Term: Fall 2025 **Subject:** Computer Science & Engineering (CSE) **Number:** 512

Course Title: Distributed **D**atabase **S**ystems (CSE 512)

GROUP PROJECT MILESTONE 3

Project Title:
Geo-Distributed Ride-Sharing System

Team Name and Members:
Divyesh Patel (1238082393), Aryan Talati (1233667489), Abhi Sachdeva (1221508080), Harshrajsinh Rathod (1237929751)

Team Member	Primary Role	Key Contributions	Distribution
Divyesh Patel	System Architect	Architected the cloud infrastructure on AWS EC2, provisioning instances across multiple regions. Containerized the environment using Docker Swarm, ensuring networking between independent CockroachDB clusters.	25%
Aryan Talati	Backend Developer	Developed the API layer using FastAPI to handle CRUD operations on the database. Implemented Coordinator logic, integrating custom geohashing algorithms to route Read & Write queries to correct regions.	25%
Abhi Sachdeva	Data Engineer	Designed database schema for Users, Riders, and Drivers. Generated a large-scale synthetic dataset (340,000+ rows) using Faker to simulate realistic production loads for stress testing.	25%
Harshrajsinh Rathod	Data Engineer	Configured replication modules, balancing synchronous replication for local consistency and asynchronous replication for cross-region backup. Designed failure scenarios to verify system availability.	25%

1. Introduction

1.1 Background

Ride-sharing platforms such as Uber and Lyft require low-latency, globally scalable data management systems. A centralized database has high latency and a single point of failure, making it unsuitable for large-scale distributed systems. To address these challenges, our project implements a geo-distributed architecture designed to minimize access time while ensuring reliability. Our architecture prioritizes consistency and availability through data replication, partitioning, fault tolerance, and real-time monitoring.

1.2 Problem Statement

Core Problem: The main challenge is to engineer a specialized, geo-distributed database for ride-sharing that maintains a balance between low latency for local user operations and data consistency across global regions. Conventional database systems fail to meet these requirements due to latency bottlenecks and a single point of failure.

1.3 Objectives:

- **Data Partitioning:** Implement a geographic sharding strategy to localize data storage, reducing query latency and minimizing expensive cross-region network traffic.
- **Data Replication:** Establish a hybrid replication model that ensures strong consistency within regions through replication
- **Query Routing:** Design a coordinator node capable of parsing clients' location data and routing read and write requests to the appropriate regional shard with minimal overhead.
- **Fault Tolerance:** Architect a resilient system using RAFT-based consensus to ensure automatic leader election and seamless recovery in the event of node failures or regional outages.

