTP endpoints; it operates as a periodic scheduler.
- Interacts with the system through:
  - **Redis** for retrieving rider and driver entries.
  - **gRPC** to validate metro-station proximity.
  - **Kafka** to publish match events to the trip workflow.

Trip Service

Overview

The Trip Service manages the lifecycle of every ride after a rider–driver match is created. It handles trip creation, state transitions, driver arrival detection, and trip completion while coordinating updates with other services through event-driven communication.

Responsibilities

- Create new trips based on matching events.
- Maintain trip statuses: SCHEDULED, ACTIVE, COMPLETED.
- Detect driver arrival at pickup stations and trigger notifications.
- Detect destination arrival and mark trips as completed.
- Publish status updates to Rider Service and notification events to Notification Service.

Architecture

The service follows an event-driven structure. Incoming matching events create and initialize trip records. A scheduled background process periodically reads driver location data from Redis to determine pickup and drop-off events. The service updates trip state in the database and emits corresponding events to maintain consistency across the ecosystem. REST endpoints allow riders to confirm trips, moving them from scheduled to active.

Data and Interactions

The service stores trip records in a relational database. It reads real-time driver locations from Redis, updated externally. It consumes events from the `trip-service` Kafka topic and publishes updates to `rider-service` and `notification-service`. Communication with clients happens over HTTP for actions such as confirming trips.

Endpoints

- **POST /api/confirmTrip** - Confirms a scheduled trip and updates its status to ACTIVE. Also triggers rider status updates and notification events.

Notification Service

Overview

The Notification Service delivers real-time updates to riders and drivers. It uses Server-Sent Events (SSE) for persistent streaming connections and listens to domain events published across the system to notify users about trip-related changes.

Responsibilities

- Maintain SSE connections for riders and drivers.
- Stream real-time notifications such as matching events, driver arrival, trip start, and trip completion.
- Consume notification-related events from Kafka and route them to the appropriate user.
- Handle emitter lifecycle (disconnects, errors, timeouts).

Architecture

The service manages a set of active SSE emitters, separated for riders and drivers. It consumes multiple event types from the `notification-service` Kafka topic and formats messages before streaming them to connected clients. A REST endpoint establishes SSE connections, validating identity through JWT before registering the emitter. The service operates in an event-driven manner, with each Kafka message translated into an immediate push notification.

Data and Interactions

The service does not maintain persistent user data. Instead, it keeps in-memory emitter maps for active sessions. It consumes events such as matching notifications, driver arrival updates, trip start confirmations, and destination arrival signals. It interacts with the Authentication component to extract user identity from tokens and exposes one streaming endpoint for SSE communication.

Endpoints

- **GET /notifications/stream** - Establishes an SSE connection for the authenticated rider or driver. The service streams real-time trip and status updates to the client.

Location Service

Overview

The Location Service provides proximity checking between a driver's current location and a metro station. It enables the Matching Service to validate whether a driver is close enough to a rider's arrival station before creating a match.

Responsibilities

- Perform proximity validation between driver location and a target station.
- Communicate with the Station service to retrieve nearby areas.
- Provide a simple gRPC interface for other services (mainly Matching Service).

Architecture

The service exposes a gRPC endpoint that receives station and driver location information. It internally calls a Station Service client stub to fetch a list of predefined nearby areas for a station. The logic checks if the driver's current location appears in the station's nearby area list and returns a boolean response.

Data and Interactions

The service does not store data-even proximity checks are stateless. It interacts with:

- **Station Service (gRPC)** - to fetch nearby station areas.
- **Matching Service** - which invokes this service for location-based filtering.

Endpoints

- **gRPC: checkIfNearby(stationName, driverLocation)** - Returns whether the driver is within the valid proximity range of the given station.

Station Service

Overview

The Station Service maintains metadata for metro stations and provides information about areas that are considered nearby to each station. This data is used by the Location Service and Matching Service to determine proximity between riders, drivers, and metro stations.

Responsibilities

- Store and manage nearby-area mappings for each metro station.
- Provide a gRPC interface for retrieving nearby places.
- Act as a source of truth for station-level spatial metadata.

Architecture

The service exposes a gRPC endpoint. Upon receiving a request for a station, it fetches a list of nearby areas from Redis and returns it as a gRPC response. It does not perform computations-its primary role is serving station metadata efficiently.

Data and Interactions

- **Redis** - Stores lists of nearby areas keyed by station name.
- **Location Service (gRPC client)** - Consumes the nearby-area information to evaluate proximity.
- **Matching Service (indirectly)** - Uses results from the Location Service's proximity checks.

Endpoints

- **gRPC: getNearbystations(stationName)** - Returns the list of areas considered near the specified station.

Containerization Using Distroless Images

All microservices are containerized using Distroless images, specifically `gcr.io/distroless/java21-debian12:nonroot`, which provide a minimal and secure runtime environment for Java applications.

How Distroless Works

Distroless images contain only the essential runtime dependencies required to execute the application. They intentionally exclude components such as:

- Shells (e.g., `/bin/bash`)
- Package managers (e.g., `apt`, `yum`)
- System utilities and debuggers

Because of this, the container attack surface is significantly reduced, and the image size becomes much smaller.

Build Mechanism

Each microservice follows a simple and consistent mechanism:

1. A Spring Boot JAR is generated using Maven.
2. The Dockerfile copies the JAR into the image:
   `COPY ./target/app.jar /target.jar`
3. The service is executed using the non-root Java runtime provided by Distroless:
   `CMD ["/target.jar"]`

Non-Root Execution

The `nonroot` variant enforces that all processes run without root privileges. This prevents privilege escalation and aligns with Kubernetes best practices for secure workloads.

Benefits

- Smaller image footprint compared to traditional Ubuntu or Alpine-based images.
- Improved security due to lack of interactive shells and system packages.
- Faster startup times and reduced attack vectors.
- Consistent runtime across all microservices.

Using Docker Compose for Image Builds

Docker Compose is used solely during the CI/CD pipeline to build and push Docker images for all microservices. Each microservice has its own build context and Dockerfile, ensuring isolated and consistent builds. Compose also injects necessary environment variables at build time, such as database URLs, Redis configuration, Kafka brokers, and log paths.

Since deployment is handled by Kubernetes, Docker Compose is **not** used for running or orchestrating containers in any environment. Its purpose is limited to providing a reproducible and automated multi-service build setup before the images are pushed to Docker Hub as part of the Jenkins pipeline.

