Design Uber - System Design Interview

ASHISH PRATAP SINGH
FEB 27, 2025 · PAID

59 7 Share

The concept of **ride-hailing** has transformed how we travel. Platforms like **Uber**, **L**
and **Ola** seamlessly connect riders with drivers through intuitive smartphone apps.

By simply entering a destination and tapping a button, users can summon a nearby vehicle and monitor its arrival in real time.

However, building such a service at scale involves more than just connecting driver and riders. Behind every "Request Ride" tap lies a sophisticated system coordinatin **real-time driver matching**, **efficiently finding nearby drivers**, **high-throughput da processing**, **dynamic pricing**, and **payment workflows**.

In this article, we will explore how to **design an Uber-like system** that can handle millions of rides every day.

We'll walk through every step of the design—from **requirements** and **high-level architecture** to **database** and **API design**. Finally, we'll take a deep dive into **core u: cases** like how to efficiently find nearby drivers.

1. Requirement Gathering

Before diving into the design, lets outline the functional and non-functional requirements.

Functional Requirements:

1. **Ride requests:** Riders should be able to input their pickup and destination locations and request a ride.
2. **ETA/Fare Estimation:** The system should provide an estimated time of arrival (ETA) and estimated fare to riders before they confirm the booking.
3. **Driver-rider matching:** The system should match riders with available drivers who are in close proximity.
4. **Accept/Decline**: Drivers should be able to accept or decline incoming ride requests.
5. **Driver tracking**: Once a rider is matched with a driver, the rider should be able track the driver's location and view the estimated time of arrival (ETA).
6. **Ratings**: Both riders and drivers should have the ability to rate each other after ride is completed.
7. **Payments**: The user should be able to complete the payment after the ride is completed.

Non-Functional Requirements:

1. **Low latency**: The system should provide real-time location updates and fast driver-rider matching.
2. **High availability**: The system should be up 24/7 with minimal downtime.
3. **Scalability**: The system must handle peak loads (e.g., New Year's Eve, sporting events).

2. Capacity Estimation

Assumptions

- **Total Users:** 50 million riders, 5 million drivers
- **Daily Active Users (DAU):** 10 million riders, 1 million drivers
- **Peak concurrent users**: 1 million riders, ~100,000 drivers (assuming 10% of DAU are active at peak hours)
- **Average Daily Ride Requests:** 10 million globally
- **Peak rides per second (RPS):** ~5,000

Location Updates

- A driver sends a location update **every 3 seconds** while active.
- Assuming **100,000 active drivers** at peak time:
    - **Location updates per second**: 100,000 / 3 ≈ 33,333 updates/sec

Data Storage Estimation

User & Driver Profiles

- **Rider profile**: ~2 KB per user (name, email, phone, payment method, preferenc
- **Driver profile**: ~5 KB per driver (vehicle details, license, payment details, ratin

- **Total storage for 50M users:** (50M × 2 KB) + (5M × 5 KB) = (**100** + **25**) **GB** = **125 (**

Ride Data

Each ride stores:

- Ride ID (UUID) → 16 bytes
- Rider ID, Driver ID → 8 bytes each
- Start & end location (lat/lon) → 16 bytes
- Fare, pickup/dropoff time → 24 bytes
- Status → 8 bytes

Total ride entry size: ~80 bytes

- **Total daily rides:** 10M
- **Storage per day:** 10M × 80 Bytes = **800 MB**
- **Storage per year** (**365 days**): ~**300 GB**