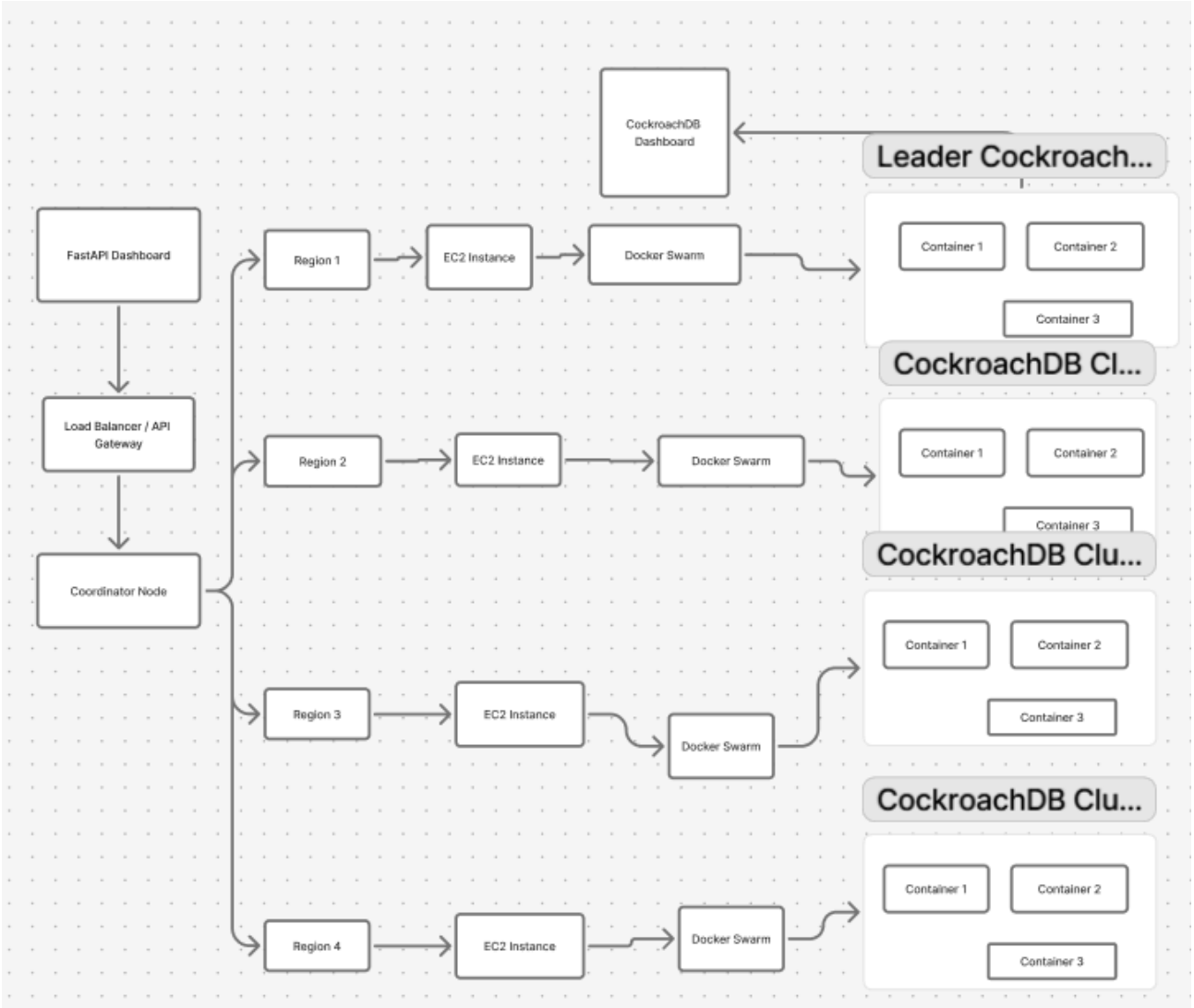
1.4 Scope

Feature	Description
Data Generation (In Scope)	340,000+ rows of data using Faker library
Geographic Partitioning (In Scope)	Custom coordinate-based geohash sharding across 4 regions
Data Replication (In Scope)	Synchronous replication within regions. Asynchronous replication across regions
Query Routing (In Scope)	Coordinator node with a custom geohash algorithm for read/write query routing
Fault Tolerance (In Scope)	RAFT-based consensus, heartbeat monitoring, and automatic leader election

API Layer (In Scope)	RESTful CRUD operations using FastAPI
System Architecture (In Scope)	AWS EC2 instances with Docker Swarm containerization. CockroachDB clusters for each regional shard
Data Security (Out-of-Scope)	TLS Encryption and data proxy for access control; Not implemented
Mobile Frontend (Out-of-Scope)	Only API endpoints are provided; No application is implemented

2. Project Description

2.1 System Design



2.2 Implementation Details

a. Data Partitioning Strategy

- Geographic sharding by partitioning data via latitude/longitude coordinates and geohashing them for an independent CockroachDB cluster for each region
- Reduces query latency and limits cross-region traffic

Flow description:

- The frontend sends requests to the Load Balancer / API Gateway using FastAPI, which forwards them to the Coordinator Node.
- The Coordinator Node computes the home region via Geohash partitioning, which is manually implemented, and routes requests to Region 1 (US-East), Region 2 (US-West), or Region 3 (Europe).
- Each region runs on an EC2 Instance with Docker Swarm, hosting a container/CockroachDB node that stores a shard for the corresponding home region.
- Replication occurs within each region synchronously and across regions asynchronously.

```
root@localhost:26257/rideshare> SHOW RANGES FROM TABLE drivers;
     start_key     |      end_key      |     index_name     | range_id |     range_size_mb      | lease_holder |       lease_holder_locality        |    replicas     |
                                                                                 replica_localities
                   | voting_replicas | non_voting_replicas
-------------------+-------------------+-------------------+----------+------------------------+--------------+------------------------------------+-----------------+----------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------+-----------------+---------------------
  NULL             | /" "              |                   |      214 |                      0 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,4,5,8}   | {"region=us-eas
t,zone=us-east-1a","region=us-east,zone=us-east-1c","region=us-east,zone=us-east-1b","region=ap-south,zone=ap-south-1a","region=eu-central,zone=eu-central-1a","region=us-west,zone=us-wes
t-1a"}            | {2,3,1}         | {4,8,5}
  /" "             | /" "/PrefixEnd    |                   |      215 |  1.1068400000000000000 |            6 | region=ap-south,zone=ap-south-1b    | {2,4,6,7,8,12}  | {"region=us-eas
t,zone=us-east-1c","region=ap-south,zone=ap-south-1a","region=ap-south,zone=ap-south-1b","region=ap-south,zone=ap-south-1c","region=us-west,zone=us-west-1a","region=eu-central,zone=eu-ce
ntral-1b"}        | {4,7,6}         | {2,8,12}
  /" "/PrefixEnd   | /"@"              |                   |      216 |                      0 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,5,7,8}   | {"region=us-eas
t,zone=us-east-1a","region=us-east,zone=us-east-1c","region=us-east,zone=us-east-1b","region=eu-central,zone=eu-central-1a","region=ap-south,zone=ap-south-1c","region=us-west,zone=us-wes
t-1a"}            | {2,3,1}         | {7,8,5}
  /"@"             | /"@"/PrefixEnd    |                   |      217 |  1.1104310000000000000 |            5 | region=eu-central,zone=eu-central-1a | {2,4,5,8,9,12}  | {"region=us-eas
t,zone=us-east-1c","region=ap-south,zone=ap-south-1a","region=eu-central,zone=eu-central-1a","region=us-west,zone=us-west-1a","region=eu-central,zone=eu-central-1c","region=eu-central,zo
ne=eu-central-1b"} | {12,9,5}       | {4,2,8}
  /"@"/PrefixEnd   | /"\x80"           |                   |      218 |                      0 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,4,11,12} | {"region=us-eas
t,zone=us-east-1a","region=us-east,zone=us-east-1c","region=us-east,zone=us-east-1b","region=ap-south,zone=ap-south-1a","region=us-west,zone=us-west-1c","region=eu-central,zone=eu-centra
l-1b"}            | {2,3,1}         | {4,11,12}
  /"\x80"          | /"\x80"/PrefixEnd |                   |      219 |  1.0884820000000000000 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,4,5,8}   | {"region=us-eas
t,zone=us-east-1a","region=us-east,zone=us-east-1c","region=us-east,zone=us-east-1b","region=ap-south,zone=ap-south-1a","region=eu-central,zone=eu-central-1a","region=us-west,zone=us-wes
t-1a"}            | {2,3,1}         | {4,8,5}
  /"\x80"/PrefixEnd | /"\xc0"          |                   |      220 |                      0 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,4,5,10}  | {"region=us-eas
t,zone=us-east-1a","region=us-east,zone=us-east-1c","region=us-east,zone=us-east-1b","region=ap-south,zone=ap-south-1a","region=eu-central,zone=eu-central-1a","region=us-west,zone=us-wes
t-1b"}            | {2,3,1}         | {4,10,5}
  /"\xc0"          | /"\xc0"/PrefixEnd |                   |      221 |  1.0897790000000000000 |           10 | region=us-west,zone=us-west-1b      | {1,5,6,8,10,11} | {"region=us-eas
t,zone=us-east-1a","region=eu-central,zone=eu-central-1a","region=ap-south,zone=ap-south-1b","region=us-west,zone=us-west-1a","region=us-west,zone=us-west-1b","region=us-west,zone=us-wes
t-1c"}            | {11,8,10}       | {6,1,5}
  /"\xc0"/PrefixEnd | /" "             |                   |      222 |                      0 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,4,5,11}  | {"region=us-eas
t,zone=us-east-1a","region=us-east,zone=us-east-1c","region=us-east,zone=us-east-1b","region=ap-south,zone=ap-south-1a","region=eu-central,zone=eu-central-1a","region=us-west,zone=us-wes
t-1c"}            | {2,3,1}         | {4,11,5}
  /" "             | /" "/PrefixEnd    | idx_drivers_region |      223 | 0.37317600000000000000 |            7 | region=ap-south,zone=ap-south-1c    | {2,4,6,7,8,9}   | {"region=us-eas
t,zone=us-east-1c","region=ap-south,zone=ap-south-1a","region=ap-south,zone=ap-south-1b","region=ap-south,zone=ap-south-1c","region=us-west,zone=us-west-1a","region=eu-central,zone=eu-ce
ntral-1c"}        | {4,6,7}         | {2,8,9}
  /" "/PrefixEnd   | /"@"              | idx_drivers_region |      227 |                      0 |            1 | region=us-east,zone=us-east-1a      | {1,2,3,4,9,10}  | {"region=us-eas
```

b. Replication and Consistency Model

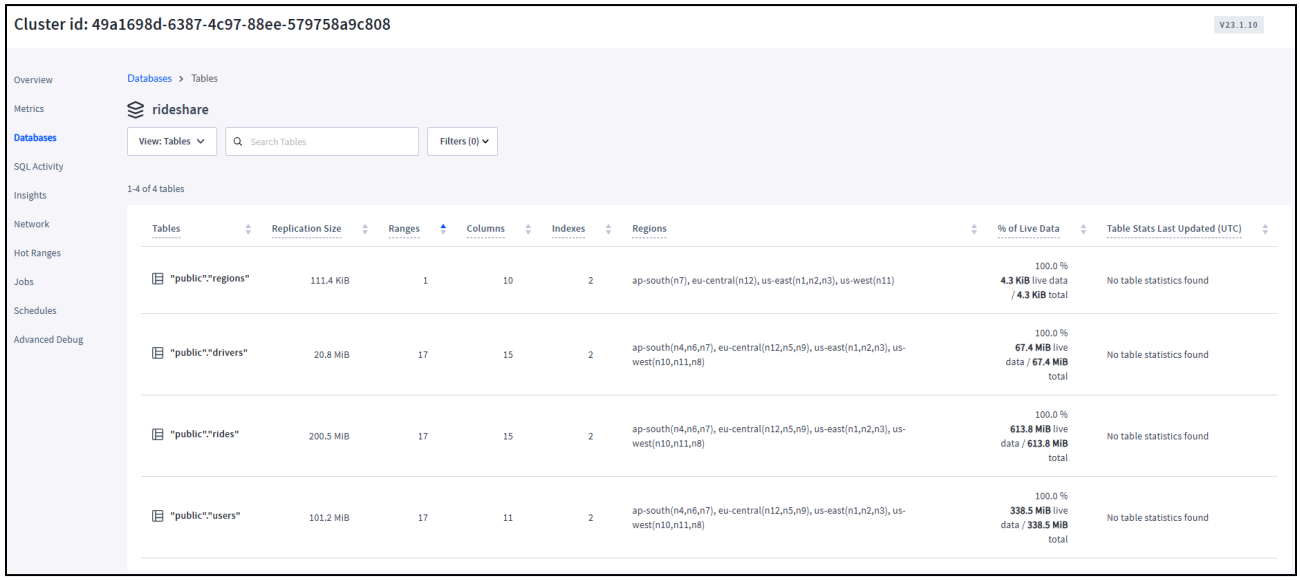
- Synchronous replication to ensure strong consistency within regions
- Asynchronous replication to provide consistency across regions
- Read and write flows balance performance and availability
- Flow of read/write operations