Kubernetes Setup for Microservices

- **Deployments**
  - Each microservice is deployed as an independent Kubernetes Deployment.
  - Specifies the container image, exposed ports, environment variables, and runtime configs.
  - Uses Secrets and ConfigMaps for securely managing credentials and environment settings.
  - Resource requests and limits ensure predictable scheduling and autoscaler compatibility.
  - Label-based selectors maintain replica consistency and high availability.
- **Horizontal Pod Autoscalers (HPAs)**

  - All microservices use an HPA for automatic scaling based on workload.
  - Scales replicas within defined min–max limits depending on CPU utilization.
  - Stabilization windows prevent rapid downscaling oscillations.
  - Helps maintain performance under heavy load while optimizing resource usage during low load.

- **Services**
  - Each microservice is exposed internally through a ClusterIP Service.
  - Provides a stable virtual IP and built-in load balancing across pods.
  - Traffic routing is handled using label selectors pointing to the correct Deployment.

- **Ingress (Networking)**
  - An Ingress resource is used to route external HTTP/HTTPS traffic into the cluster.
  - Acts as a single entry point for all microservices, simplifying access management.
  - Allows path-based routing to individual services without exposing them externally.
  - Reduces the need for multiple LoadBalancers and improves network efficiency.

- **Credential and Configuration Management**
  - Kubernetes Secrets store sensitive information such as database usernames, passwords, and tokens.
  - Secrets are mounted or injected as environment variables, keeping credentials out of code.
  - ConfigMaps provide non-sensitive runtime configuration shared across services.
  - This separation ensures cleaner deployments and secure credential handling.

- **Overall Architecture Advantages**
  - Standardized Kubernetes patterns ensure consistent deployment across all microservices.
  - Enables independent scaling, updates, and fault isolation per service.
  - Centralized networking, discovery, and routing simplify inter-service communication.
  - Supports a scalable, secure, and maintainable microservices ecosystem.

Stateful Component Deployment

- **Use of StatefulSets**
  - All stateful components (Kafka, Redis, PostgreSQL and Zookeeper) are deployed using Kubernetes StatefulSet.
  - StatefulSets provide stable network identities, persistent storage, and ordered pod management, which are essential for reliable stateful workloads.
  - Each StatefulSet is paired with a headless service (`clusterIP: None`) to ensure consistent DNS names for the underlying pods.

- **Kafka Deployment**

  - Kafka runs as a single-broker StatefulSet with persistent storage for message logs.
  - Uses a headless service for stable broker identity and a ClusterIP service for internal client communication.
  - Configured with Zookeeper connection, advertised listeners, and replication factors suitable for single-node operation.
  - PersistentVolumeClaims store Kafka logs at `/var/lib/kafka/data`.

- **Zookeeper Deployment**
  - Zookeeper is deployed as a StatefulSet to maintain consistent server IDs and stable storage.
  - Uses a headless service to provide a predictable DNS address for Kafka.
  - Stores data in persistent volumes mounted at `/var/lib/zookeeper/data`.

- **PostgreSQL Deployment**
  - PostgreSQL is deployed as a StatefulSet with persistent volumes for durable database storage.
  - Credentials are securely injected using Kubernetes Secrets.
  - Initialization scripts are mounted through ConfigMaps into `/docker-entrypoint-initdb.d`.
  - Exposed internally through both a headless service (for StatefulSet identity) and a ClusterIP service (for microservice access).

- **Redis Deployment**
  - Redis runs as a StatefulSet with Append-Only File (AOF) persistence enabled for durability.
  - A headless service ensures stable DNS names for the Redis pod, while a ClusterIP service exposes port 6379 to other microservices.
  - Uses a persistent volume for storage under `/data`.
  - Includes a postStart hook to pre-populate station metadata into Redis after startup.

- **Benefits of StatefulSet Architecture**
  - Guarantees stable storage and identity for databases, brokers, and message systems.
  - Ensures clean restart behavior and predictable pod ordering.
  - Provides reliable persistence for components where data consistency and recovery are critical.
  - Simplifies internal discovery and networking for stateful systems.

Centralized Logging with ELK Stack

- **Filebeat DaemonSet**
  - Filebeat is deployed as a DaemonSet so that an agent runs on every Kubernetes node.
  - Collects container logs from `/var/log/containers` and enriches them with Kubernetes metadata.

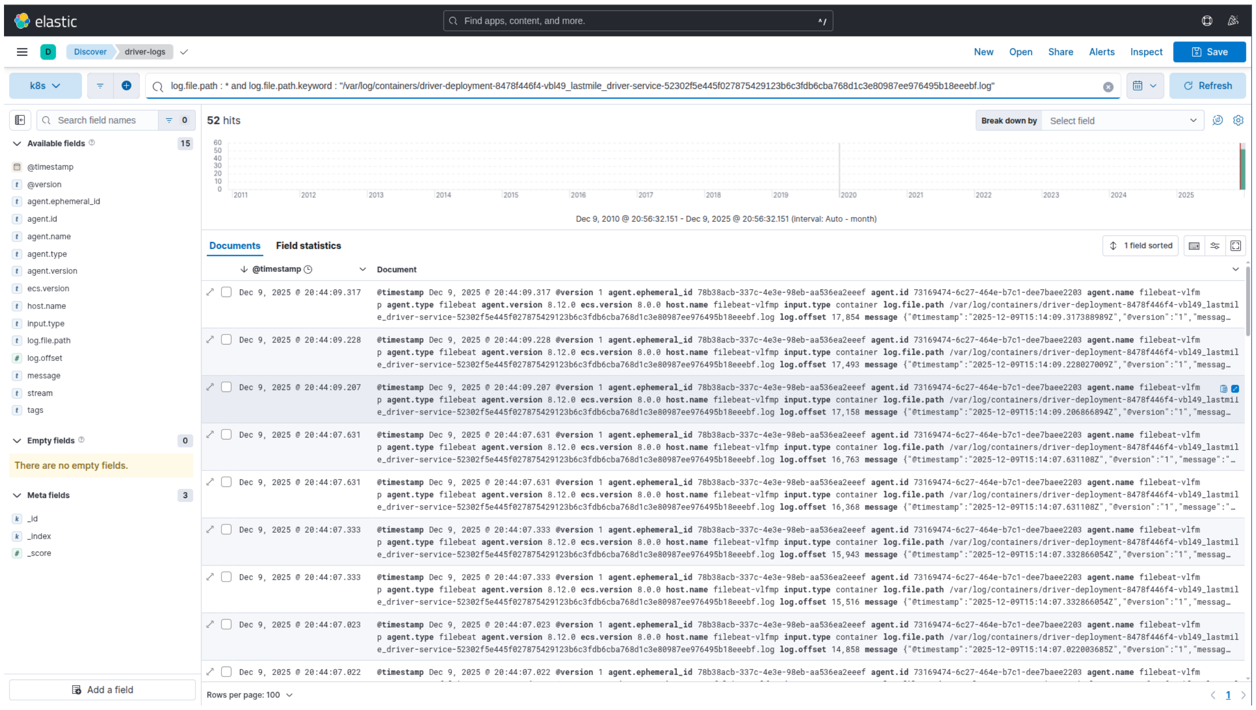
  - Uses RBAC (ServiceAccount, ClusterRole, ClusterRoleBinding) to access pod, node, and namespace information.
  - A ConfigMap provides the Filebeat configuration, specifying Logstash as the output endpoint.
  - HostPath volumes allow Filebeat to read logs directly from the node filesystem, ensuring complete log capture.