Network Bandwidth Estimation

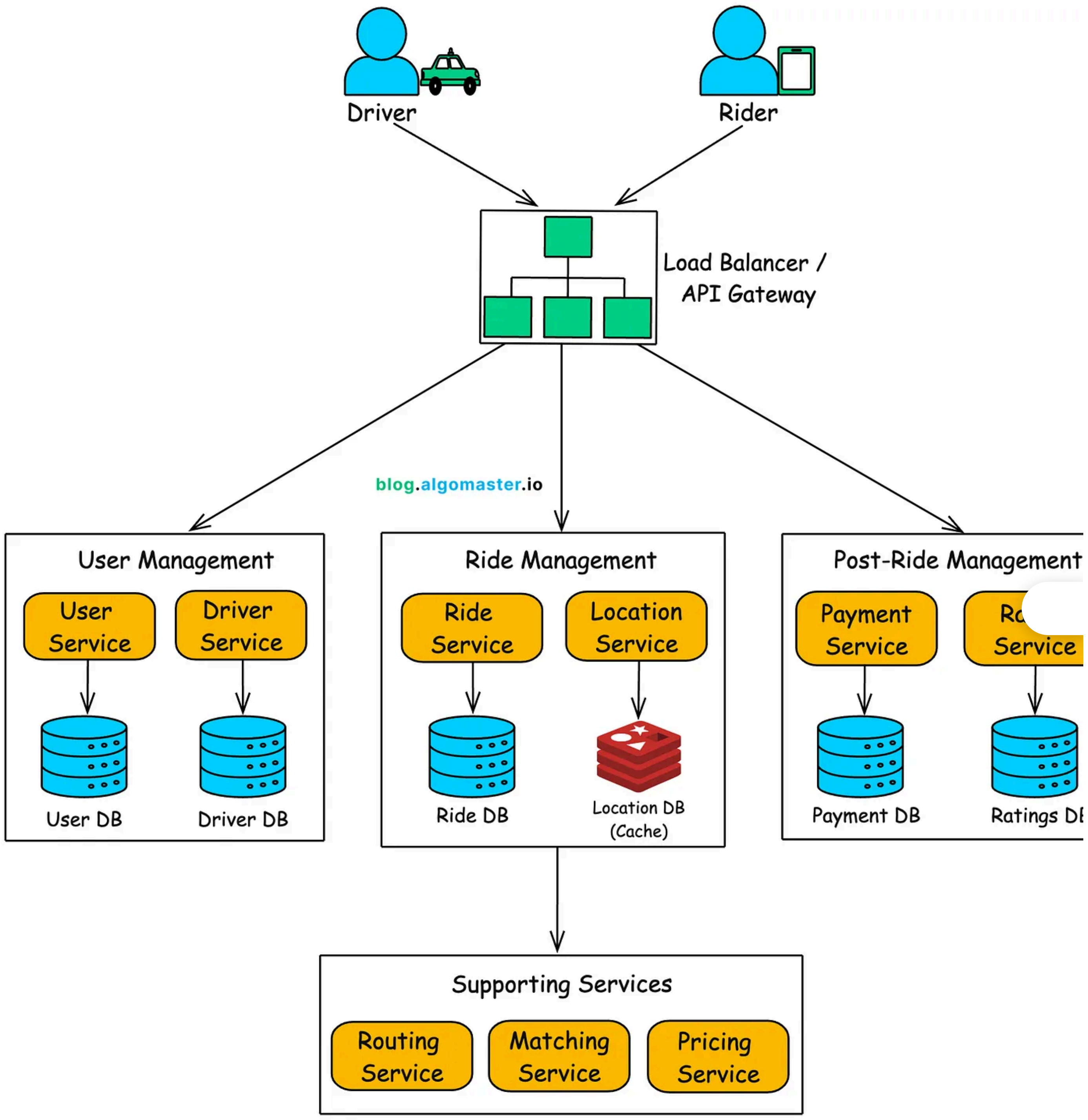
Each API call (ride request, driver update, fare estimation, etc.) contributes to netw usage.

- **Ride requests per second**: ~5,000 RPS
- **Driver location updates per second**: ~33,333 RPS
- **Total peak API requests:** ~40,000 RPS

Assuming an **average API payload size** of **5 KB**, network bandwidth usage at peak:

- **40,000 RPS** × **5 KB** = **200 MB/sec**

3. High-Level Design



Here's a breakdown of the key components:

1. Client Applications (Rider & Driver)

These are the primary user-facing interfaces for interacting with the system.

- Riders can **request rides**, **track drivers in real-time**, **make payments**, and **rate drivers**.
- Drivers can **accept or decline ride requests**, **update their availability**, and **navigate routes**.

- The applications communicate with the backend via an **API Gateway** using **HTTPS REST or gRPC**.

2. API Gateway / Load Balancer

The **API Gateway** serves as the central entry point for all external requests.

It handles:

- **Request Routing**: Receives requests from client applications and directs them the appropriate microservices.
- **Load Balancing**: Distributes traffic across multiple instances of a service to prevent overloading.
- **Security**: Performs authentication (token validation) and authorization.

3. Core Microservices

The system can be divided into three major service groups: **User Management, Rid Management, and Post-Ride Management**.

1. User Management

Responsible for managing riders and drivers.

- **Rider Service:**
    - Manages rider accounts (registration, login, and profile updates).

- Stores user preferences (default payment methods, favorite locations).

- **Driver Service:**
  - Tracks driver availability (online/offline status).
  - Maintains vehicle details (make, model, license plate).

2. Ride Management

Handles the **end-to-end lifecycle of a ride**, from request to completion.