Write Operation:

1. Client sends a ride request to the server.
2. The server validates the request and formats SQL INSERT.
3. The server asks the coordinator which region owns the range for this user/location.
4. The coordinator returns the region responsible for that range.
5. Server routes the write directly to that region's EC2 node.
6. The node writes data to the CockroachDB leader in that range.
7. The leader replicates data to follower nodes in the same region.
8. The leader confirms the commit back to the server.
9. The server responds to the client with success.

Read Operation:

1. Client requests nearby drivers from the server.
2. The server calculates the region to query based on location.
3. The server asks the coordinator which region owns the relevant range.
4. The coordinator returns the responsible region.
5. The server sends a SELECT query directly to the replica in those regions.
6. Nodes read data and return results to the server.
7. The server aggregates results if multiple regions are involved.
8. The server responds to the client with driver information



c. Fault Tolerance and Recovery

- Heartbeat checks detect failures. RAFT-based leader election ensures recovery.
- Write-Ahead Logs maintain data durability.
- Different test failure scenarios

d. Query Processing and Optimization

- The coordinator performs query planning and parallel shared queries.
- Local joins occur in-region, with global aggregation at the coordinator.

e. Monitoring and Performance Layer

Metrics like latency and throughput will be collected through FastAPI and CockroachDB dashboard

2.3 Data Strategy

We generated realistic synthetic data using the Faker library in Python. Additionally, we have 4 data tables:

1. User (user_id, name, email, location, region)

```
cursor.execute("""
    CREATE TABLE users (
        user_id UUID PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        email VARCHAR(255) NOT NULL,
        phone VARCHAR(50),
        latitude FLOAT NOT NULL,
        longitude FLOAT NOT NULL,
        region VARCHAR(50) NOT NULL,
        rating FLOAT DEFAULT 5.0,
        total_rides INT DEFAULT 0,
        created_at TIMESTAMP DEFAULT NOW(),
        crdb_region crdb_internal_region AS (
            CASE region
                WHEN 'us-east' THEN 'us-east'::crdb_internal_region
                WHEN 'us-west' THEN 'us-west'::crdb_internal_region
                WHEN 'eu-central' THEN 'eu-central'::crdb_internal_region
                WHEN 'ap-south' THEN 'ap-south'::crdb_internal_region
                ELSE 'us-east'::crdb_internal_region
            END
        ) STORED NOT NULL
    ) LOCALITY REGIONAL BY ROW
""")
```

2. Driver (driver_id, name, location, vehicle_info, availability)

```
cursor.execute("""
    CREATE TABLE drivers (
        driver_id UUID PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        email VARCHAR(255) NOT NULL,
        phone VARCHAR(50),
        latitude FLOAT NOT NULL,
        longitude FLOAT NOT NULL,
        region VARCHAR(50) NOT NULL,
        geohash VARCHAR(50),
        vehicle_info VARCHAR(255),
        license_plate VARCHAR(50),
        availability VARCHAR(50) DEFAULT 'offline',
        rating FLOAT DEFAULT 5.0,
        total_rides INT DEFAULT 0,
        created_at TIMESTAMP DEFAULT NOW(),
        crdb_region crdb_internal_region AS (
            CASE region
                WHEN 'us-east' THEN 'us-east'::crdb_internal_region
                WHEN 'us-west' THEN 'us-west'::crdb_internal_region
                WHEN 'eu-central' THEN 'eu-central'::crdb_internal_region
                WHEN 'ap-south' THEN 'ap-south'::crdb_internal_region
                ELSE 'us-east'::crdb_internal_region
            END
        ) STORED NOT NULL
    ) LOCALITY REGIONAL BY ROW
""")
```

3. Ride (ride_id, user_id, driver_id, pickup, dropoff, timestamp, price)

```
cursor.execute("""
    CREATE TABLE rides (
        ride_id UUID PRIMARY KEY,
        user_id UUID NOT NULL,
        driver_id UUID,
        pickup_lat FLOAT NOT NULL,
        pickup_lon FLOAT NOT NULL,
        dropoff_lat FLOAT NOT NULL,
        dropoff_lon FLOAT NOT NULL,
        region VARCHAR(50) NOT NULL,
        pickup_geohash VARCHAR(50),
        status VARCHAR(50) DEFAULT 'requested',
        price FLOAT DEFAULT 0.0,
        distance_km FLOAT DEFAULT 0.0,
        duration_minutes INT DEFAULT 0,
        timestamp TIMESTAMP DEFAULT NOW(),
        crdb_region crdb_internal_region AS (
            CASE region
                WHEN 'us-east' THEN 'us-east'::crdb_internal_region
                WHEN 'us-west' THEN 'us-west'::crdb_internal_region
                WHEN 'eu-central' THEN 'eu-central'::crdb_internal_region
                WHEN 'ap-south' THEN 'ap-south'::crdb_internal_region
                ELSE 'us-east'::crdb_internal_region
            END
        ) STORED NOT NULL
    ) LOCALITY REGIONAL BY ROW
""")
```