- **Logstash Deployment**
  - Logstash runs as a Deployment and receives log data from Filebeat over port 5044.
  - Its configuration (pipelines and main.conf) is injected using a ConfigMap.
  - Processes incoming logs and routes them to Elasticsearch using a daily index pattern.
  - A ClusterIP service exposes Logstash internally to the Filebeat agents.

- **Elasticsearch Integration**
  - Logstash forwards logs to Elasticsearch via the internal Kubernetes service `elasticsearch-service`.
  - Elasticsearch serves as the central storage and indexing engine for cluster logs.
  - All logs are stored in indices named `k8s-logs-YYYY.MM.dd`.

- **Kibana Deployment**
  - Kibana runs as a Deployment and connects to Elasticsearch for log visualization.
  - Environment variables configure the Elasticsearch host and disable security for local development.
  - Exposed externally using a NodePort service for easy dashboard access.
  - Provides a graphical interface[2] to query, filter, and analyze logs from all microservices.

- **Logging Architecture Advantages**
  - Ensures unified, cluster-wide logging for all microservices.
  - Filebeat + Logstash enables reliable log collection, transformation, and routing.
  - Elasticsearch stores logs with powerful indexing and search capabilities.
  - Kibana provides real-time insights, monitoring, and debugging support.



```
log.file.path : * and log.file.path.keyword : "/var/log/containers/driver-deployment-8478f446f4-vbl49_lastmile_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log"
```

```
Dec 9, 2010 @ 20:56:32.151 - Dec 9, 2025 @ 20:56:32.151 (interval: Auto - month)
```

```
Dec 9, 2025 @ 20:44:09.317   @timestamp Dec 9, 2025 @ 20:44:09.317 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 17,854 message {"@timestamp":"2025-12-09T15:14:09.317388989Z","@version":"1","messag…
Dec 9, 2025 @ 20:44:09.228   @timestamp Dec 9, 2025 @ 20:44:09.228 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 17,493 message {"@timestamp":"2025-12-09T15:14:09.228027009Z","@version":"1","messag…
Dec 9, 2025 @ 20:44:09.207   @timestamp Dec 9, 2025 @ 20:44:09.207 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 17,158 message {"@timestamp":"2025-12-09T15:14:09.206866894Z","@version":"1","messag…
Dec 9, 2025 @ 20:44:07.631   @timestamp Dec 9, 2025 @ 20:44:07.631 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 16,763 message {"@timestamp":"2025-12-09T15:14:07.631108Z","@version":"1","message":"…
Dec 9, 2025 @ 20:44:07.631   @timestamp Dec 9, 2025 @ 20:44:07.631 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 16,368 message {"@timestamp":"2025-12-09T15:14:07.631108Z","@version":"1","message":"…
Dec 9, 2025 @ 20:44:07.333   @timestamp Dec 9, 2025 @ 20:44:07.333 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 15,943 message {"@timestamp":"2025-12-09T15:14:07.332866054Z","@version":"1","messag…
Dec 9, 2025 @ 20:44:07.333   @timestamp Dec 9, 2025 @ 20:44:07.333 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 15,516 message {"@timestamp":"2025-12-09T15:14:07.332866054Z","@version":"1","messag…
Dec 9, 2025 @ 20:44:07.023   @timestamp Dec 9, 2025 @ 20:44:07.023 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
                             p agent.type filebeat agent.version 8.12.0 ecs.version 8.0.0 host.name filebeat-vlfmp input.type container log.file.path /var/log/containers/driver-deployment-8478f446f4-vbl49_lastmil
                             e_driver-service-52302f5e445f027875429123b6c3fdb6cba768d1c3e80987ee976495b18eeebf.log log.offset 14,858 message {"@timestamp":"2025-12-09T15:14:07.022003685Z","@version":"1","messag…
Dec 9, 2025 @ 20:44:07.022   @timestamp Dec 9, 2025 @ 20:44:07.022 @version 1 agent.ephemeral_id 78b38acb-337c-4e3e-98eb-aa536ea2eeef agent.id 73169474-6c27-464e-b7c1-dee7baee2203 agent.name filebeat-vlfm
```