- **Ride Service:**
  - Manages ride creation and updates ride status (requested → driver assigned in-progress → completed).
  - Coordinates with other services (Matching Service, Routing Service, Payme Service).
- **Matching Service:**
  - Finds the nearest available driver(s) for a ride request.
  - If a driver declines, it searches for the next best match.
  - Queries the **Location Service** or a **geo-indexed datastore** to fetch nearby drivers.
  - Updates the **Ride Service** with the assigned driver.
- **Location Service:**
  - Stores real-time driver locations in an in-memory datastore (e.g., **Redis**, NoSQL with geospatial indexing).
  - Receives **frequent location updates** from drivers (every **3 seconds**).
  - Supports **driver tracking** for riders and helps the **Matching Service** find th closest drivers.
- **Routing Service:**

  - Calculates **optimal routes, estimated time of arrival** (ETA), **and turn-by-tu directions.**
  - Uses external **map APIs** (e.g., **Google Maps, Mapbox**) or an **internally host geospatial system.**
- **Pricing Service:**
  - Computes ride fares based on **distance, time, and surge pricing.**
  - Works with the **Ride Service** to provide **real-time fare estimates** and **finali the trip cost.**

3. Post-Ride Management

Handles payments and ratings after the ride is completed.

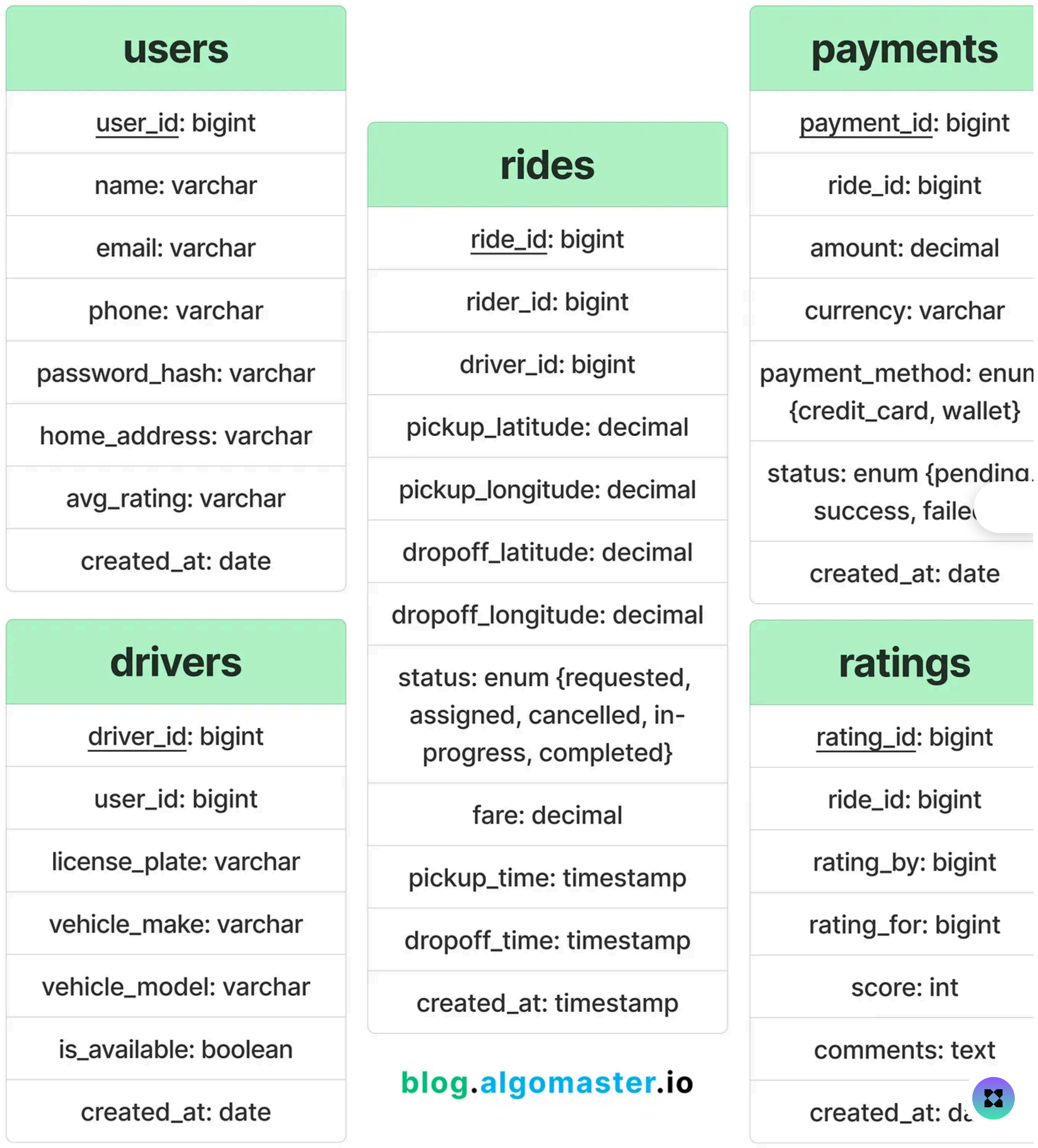
- **Payment Service:**
  - Processes transactions and **stores ride payment history** in a SQL database.
  - Integrates with external payment providers (e.g., **Stripe, PayPal**) for **cre card or digital wallet transactions.**
- **Rating Service:**
  - Allows both riders and drivers to **rate each other** after a ride.