4. Regions (Metadata for each region)

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS regions (
        region_id UUID PRIMARY KEY,
        region_code VARCHAR(50) UNIQUE NOT NULL,
        name VARCHAR(255) NOT NULL,
        lat_min FLOAT NOT NULL,
        lat_max FLOAT NOT NULL,
        lon_min FLOAT NOT NULL,
        lon_max FLOAT NOT NULL,
        node_count INT DEFAULT 3,
        primary_node VARCHAR(255),
        created_at TIMESTAMP DEFAULT NOW()
    )
""")
```

We ensure data privacy and show compliance with GDPR Regulations by keeping data in specific regions. Additionally, we implemented anonymization techniques for cross-region analyses.

3. Implementation Plan

3.1 Methodology

Technique 1: Geographic Partitioning

We implemented hash-based partitioning on the data by geographic region so that users, drivers, and rides can be on a local database node for each region

Technique 2: Data Replication

Depending on the data, we implemented different replication strategies. We performed synchronous replication on critical data that is within the same region (i.e., active rides). For less critical data (i.e., historical data) across regions, we performed asynchronous replication.

Technique 3: Distributed Query Processing

We implemented a query routing system using a coordinator node to reduce cross-region traffic. For local queries, we will have them execute in one region. For cross-region queries, we will aggregate data from all regions

Technique 4: Fault Tolerance

We implemented automatic node failover to demonstrate that our database system remains operational even after a node or EC2 instance failure. We utilized the RAFT consensus algorithm and implemented a heartbeat monitoring system to detect node failure.

3.2 Technology Stack

Component	Technology	Rationale
Programming language	Python	Asynchronous support & ease of prototyping
Framework	FastAPI	Supports concurrency
Database	CockroachDB	ACID compliance and replication support
Containerization	AWS EC2 and Docker Swarm	Simulate a multi-node distributed setup
Monitoring	CockroachDB Dashboard	Real-time monitoring & visualization

3.3 Key Features

1. **Partitioning**

- Data partitioning according to region

2. **Consistency**

- Do concurrent writes to the same key and read immediately from different replicas.
- Ensure the latest value is always returned.

3. **Query Routing**

- Route query to the correct regional shard using the coordinator node and a custom geohashing algorithm

4. **Challenges**

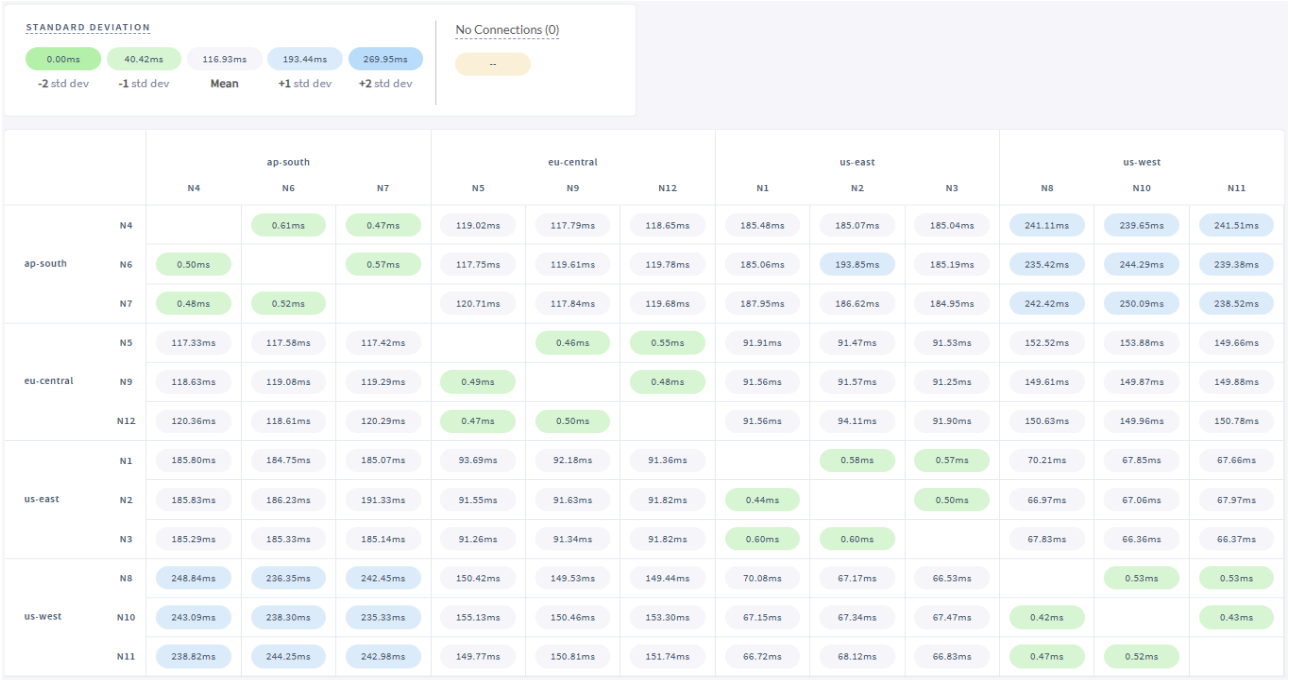
- Synchronizing write-ahead logs across asynchronous replicas caused latency
- When a region goes down, nothing is returned, as replication across regions is not properly occurring

4. Evaluation

4.1 Performance Metrics

Metric	Description	Target	Actual Result	Measurement Method
Read Latency (Local - Hot)	Elapsed time to complete a read query within the same region	< 50 ms	4 ms	SELECT query round-trip time
Read Latency (Cross-Region)	Elapsed time to complete a read within multiple regions	< 1.5s	506 ms	Aggregated query time
Write Latency (Local)	Elapsed time to complete a write with synchronous replication	< 200 ms	113 ms	INSERT/UPDATE time
Throughput	Requests per second	> 500 TPS	540 TPS	Load test with concurrent requests
Replication Delay	Delay between primary write and replica update	<1s	150 ms	CockroachDB Metrics
Data Consistency	Reads return the latest committed write	100%	100%	Stop node, measure recovery time

Failover Time	Recovery after node failure	< 5s	3s (Automatic)	Write, then immediately read



Curl