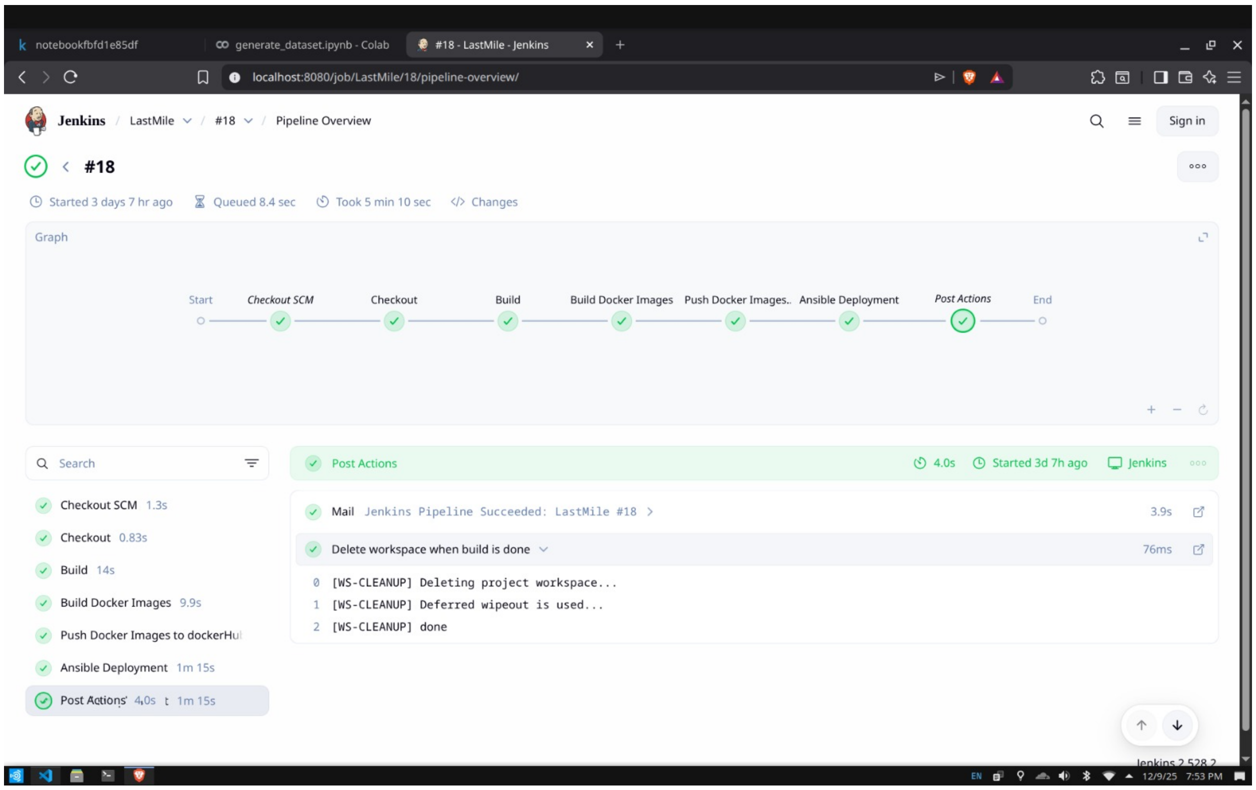
Figure 2: Kibana Dashboard

Jenkins Pipeline

The project uses a Jenkins declarative pipeline to automate the complete build and deployment process. The pipeline is triggered on every GitHub push and performs source checkout, application build, container image creation, and remote deployment using Ansible. Jenkins also sends email notifications for both successful and failed runs.

- **Source Code Checkout**
  - The pipeline automatically pulls the latest code from the `main` branch of the GitHub repository using the `git` step.
- **Build Stage**
  - Maven is executed inside the `backend-services` directory to compile the microservices and generate JAR artifacts.
  - Unit tests are skipped for faster CI, using the `-DskipTests` flag.
- **Docker Image Build and Push**
  - Docker images for all microservices are built using `docker compose build`.
  - Jenkins pulls Docker Hub credentials stored securely in the Jenkins Credentials Store (ID: `DockerHubCred`).
  - Credential injection occurs inside a `docker.withRegistry` block, ensuring passwords are never exposed in logs.
  - Images are pushed to Docker Hub with `docker compose push`.

- **Secrets Management and Environment Variables**
  - Sensitive values such as Docker Hub username/password are stored as Jenkins credentials.
  - These credentials are referenced in the pipeline without hardcoding, ensuring secure authentication.
  - Non-sensitive configuration such as notification emails are stored as environment variables using the `environment` block.
- **Ansible-Based Deployment**
  - Jenkins triggers an Ansible playbook located in the `ansible` directory to deploy the latest Docker images.
  - Uses the inventory file to connect to remote nodes and update running containers.
  - Deployment is executed via a non-root user for better security.
- **Post Actions**
  - On success or failure, Jenkins sends automated email notifications using the configured `EMAIL_ID`.
  - The workspace is cleaned after each run to prevent leftover artifacts.

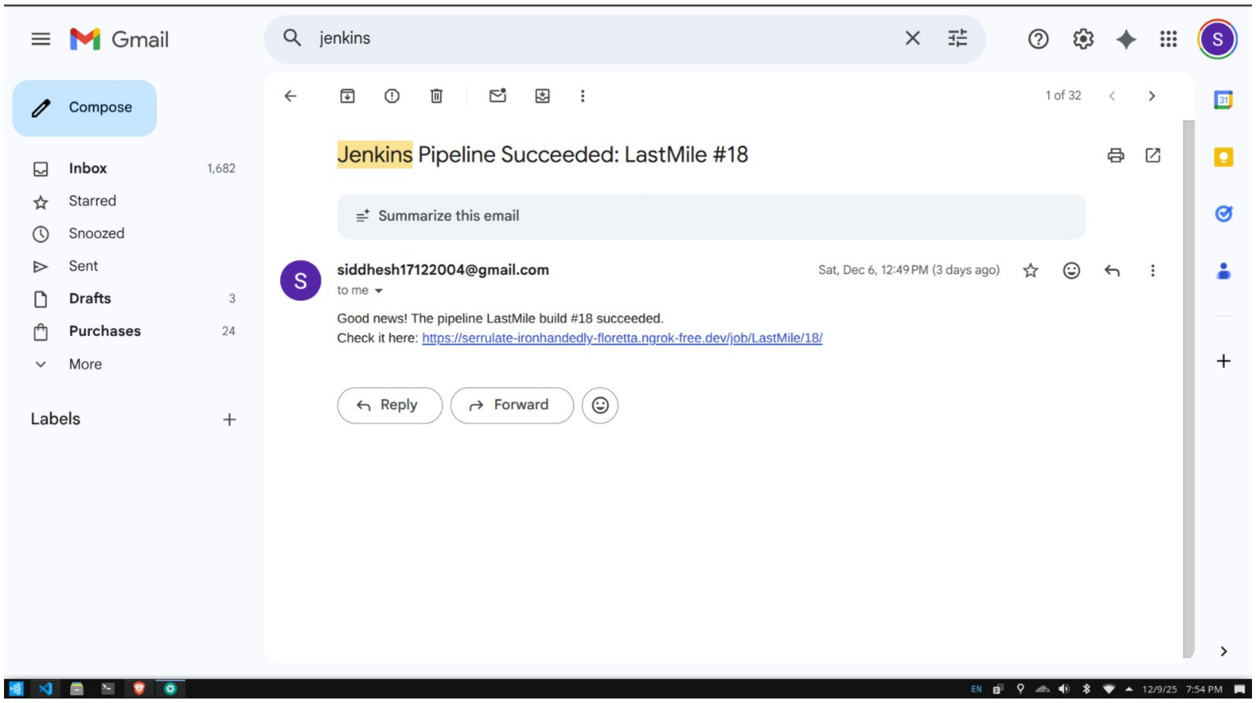


Mail `Jenkins Pipeline Succeeded: LastMile #18` 3.9s

Delete workspace when build is done 76ms

```
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Deferred wipeout is used...
[WS-CLEANUP] done
```