4. Database Design

The system requires a combination of **SQL** (**relational database**) for structured transactional data and **NoSQL** (**document-based or key-value stores**) for high-volu real-time operations.

4.1 SQL Database (Transactional Data)

A **relational database** (e.g., **PostgreSQL**, **MySQL**) is used for **structured, transactio data** that requires strong consistency. This includes user accounts, ride history, payments, and ratings.



users
user_id: bigint
name: varchar
email: varchar
phone: varchar
password_hash: varchar
home_address: varchar
avg_rating: varchar
created_at: date

drivers
driver_id: bigint
user_id: bigint
license_plate: varchar
vehicle_make: varchar
vehicle_model: varchar
is_available: boolean
created_at: date

rides
ride_id: bigint
rider_id: bigint
driver_id: bigint
pickup_latitude: decimal
pickup_longitude: decimal
dropoff_latitude: decimal
dropoff_longitude: decimal
status: enum {requested, assigned, cancelled, in-progress, completed}
fare: decimal
pickup_time: timestamp
dropoff_time: timestamp
created_at: timestamp

payments
payment_id: bigint
ride_id: bigint
amount: decimal
currency: varchar
payment_method: enun {credit_card, wallet}
status: enum {pending, success, faile
created_at: date

ratings
rating_id: bigint
ride_id: bigint
rating_by: bigint
rating_for: bigint
score: int
comments: text
created_at: da

4.2 NoSQL Database (High-volume, Real-time Data)

A **document-based NoSQL database** (**e.g.**, **MongoDB**, **DynamoDB**) is used for high frequency, real-time data that doesn't require strict consistency.

Driver Location Collection

Stores frequent driver location updates for real-time operations.

Example Document:

```
{
  "driver_id": "d456",
  "location": {
    "latitude": 37.7749,
    "longitude": -122.4194
  },
  "timestamp": "2025-02-27T12:34:56Z"
}
```

Index:

- **driver_id**: For quick lookup of a particular driver's last known location.
- **Geospatial index** on `location` to query by proximity (e.g., find drivers within km of a rider).

A **high-performance caching layer** (**e.g.**, **Redis**, **Memcached**) can be used to reduce database load and improve response times.

We will discuss more approaches to support **nearby driver queries** in the deep dive later.

5. API Design

Below is a **RESTful API design** that aligns with the microservices architecture described earlier.

The APIs are grouped based on the microservice responsible for handling their log and data.

- **Public APIs**: Exposed to client applications (rider and driver apps).
- **Internal APIs:** Used for inter-service communication, not directly accessible by clients.

Here we will only cover the APIs that are most relevant to this problem.

5.1 Driver Service (Public)

Update Driver Location

```
PUT /drivers/{driverId}/location
Authorization: Bearer <accessToken>
Content-Type: application/json

{
  "latitude": 37.7749,
  "longitude": -122.4194,
  "timestamp": "2025-02-27T12:34:56Z"
}
```

5.2 Ride Service (Public)

Request a Ride

```
POST /rides
Authorization: Bearer <accessToken>
Content-Type: application/json

{
  "riderId": "u123",
  "pickupLocation": {
    "latitude": 37.7749,
    "longitude": -122.4194
  },
  "dropoffLocation": {
    "latitude": 37.7849,
    "longitude": -122.4094
  }
}
```

Get Ride Details

```
GET /rides/{rideId}
Authorization: Bearer <accessToken>
```

Response:

```
{
  "rideId": "r789",
  "riderId": "u123",
  "driverId": "d456",
  "status": "DRIVER_ASSIGNED",
  "currentDriverLocation": { "latitude": 37.7760, "longitude": -122.41
},
  "estimatedArrivalTime": "3 minutes",
  "fare": {
    "estimate": 15.25,
    "final": null
  }
}
```

5.3 Matching Service (Internal)

Responsible for finding the nearest available driver. Often called by the **Ride Servic**

Find Nearest Driver

```
POST /match/nearest
Content-Type: application/json

{
  "rideId": "r789",
  "pickupLocation": {
    "latitude": 37.7749,
    "longitude": -122.4194
  }
}
```

Response:

```
{
  "driverId": "d456",
  "distance": 1.2,
  "eta": "5 minutes"
}
```

Assign Driver

```
POST /match/assign
Content-Type: application/json
```

```
{
  "rideId": "r789",
  "driverId": "d456"
}
```

Response:

```
{
  "rideId": "r789",
  "driverId": "d456",
  "status": "DRIVER_ASSIGNED"
}
```

5.4 Location Service (Public)

Receive Driver Location Update (High-frequency endpoint)

```
POST /location/driver
Content-Type: application/json

{
  "driverId": "d456",
  "latitude": 37.7749,
  "longitude": -122.4194,
  "timestamp": "2025-02-27T12:34:56Z"
}
```