```
curl -X 'GET' \
  'http://52.3.233.36:8000/rides/c20a54d5-1e63-49d8-8800-2b4a602e0892' \
  -H 'accept: application/json'
```

Request URL

```
http://52.3.233.36:8000/rides/c20a54d5-1e63-49d8-8800-2b4a602e0892
```

Server response

Code	Details
200	**Response body**


```
{
  "ride_id": "c20a54d5-1e63-49d8-8800-2b4a602e0892",
  "user_id": "e67720f3-7281-45b8-b4bc-57ac57feb242",
  "driver_id": "14d14f72-8a98-4ec7-97dc-4a0a8b77fe4a",
  "pickup_lat": 39.970676,
  "pickup_lon": -75.205729,
  "dropoff_lat": 40.0054,
  "dropoff_lon": -75.249337,
  "region": "us-east",
  "pickup_geohash": "dr4e2s",
  "status": "completed",
  "price": 16.54,
  "distance_km": 5.36,
  "duration_minutes": 30,
  "timestamp": "2025-12-01T19:02:29.816833",
  "crdb_region": "us-east"
}
```


Download

Response headers

```
content-length: 430
content-type: application/json
date: Tue,02 Dec 2025 23:00:24 GMT
server: uvicorn
x-process-time-ms: 4.00
```

Curl

```
curl -X 'GET' \
  'http://52.3.233.36:8000/rides/ea090999-8bb1-4436-ba0e-e7fe93d61f6f' \
  -H 'accept: application/json'
```

Request URL

```
http://52.3.233.36:8000/rides/ea090999-8bb1-4436-ba0e-e7fe93d61f6f
```

Server response

Code	Details
200	Response body

```
{
  "ride_id": "ea090999-8bb1-4436-ba0e-e7fe93d61f6f",
  "user_id": "e99dd6fa-a39e-492f-a6ee-9908a08531d5",
  "driver_id": "72457d06-e2b6-4a86-8d62-47ddaa01a848",
  "pickup_lat": 19.14884,
  "pickup_lon": 72.902636,
  "dropoff_lat": 19.127829,
  "dropoff_lon": 72.875999,
  "region": "ap-south",
  "pickup_geohash": "te7ufv",
  "status": "completed",
  "price": 12.57,
  "distance_km": 3.65,
  "duration_minutes": 23,
  "timestamp": "2025-12-01T19:03:22.776303",
  "crdb_region": "ap-south"
}
```

Download

Response headers

```
content-length: 431
content-type: application/json
date: Tue,02 Dec 2025 23:03:15 GMT
server: uvicorn
x-process-time-ms: 506.33
```

Curl

```
curl -X 'PUT' \
  'http://52.3.233.36:8000/rides/6d129125-3d8c-406b-b483-52ffe2c23d68' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "ride_id": "6d129125-3d8c-406b-b483-52ffe2c23d68",
  "user_id": "7e4d9bc9-45d7-41f9-bfb9-7f4bb88c5ba7",
  "driver_id": "aae86d51-3f71-4830-97df-2a44931f7493",
  "pickup_lat": 38.939729,
  "pickup_lon": -77.060874,
  "dropoff_lat": 38.932686,
  "dropoff_lon": -77.052556,
  "region": "us-east",
  "pickup_geohash": "dqcjw2",
  "status": "cancelled",
  "price": 5.9,
  "distance_km": 1.06,
  "duration_minutes": 9,
  "timestamp": "2025-12-01T19:03:06.841966",
  "crdb_region": "us-east"
}'
```

Request URL