Figure 3: Jenkins build pipeline view



Jenkins Pipeline Succeeded: LastMile #18

siddhesh17122004@gmail.com
to me
Sat, Dec 6, 12:49 PM (3 days ago)

Good news! The pipeline LastMile build #18 succeeded.
Check it here: https://serrulate-ironhandedly-floretta.ngrok-free.dev/job/LastMile/18/

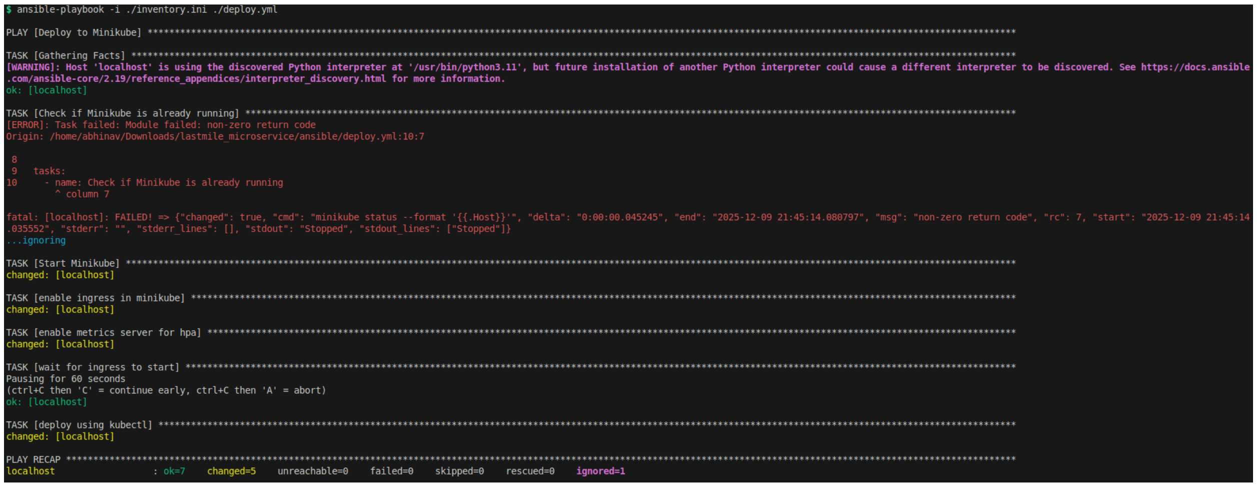
Figure 4: mail received after successful build

Ansible deployment

Ansible is used to automate the deployment of all Kubernetes resources into a local Minikube cluster. The playbook runs on the Jenkins host itself using a local connection and performs cluster initialization, addon configuration, and application deployment.

- **Inventory Configuration**
  - A simple inventory is used where the `localhost` host is mapped to the group `kubehosts` with a `local` connection, allowing the playbook to run directly on the machine where Minikube is installed.
- **Minikube Initialization**
  - The playbook first checks Minikube status and starts the cluster only if it is not already running.
  - Configurable variables define the Minikube driver, CPU allocation, and memory limits.
- **Enabling Required Addons**
  - The Ingress controller addon is enabled to expose services inside the Kubernetes cluster.
  - The Metrics Server addon is enabled to support Horizontal Pod Autoscaling (HPA).
  - A short wait period ensures the Ingress controller becomes fully available before deploying services.
- **Kubernetes Deployment**

- All deployment manifests located in the defined `deployments_dir` are applied recursively using `kubectl apply -R`.
- This includes Deployments, Services, ConfigMaps, Secrets, Ingress, and HPA configurations for all microservices.



```
$ ansible-playbook -i ./inventory.ini ./deploy.yml

PLAY [Deploy to Minikube] ******************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************************
[WARNING]: Host 'localhost' is using the discovered Python interpreter at '/usr/bin/python3.11', but future installation of another Python interpreter could cause a different interpreter to be discovered. See https://docs.ansible
.com/ansible-core/2.19/reference_appendices/interpreter_discovery.html for more information.
ok: [localhost]

TASK [Check if Minikube is already running] ************************************************************************************************************
[ERROR]: Task failed: Module failed: non-zero return code
Origin: /home/abhinav/Downloads/lastmile_microservice/ansible/deploy.yml:10:7

 8
 9   tasks:
10     - name: Check if Minikube is already running
         ^ column 7

fatal: [localhost]: FAILED! => {"changed": true, "cmd": "minikube status --format '{{.Host}}'", "delta": "0:00:00.045245", "end": "2025-12-09 21:45:14.080797", "msg": "non-zero return code", "rc": 7, "start": "2025-12-09 21:45:14
.035552", "stderr": "", "stderr_lines": [], "stdout": "Stopped", "stdout_lines": ["Stopped"]}
...ignoring

TASK [Start Minikube] **********************************************************************************************************************************
changed: [localhost]

TASK [enable ingress in minikube] **********************************************************************************************************************
changed: [localhost]

TASK [enable metrics server for hpa] *******************************************************************************************************************
changed: [localhost]

TASK [wait for ingress to start] ***********************************************************************************************************************
Pausing for 60 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [deploy using kubectl] ****************************************************************************************************************************
changed: [localhost]

PLAY RECAP *********************************************************************************************************************************************
localhost                  : ok=7    changed=5    unreachable=0    failed=0    skipped=0    rescued=0    ignored=1
```

Figure 5: Ansible deployment workflow

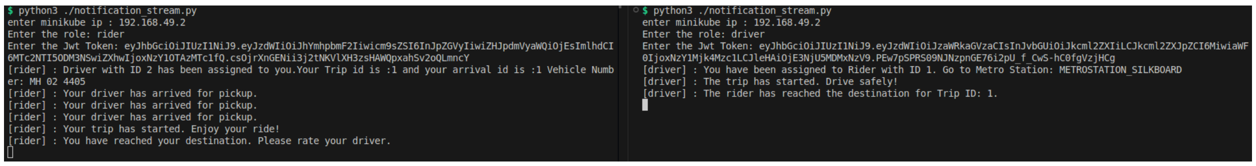
System Testing

System testing was performed to validate the complete end-to-end flow of the LastMile application through the ingress layer. All services were deployed on Minikube, and external interactions were simulated using custom Python test scripts. These tests verified user onboarding, authentication, real-time notifications, driver–rider matching, trip confirmation, and live location updates.