Get Driver Location

```
GET /location/driver/{driverId}
```

Response:

```
{
  "driverId": "d456",
  "latitude": 37.7749,
  "longitude": -122.4194,
  "timestamp": "2025-02-27T12:34:56Z"
}
```

5.5 Routing Service (Internal)

Computes optimal routes and ETAs. Generally invoked by the **Ride Service** or **Matching Service.**

Get Route & ETA

```
POST /routing/eta
Content-Type: application/json

{
  "startLocation": { "latitude": 37.7749, "longitude": -122.4194 },
  "endLocation": { "latitude": 37.7849, "longitude": -122.4094 }
}
```

Response:

```
{
  "distance": 2.5,
  "eta": "5 minutes",
  "directions": [
    "Head north on Market St",
    "Turn right on 5th St",
    "Continue until ...",
    "... (etc.)"
  ]
}
```

5.6 Pricing Service (Internal)

Calculates fare estimates, including surge pricing or dynamic pricing based on sup demand.

Get Fare Estimate

```
POST /pricing/estimate
Content-Type: application/json

{
  "distance": 2.5,
  "timeInMinutes": 10,
  "surgeMultiplier": 1.5
}
```

Response:

```
{
  "baseFare": 10.00,
  "surgeMultiplier": 1.5,
  "estimatedFare": 15.00,
  "currency": "USD"
}
```

5.7 Rating Service (Public)

Submit Rating

```
POST /ratings
Authorization: Bearer <accessToken>
Content-Type: application/json

{
  "rideId": "r789",
  "ratingBy": "u123",        // or "d456"
  "ratingFor": "d456",       // or "u123"
  "score": 5,
  "comments": "Excellent ride!"
}
```

Get Ratings for a User

```
GET /ratings/user/{userId}
Authorization: Bearer <accessToken>
```

Response:

```
{
  "userId": "d456",
  "averageRating": 4.85,
  "totalRidesRated": 1200
}
```

API Considerations

1. **Authentication & Security**
    - Each API call should require a valid **access token** (e.g., JWT) in the `Authorization: Bearer <token>` header.
2. **Response Codes & Error Handling**

   - Use standard HTTP status codes (e.g., `200 OK`, `201 Created`, `400 Bad Request`, `401 Unauthorized`, `404 Not Found`, `500 Internal Ser Error`).
   - Include error messages and error codes in JSON responses for clarity.
3. **Pagination & Filtering**
   - For queries like `GET /rides` (to list all rides for a user) or `GET /ratings` implement pagination (`page`, `limit`) and possible filters (date ranges, ratir thresholds).
4. **Internal vs. External Endpoints**
   - Many endpoints (Matching, Routing, Pricing) are typically internal microservice APIs, not exposed directly to mobile/web clients.
   - The **API Gateway** ensures that only relevant endpoints (e.g., `POST /rides` `GET /rides/{id}`, `POST /payments`) are accessible externally.
5. **Rate Limiting:** Public APIs are protected with **rate limits** to prevent abuse.
6. **Event-Driven Triggers**
   - Certain workflows (e.g., "ride completed" → "send rating request" or "matching succeeded" → "notify ride service") can be handled asynchronou via a **Message Queue** (Kafka, RabbitMQ, etc.).

6. Deep Dive into Core Use Cases

Lets dive into the core use cases that the system need to support.

6.1 Booking a Ride

Rider Initiates a Ride Request

- The rider opens the app, enters pickup and destination locations, and taps a "Request Ride" button.
- The request (including the rider's ID and locations) is sent to the **API Gateway** HTTPS.
- The API Gateway authenticates the request (validates the rider's JWT token, et and forwards it to the **Ride Service**.

Ride Service Creates a New Ride

- **Creates a new ride record** in the **SQL database** with status set to something lik `REQUESTED`.
- Calls the **Pricing Service** to get a fare estimate and calls the **Routing Service** to get the ETA before finalizing the request.

Matching Service Locates an Available Driver

- Once the rider confirm the request, the Ride Service invokes the **Matching Service**. It passes the ride's pickup coordinates and other details (rider ID, ride etc.).
- Matching Service queries the **Location Service** (or a geo-enabled data store) to find online drivers within a certain radius of the pickup location.
- **Sorts or ranks** these drivers based on proximity, driver rating, or other busines logic (e.g., driver acceptance rate, predicted demand).
- **Selects** the best candidate (closest and/or most suitable driver).
- Sends a **push notification** to the selected driver's mobile app, requesting acceptance of the ride.
- The driver sees ride details (pickup location, estimated fare) and can accept or decline (within a time limit e.g., 30 seconds).