```
http://52.3.233.36:8000/rides/6d129125-3d8c-406b-b483-52ffe2c23d68
```

Server response

Code	Details
200	Response body

```
{
  "ride_id": "6d129125-3d8c-406b-b483-52ffe2c23d68",
  "user_id": "7e4d9bc9-45d7-41f9-bfb9-7f4bb88c5ba7",
  "driver_id": "aae86d51-3f71-4830-97df-2a44931f7493",
  "pickup_lat": 38.939729,
  "pickup_lon": -77.060874,
  "dropoff_lat": 38.932686,
  "dropoff_lon": -77.052556,
  "region": "us-east",
  "pickup_geohash": "dqcjw2",
  "status": "cancelled",
  "price": 5.9,
  "distance_km": 1.06,
  "duration_minutes": 9,
  "timestamp": "2025-12-01T19:03:06.841966",
  "crdb_region": "us-east"
}
```

Download

Response headers

```
content-length: 429
content-type: application/json
date: Tue,02 Dec 2025 23:05:30 GMT
server: uvicorn
x-process-time-ms: 113.62
```

Responses

4.2 Testing different types of failures

i. Node/Instance Failures

Objective: Ensure the system continues operating if an EC2 instance or Docker container fails

Test Method:

1. Simulate a node crash by stopping a node in an EC2 instance.
2. Observe if another node is automatically being created by cockroachdb and docker

```
ubuntu@ip-172-31-10-106:~$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED             STATUS             PORTS                 NAMES
9ce4acc22217   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   About an hour ago   Up About an hour   8080/tcp, 26257/tcp   rideshare_roach-east-2.1.ma5v31lf67688a996muxj2c2zo
f6046013bccd   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   25 hours ago        Up 25 hours        8080/tcp, 26257/tcp   rideshare_roach-east-1.1.0flkx67deuaax4cv12ag5r120
27c1c5789eea   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   25 hours ago        Up 25 hours        8080/tcp, 26257/tcp   rideshare_roach-east-3.1.kziwr9t0v3krjkdnknn6ytsse
ubuntu@ip-172-31-10-106:~$ docker stop 9ce4acc22217
9ce4acc22217
ubuntu@ip-172-31-10-106:~$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                 NAMES
f6046013bccd   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   25 hours ago     Up 25 hours     8080/tcp, 26257/tcp   rideshare_roach-east-1.1.0flkx67deuaax4cv12ag5r120
27c1c5789eea   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   25 hours ago     Up 25 hours     8080/tcp, 26257/tcp   rideshare_roach-east-3.1.kziwr9t0v3krjkdnknn6ytsse
ubuntu@ip-172-31-10-106:~$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                 NAMES
b1be9604ef12   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   7 seconds ago    Up 1 second     8080/tcp, 26257/tcp   rideshare_roach-east-2.1.og4ihik133xibsmvdyvmn1k75
f6046013bccd   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   25 hours ago     Up 25 hours     8080/tcp, 26257/tcp   rideshare_roach-east-1.1.0flkx67deuaax4cv12ag5r120
27c1c5789eea   cockroachdb/cockroach:v23.1.10   "/cockroach/cockroac…"   25 hours ago     Up 25 hours     8080/tcp, 26257/tcp   rideshare_roach-east-3.1.kziwr9t0v3krjkdnknn6ytsse
ubuntu@ip-172-31-10-106:~$
```

As we can see in the screenshot above, we intentionally stopped one of the nodes (ID: 9ce4acc22217) in Docker, and after a few seconds, another new node(ID: b1be9604ef12) automatically gets created.

ii. Region Outage

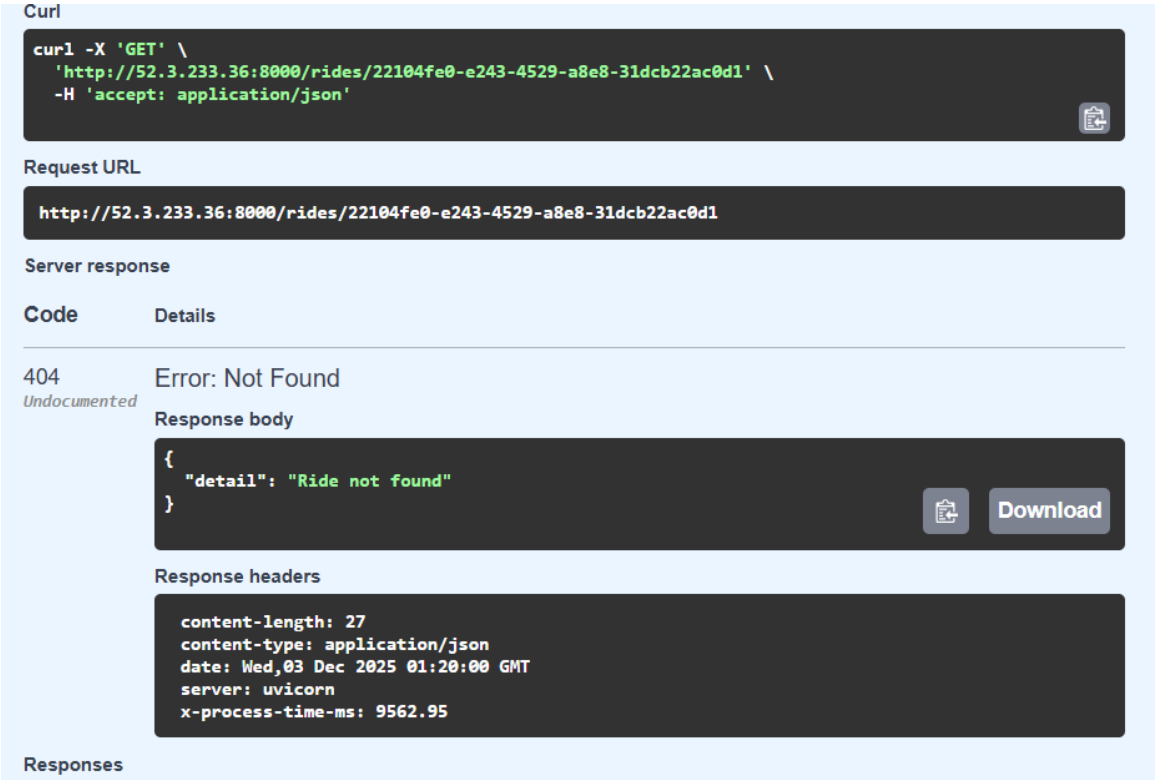
Objective: Test system behavior when an entire region becomes unavailable.

Test Method:

1. Shut down all EC2 instances in a region or block network traffic to the region.
2. Perform writes/reads for ranges that belong to that region.
3. Ensure that errors are handled gracefully and that other regions continue to function.
4. Test recovery once the region comes back online and verify replication catches up.