SSE Stream Testing

A Python script using `aiohttp_sse_client` was used to subscribe to the notification service's SSE endpoint through the ingress. The script establishes an authenticated connection using a JWT token and prints any real-time events received by the client. This validated:

- authentication over SSE,
- stable event streaming through ingress,
- delivery of real-time driver/rider updates.



```
$ python3 ./notification_stream.py
enter minikube ip : 192.168.49.2
Enter the role: rider
Enter the Jwt Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhYmhpbmF2Iiwicm9sZSI6InJpZGVyIiwiZHJpdmVyaWQiOjEsImlhdCI
6MTc2NTI5ODM3NSwiZXhwIjoxNzY1OTAzMTc1fQ.csOjrXnGENii3j2tNKVlXH3zsHAWQpxahSv2oQLmncY
[rider] : Driver with ID 2 has been assigned to you.Your Trip id is :1 and your arrival id is :1 Vehicle Numb
er: MH 02 4405
[rider] : Your driver has arrived for pickup.
[rider] : Your driver has arrived for pickup.
[rider] : Your driver has arrived for pickup.
[rider] : Your trip has started. Enjoy your ride!
[rider] : You have reached your destination. Please rate your driver.
```

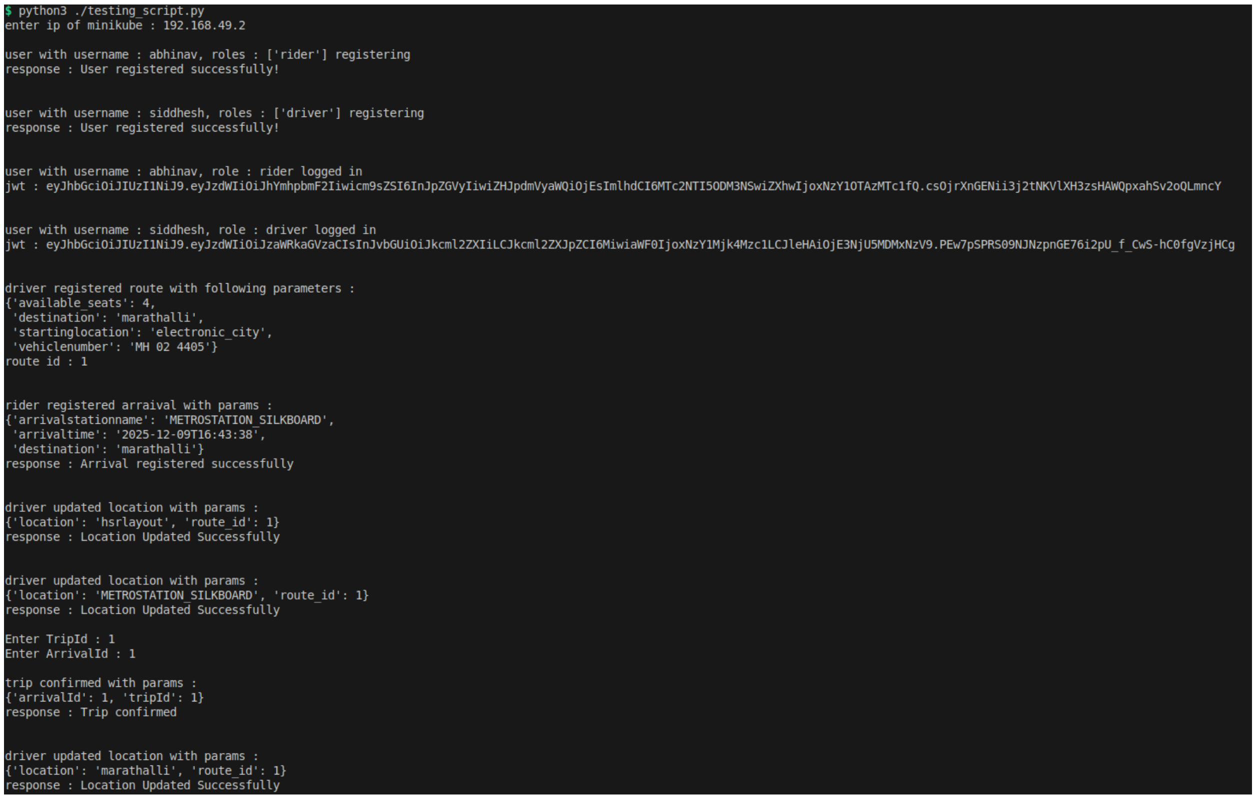
```
$ python3 ./notification_stream.py
enter minikube ip : 192.168.49.2
Enter the role: driver
Enter the Jwt Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzaWRkaGVzaCIsInJvbGUiOiJkcml2ZXIiLCJkcml2ZXJpZCI6MiwiaWF
0IjoxNzY1Mjk4Mzc1LCJleHAiOjE3NjU5MDMxNzV9.PEw7pSPRS09NJNzpnGE76i2pU_f_CwS-hC0fgVzjHCg
[driver] : You have been assigned to Rider with ID 1. Go to Metro Station: METROSTATION_SILKBOARD
[driver] : The trip has started. Drive safely!
[driver] : The rider has reached the destination for Trip ID: 1.
```

Figure 6: notications test

End-to-End Workflow Testing

A separate Python script was created to simulate the full workflow of both a rider and a driver. It issued HTTP requests directly to the ingress controller and validated all major flows, including:

- registration of users with different roles,
- login and retrieval of JWT tokens,
- driver route creation,
- rider arrival registration,
- driver live location updates,
- trip confirmation once a match was found.



```
$ python3 ./testing_script.py
enter ip of minikube : 192.168.49.2

user with username : abhinav, roles : ['rider'] registering
response : User registered successfully!


user with username : siddhesh, roles : ['driver'] registering
response : User registered successfully!


user with username : abhinav, role : rider logged in
jwt : eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhYmhpbmF2Iiwicm9sZSI6InJpZGVyIiwiZHJpdmVyaWQiOjEsImlhdCI6MTc2NTI5ODM3NSwiZXhwIjoxNzY1OTAzMTc1fQ.csOjrXnGENii3j2tNKVlXH3zsHAWQpxahSv2oQLmncY


user with username : siddhesh, role : driver logged in
jwt : eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzaWRkaGVzaCIsInJvbGUiOiJkcml2ZXIiLCJkcml2ZXJpZCI6MiwiaWF0IjoxNzY1Mjk4Mzc1LCJleHAiOjE3NjU5MDMxNzV9.PEw7pSPRS09NJNzpnGE76i2pU_f_CwS-hC0fgVzjHCg


driver registered route with following parameters :
{'available_seats': 4,
 'destination': 'marathalli',
 'startinglocation': 'electronic_city',
 'vehiclenumber': 'MH 02 4405'}
route id : 1


rider registered arraival with params :
{'arrivalstationname': 'METROSTATION_SILKBOARD',
 'arrivaltime': '2025-12-09T16:43:38',
 'destination': 'marathalli'}
response : Arrival registered successfully


driver updated location with params :
{'location': 'hsrlayout', 'route_id': 1}
response : Location Updated Successfully


driver updated location with params :
{'location': 'METROSTATION_SILKBOARD', 'route_id': 1}
response : Location Updated Successfully

Enter TripId : 1
Enter ArrivalId : 1

trip confirmed with params :
{'arrivalId': 1, 'tripId': 1}
response : Trip confirmed


driver updated location with params :
{'location': 'marathalli', 'route_id': 1}
response : Location Updated Successfully
```