Driver Accepts (or Declines) the Ride

- If the driver **accepts**, the driver app notifies the Matching Service.
- If the driver **declines**, the Matching Service selects the **next** available driver fro its candidate list.
- Upon driver acceptance, the Matching Service updates the Ride Service with tl assigned driver.
- The Ride Service **updates** the ride record in the database to reflect the assignec driver and new status (`DRIVER_ASSIGNED`).

Rider Receives Confirmation

- The Ride Service notifies the rider (via an app push message) that a driver is assigned.
- The app displays the driver's name, vehicle details, and **real-time ETA** (calculat via the **Routing Service**).

6.2 Finding Nearby Drivers

Finding nearby drivers is a **core use case** in any ride-hailing system. It must be **fast, accurate, and scalable** to handle **millions of concurrent ride requests** efficiently.

1. Naïve Solution: Using a Relational Database (SQL)

A **basic approach** is to store **driver locations in a SQL database** (e.g., **MySQL, PostgreSQL**) and query nearby drivers using **Haversine formula**.

Schema Example (**PostgreSQL**):

```
CREATE TABLE drivers (
    driver_id VARCHAR(50) PRIMARY KEY,
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6),
    status ENUM('available', 'busy')
);
```

Query: Finding Drivers within 5 km (Using Haversine Formula)

```
SELECT driver_id, latitude, longitude,
       (6371 * acos(cos(radians(37.7749)) * cos(radians(latitude))
       * cos(radians(longitude) - radians(-122.4194))
       + sin(radians(37.7749)) * sin(radians(latitude)))) AS distance
FROM drivers
WHERE status = 'available'
HAVING distance <= 5
ORDER BY distance ASC
LIMIT 10;
```

✅ **Pros:**

- Easy to implement.
- Works for small-scale applications.

❌ **Cons:**

- **Slow for large datasets** (full table scan every time).
- **Not scalable** for millions of requests.
- **No spatial indexing**, leading to high latency.

2. Using Database Extensions like PostGIS

PostGIS is a spatial database extension for PostgreSQL, allowing **geospatial indexi**

```
CREATE TABLE drivers (
    driver_id VARCHAR(50) PRIMARY KEY,
    location GEOGRAPHY(POINT, 4326),
    status ENUM('available', 'busy')
);

CREATE INDEX location_idx ON drivers USING GIST(location);
```

Query: Finding Nearby Drivers Using PostGIS

```
SELECT driver_id, ST_DistanceSphere(location, ST_MakePoint(-122.4194,
37.7749)) AS distance
FROM drivers
WHERE status = 'available'
AND ST_DWithin(location, ST_MakePoint(-122.4194, 37.7749)::GEOGRAPHY,
5000) -- 5 km radius
ORDER BY distance ASC
LIMIT 10;
```

✅ Pros:

- **Faster than naive SQL** due to **geospatial indexing**.
- **More accurate results** than naive SQL.

❌ Cons:

- **Still not ideal for large-scale applications** (queries slow down with millions of users).
- **Indexes require frequent updates**, which is expensive in real-time applications

3. Geohashing (Efficient Grid-Based Search)

Geohashing converts **latitude & longitude** into a **string representation** by **div**

the world into hierarchical grids. Nearby locations have **similar geohashes**, al

efficient lookups.

How Geohashing Works:

1. The world is **divided into a grid** with cells of different sizes.
2. Each **latitude/longitude pair is converted into a unique hash string** (e.g., `"9q9hv" for San Francisco`).
3. To find **nearby drivers**, search for drivers in the **same or adjacent geohash regions**.

Example: Storing Driver Locations in Redis (Key-Value Store)

```
GEOADD drivers -122.4194 37.7749 "driver_123"
```

Query: Finding Drivers in a 5 km Radius

```
GEORADIUS drivers -122.4194 37.7749 5 km WITHDIST
```