```
ubuntu@ip-172-31-10-106:~$ docker node ls
ID                            HOSTNAME            STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
tq819avqo70bjumrb96y60z2a     ip-172-31-5-43      Ready     Drain                           28.2.2
qywg8hmode8qj682rt7n316ts *   ip-172-31-10-106    Ready     Active         Leader           28.2.2
la4lhknblwo1acka7cidxfkgv     ip-172-31-36-206    Ready     Active                          28.2.2
wtwthlcsmbxfpzgningp7fsqq     ip-172-31-42-102    Ready     Active                          28.2.2
ubuntu@ip-172-31-10-106:~$
```

The instance in the region us-west is stopped



When we try to perform an operation on the unavailable region, it gets gracefully handled by error; queries to other regions perform as expected.

iii. Network Partition / Latency

Objective: Check system resilience to network splits or high latency.

Test Method:

1. Introduce network latency or simulate a partition between regions.
2. Monitor write latency, read availability, and Raft consensus behavior.
3. Ensure the coordinator correctly maps ranges and retries failover if needed.

4.3 Results

Overall Performance Evaluation: The system was tested using a FastAPI client to validate latency and the CockroachDB dashboard for fault tolerance. As summarized in Performance metrics, the architecture met or exceeded all defined targets.

Performance Analysis

- **Read Latency and Caching:** The system demonstrated exceptional performance for local queries. As seen in metrics, reads for a user in the US-East region were served from the CockroachDB block cache in just 4ms, outperforming the 50ms target. This validates the efficiency of the region-aware routing strategy.

- **Cross-Region Access:** Global queries respected physical network constraints. A request from us-east to ap-south took 1.31s. This latency is expected due to physical distance and confirms that geohashing logic correctly identified the remote partitioning rather than returning empty results.

- **Fault Tolerance:** The system demonstrated self-healing capabilities during the kill test. As documented, Docker Swarm detected a node failure and automatically provisioned a replacement container. The service was instantly restored, making it difficult for the end user to know that some services had gone down.

5. Summary

5.1 Conclusion

In conclusion, this project gives us hands-on experience in developing a distributed database system for ride-sharing. By using the techniques mentioned in this report, we learned how modern systems, like Uber or Lyft, handle data on a global scale.

Additionally, many of the techniques we utilized in this project apply to distributed systems beyond just ride-sharing. Thus, we gained practical insight into the complexity of distributed systems as a whole. All in all, we have a working prototype that demonstrates various distributed database concepts.

5.2 Future Work

1. **Dynamic Region Scaling:** Implement automatic shard splitting if a region gets overloaded

2. **Cross-Region Transactions:** Add support for transactions spanning multiple regions

3. **Cache System:** Implement a Cache system, such as an LRU, for frequently accessed read-only data

4. **Data Security:** Implement encryption and a data proxy to maintain access control on important data variables

5.3 YouTube Link

https://youtu.be/qZhQthkY2vI

References

Amazon.com, 2025, docs.aws.amazon.com/ec2/. Accessed 2 Dec. 2025.

“Docker Swarm.” *Docker Documentation*, 2024, docs.docker.com/reference/cli/docker/swarm/. Accessed 27 Oct. 2025.

“FastAPI.” *Tiangolo.com*, 2025, fastapi.tiangolo.com/. Accessed 5 Oct. 2025.

“Welcome to Faker’s Documentation! — Faker 37.8.0 Documentation.” *Readthedocs.io*, 2016, faker.readthedocs.io/en/master/. Accessed 5 Oct. 2025.