Figure 7: testing workflow

The script dynamically constructs all endpoint URLs using the Minikube IP and performs timed updates to mimic real-world behaviour (e.g., delays between driver movements). Both successful responses and intermediate states are logged, enabling easy debugging and verification of service interactions.

Validation Outcome

The combined SSE listener and workflow tester ensured that:

- the microservices interact correctly through ingress,
- state transitions propagate across services via Kafka,
- notifications reach the appropriate authenticated client,
- the system behaves as expected during real-time matching and trip tracking.

This provided complete system-level validation of the platform under a realistic interaction sequence.

Stress and Load Testing

To evaluate the performance and scalability of the authentication workflow, stress testing was conducted using `Locust`. The objective was to generate concurrent login requests against the ingress endpoint and observe system behaviour under increasing load.

Locust Test Design

A custom Locust user class was implemented to simulate repeated login attempts by multiple virtual users. The test script performs the following operations:

- Registers a test user at startup (ignored if already present),
- Executes continuous login requests against the ingress controller,
- Measures latency and response times manually,
- Reports success or failure events using Locust's request event mechanism.

Metrics Captured

The test recorded:

- average login response time,
- number of successful authentication requests per second,
- failure rate and exception traces,
- behaviour of the service under ramping concurrent load.

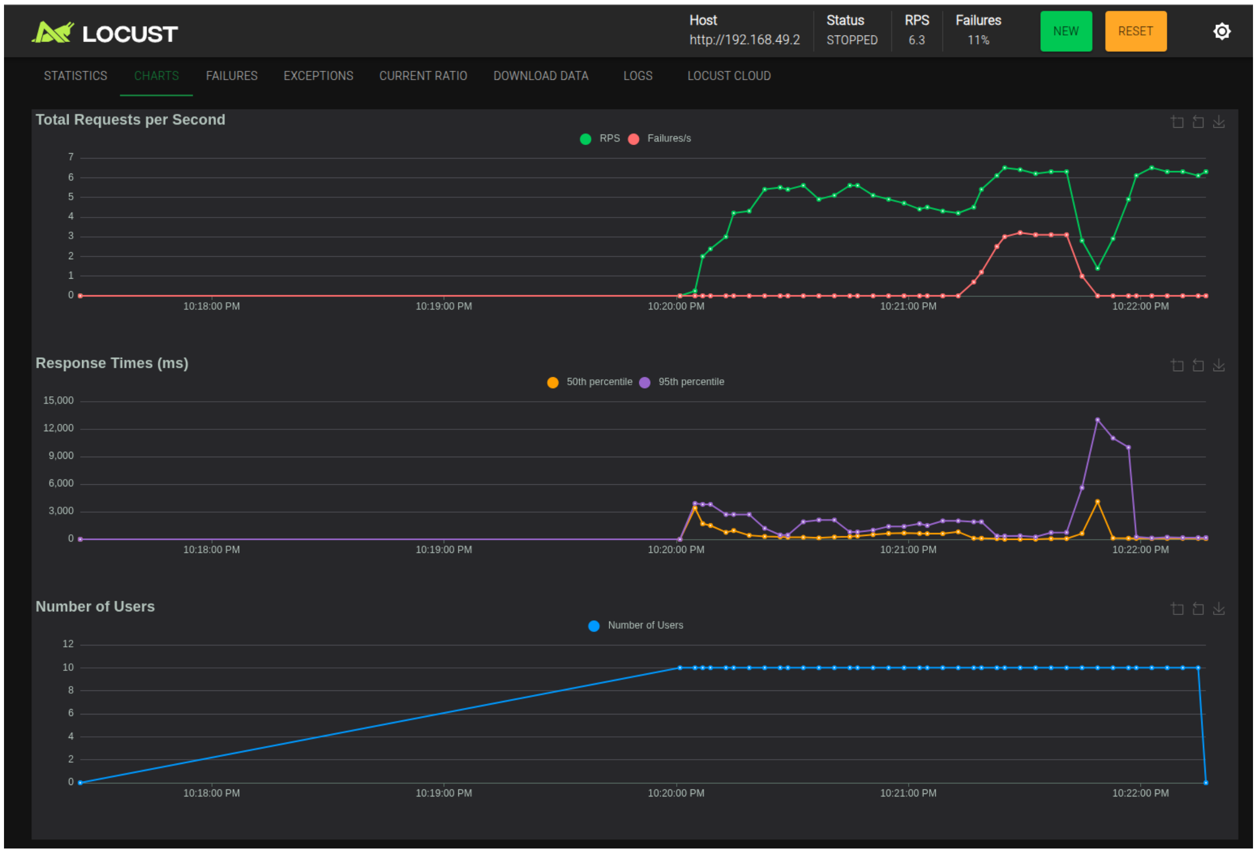


Figure 8: locust dashboard

Locust's built-in dashboard enabled live visualization of throughput (RPS), latency distributions, and error occurrences. This allowed monitoring whether the API gateway, authentication service, or backing PostgreSQL database became bottlenecks during heavy load.

Execution Setup

The Locust script targeted the Minikube ingress endpoint:

- All requests were routed through the NGINX ingress controller,
- Virtual users were gradually increased to observe scaling patterns,
- Wait times between tasks were randomized between 1–2 seconds to emulate realistic usage patterns.