✅ **Pros:**

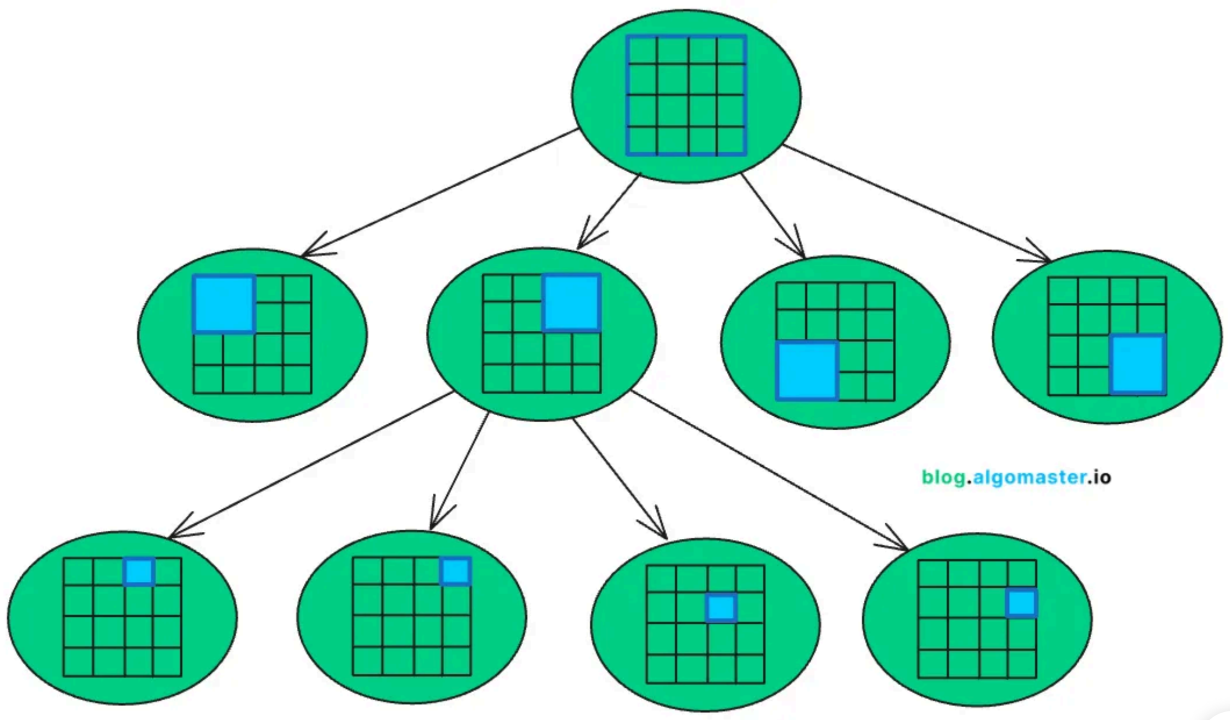
- **Extremely fast lookups** (O(log n) complexity).
- **Efficient for large-scale systems.**
- **Easily scalable** with distributed databases.

❌ **Cons:**

- **Geohashing regions are not circular** (they are rectangular), leading to **false positives**.
- **Handling edge cases** (drivers near region boundaries require extra querie

4. Quadtree (Hierarchical Spatial Indexing)

A **quadtree** is a **tree-based spatial data structure** that recursively divides a 2D spac into **quadrants**.



How Quadtree Works

1. The **map is recursively divided** into **four quadrants**.
2. Each **driver's location is stored** in the smallest quadrant possible.
3. To **find nearby drivers, search within relevant quadrants**.

Example: Querying a Quadtree for Nearby Drivers

```
quadtree.find_nearby(lat=37.7749, lon=-122.4194, radius=5000)
```

✅ **Pros:**

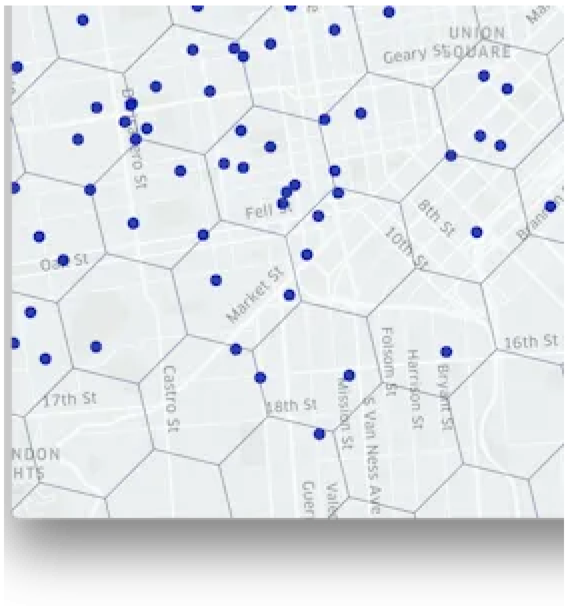
- Efficient **logarithmic search time** (O(log n)).
- Good **spatial accuracy** compared to geohashing.

❌ Cons:

- More **complex to implement.**
- **Insertion & deletion of drivers** require rebalancing the quadtree.

5. What Does Uber Actually Use?

Uber originally used **Geohashing**, but later switched to **H3: Hexagonal Hierarchica Spatial Index.**



source: https://www.uber.com/en-IN/blog/h3/

What is H3?

H3 is an **open-source geospatial indexing system** developed by Uber.

- It **divides the world into hexagonal cells** instead of squares (used in Geohashin
- Each hexagon has **better spatial coverage** (less overlap, more uniform).
- H3 supports **hierarchical indexing**, allowing **efficient nearby searches.**

Why Hexagons?

- **Uniform coverage** (less distortion than squares).

- **Efficient neighbor searches** (hexagons naturally fit together).
- **Scales well** for high-volume ride-hailing applications.