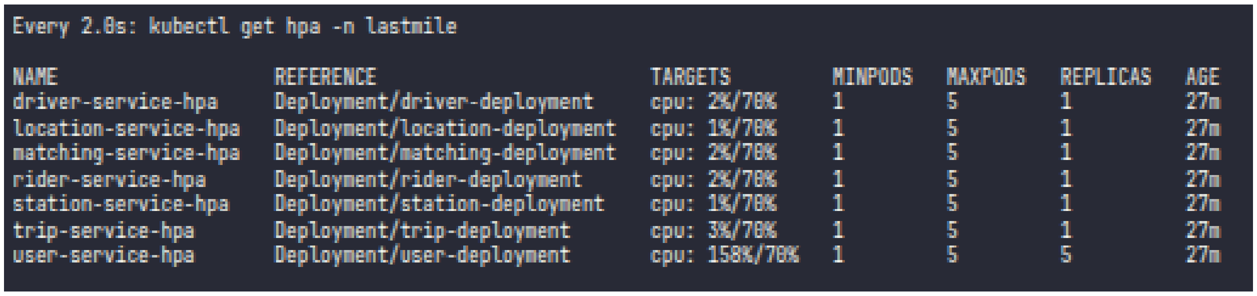
Observations

From the load test:

- authentication throughput remained stable under moderate concurrency,
- response times increased linearly when CPU limits were approached,
- the ingress and authentication layers handled concurrent requests without failures until stress threshold,

- PostgreSQL I/O was the limiting factor at high loads.

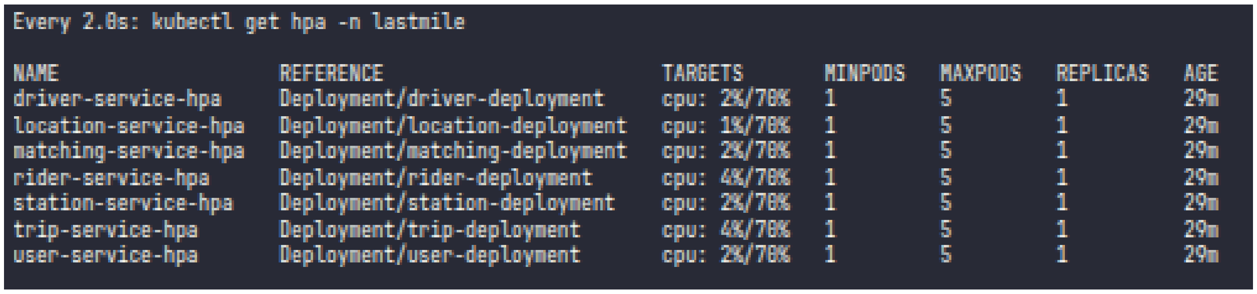
This stress test validated that the login functionality behaves reliably under load and helped identify resource constraints relevant for future autoscaling and optimizations.



```
Every 2.0s: kubectl get hpa -n lastmile

NAME                   REFERENCE                           TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
driver-service-hpa     Deployment/driver-deployment        cpu: 2%/70%     1         5         1          27m
location-service-hpa   Deployment/location-deployment      cpu: 1%/70%     1         5         1          27m
matching-service-hpa   Deployment/matching-deployment      cpu: 2%/70%     1         5         1          27m
rider-service-hpa      Deployment/rider-deployment         cpu: 2%/70%     1         5         1          27m
station-service-hpa    Deployment/station-deployment       cpu: 1%/70%     1         5         1          27m
trip-service-hpa       Deployment/trip-deployment          cpu: 3%/70%     1         5         1          27m
user-service-hpa       Deployment/user-deployment          cpu: 158%/70%   1         5         5          27m
```

Figure 9: replica count increase in userservice during stress testing (locust)



```
Every 2.0s: kubectl get hpa -n lastmile

NAME                   REFERENCE                           TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
driver-service-hpa     Deployment/driver-deployment        cpu: 2%/70%   1         5         1          29m
location-service-hpa   Deployment/location-deployment      cpu: 1%/70%   1         5         1          29m
matching-service-hpa   Deployment/matching-deployment      cpu: 2%/70%   1         5         1          29m
rider-service-hpa      Deployment/rider-deployment         cpu: 4%/70%   1         5         1          29m
station-service-hpa    Deployment/station-deployment       cpu: 2%/70%   1         5         1          29m
trip-service-hpa       Deployment/trip-deployment          cpu: 4%/70%   1         5         1          29m
user-service-hpa       Deployment/user-deployment          cpu: 2%/70%   1         5         1          29m
```

Figure 10: replica count back to normal in userservice after stress testing (locust)

link to full recording in asciineam : `https://asciinema.org/a/sXupGyJNLkK6Tfvh75dtue4wZ`

Advanced Features

The system incorporates several advanced cloud-native, distributed systems, project specific features:

- **Horizontal Pod Autoscaling (HPA)** to dynamically scale microservices based on real-time workload.
- **Kafka-based Event Streaming Pipeline** for high-throughput and decoupled communication between services.
- **gRPC with Protobuf API Contracts** for efficient, type-safe inter-service communication.
- **Kubernetes Ingress** for unified external access and traffic management.
- **Kubernetes Secrets** for secure confidential configuration management.
- **Usage of Secrets in Kubernetes** : We have used secrets in kubernetes for database credentials.

Innovative Solutions

The following innovative approaches enhance system performance, security, and user experience:

- **Server-Sent Events (SSE)** in the Notification Service providing efficient real-time updates.
- **Distroless Container Images** for optimized memory footprint and improved container security.
- **Lightweight Proximity Detection Logic** that uses curated station-to-location mappings to trigger matches internally, avoiding external geolocation APIs and reducing latency.
- **Scheduled Jobs in Spring Boot** to execute periodic tasks such as refreshing driver availability, cleaning stale ride requests, and triggering time-based matching operations. This enables autonomous background processing without manual intervention and improves the system's real-time responsiveness.
- **DaemonSet Logging Stack (ELK)** to ensure observability without affecting service containers. We used this approach instead of sidecar container approach as this is more efficient and consumes less memory.

Domain Classification

This project falls under the domain of **Real-Time Distributed Systems** and **Big Data Stream Processing**. The system continuously ingests high-frequency driver location updates, processes them using event-driven microservices, and performs real-time proximity matching using Kafka-based streaming pipelines. The architecture demonstrates fault tolerance, horizontal scalability (from 1 to 5 matching services), and real-time communication using gRPC across independently deployable Kubernetes services. Due to its emphasis on **real-time networking**, **distributed computing**, and **large-scale streaming data processing**, the project qualifies as a domain-specific application rather than a generic full-stack system.

Reference Links

- **GitHub Repository:**
  https://github.com/Abhinav-Kumar012/lastmile_microservice
- **HPA Autoscaling Demonstration (Asciinema):**
  https://asciinema.org/a/sXupGyJNLkK6Tfvh75dtue4wZ