How Uber Finds Nearby Drivers Using H3

1. Convert each **driver's location into an H3 hexagon ID**.
2. Store **drivers in a distributed key-value store** (e.g., **Cassandra, Redis**).
3. To find nearby drivers:
   - Retrieve **drivers in the same hexagon as user**.
   - Expand search **one hexagon outward** until enough drivers are found.

```
import h3

hex_id = h3.geo_to_h3(37.7749, -122.4194, resolution=7)
nearby_hexes = h3.k_ring(hex_id, 1)  # Find drivers in nearby hexes
```

✅ **Pros:**

- **Highly scalable** (used in production by Uber, Google, and others).
- **Fast nearest-neighbor lookups.**
- **More accurate than Geohashing** (hexagons cover space better).
- **Optimized for distributed storage.**

❌ **Cons:**

- **Requires a dedicated infrastructure** (custom storage solutions).
- **More complex than traditional geospatial databases.**

6.3 Real-Time Tracking

Driver Location Updates

- Driver's app **captures GPS coordinates** (latitude & longitude) every **3 seconds**.
- The app **sends location data** to the **Location Service** via an API call.
- The **Location Service** validates and **stores the latest position** in an in-memory datastore (e.g., **Redis, DynamoDB**).
- Since driver locations **frequently change**, we use a **fast, ephemeral storage syst** like **Redis** with **geospatial indexing**.

Rider Tracking the Driver's Location

There are two ways for the rider's app to receive real-time updates:

- **Polling** (**API Calls Every Few Seconds**)
  - Simple to implement
  - High network usage, increased server load
- **WebSockets** (**Push Updates**)
  - Low latency and efficient
  - More complex implementation

Recommendation: Use WebSockets for real-time tracking while allowing polling a fallback.

When a ride starts, the Rider app opens a WebSocket connection to receive live location updates.

- Rider app subscribes to driver's location updates via WebSocket.
- The Location Service sends live updates every 3 seconds.
- The rider's app renders the driver's position on the map dynamically.

6.4 ETA Computation (Estimated Time of Arrival)

ETA is calculated for **two parts** of the trip:

1. **Pickup ETA**: Time for the nearest available driver to reach the rider's pickup location.
2. **Drop-off ETA**: Time to travel from pickup to destination.

Find the closest available drivers

- Query the **Location Service** (or a geospatial index) to get nearby **active drivers**.
- The Matching Service may pre-filter drivers based on availability, ratings, and
cancellations.

Determine the estimated pickup time

- Use the **Routing Service** to compute the time for each nearby driver to rea
rider.
- Consider **real-time traffic conditions** when calculating time.
- Select the **driver with the shortest ETA** (or apply business logic to prioritize).

Calculate the estimated drop-off time

- The Routing Service computes the optimal route **from pickup to drop-off location.**
- Adjust travel time based on **historical trip data and real-time traffic updates.**

ETA Recalculation

- Periodically, the Ride Service may call the **Routing Service** to recalculate ETAs based on traffic or route changes.
- This updated ETA is then shown to the rider.

6.5 Fare Estimation

The **Pricing Service** calculates the estimated fare before booking confirmation. It considers multiple factors.

A common ride fare formula follows this structure:

```
Total Fare = Base Fare + (Cost per km × Distance)+
(Cost per minute × Time) + Surge Multiplier + Tolls
```

Where:

- **Base Fare** → Fixed charge to start a ride (e.g., $2.00).
- **Cost per km** → Charge per unit distance traveled.
- **Cost per minute** → Charge for time spent in traffic or waiting.
- **Surge Pricing** → Additional multiplier when demand is high.
- **Tolls & Taxes** → Additional charges based on location.

Steps for Fare Estimation

1. **Retrieve distance and time from the Routing Service**
   - Compute the trip's estimated **distance** (km) and **time** (minutes).
2. **Check for surge pricing**
   - The Pricing Service checks **real-time demand vs. supply**.
   - If demand is high (e.g., **rainy weather, peak hours**), a **surge multiplier** (e.g., **1.5x, 2x**) is applied.
3. **Apply pricing formula**
   - Use the **city-specific rate card** for cost-per-km and cost-per-minute va
   - Factor in tolls if applicable.
4. **Return estimated fare**

    - Provide a **price range** (e.g., $12 - $15) to account for **traffic fluctuations**.

6.6 Handling Payments Post-Ride

Ride Completion

- When the ride status changes to `COMPLETED` (usually via the driver's app), the **Ride Service** updates the ride record in the SQL database.
- The rider is notified via the app that the ride has ended and a payment will be processed.

Final Fare Calculation

- The **Ride Service** retrieves essential data points: distance traveled, time in tran surge multiplier
- The **Ride Service** calls the **Pricing Service** with updated metrics to **calculate th final fare**.
- The **Ride Service** sets `fare` in the ride record, ensuring the final fare is now locked in for payment processing.

Payment Authorization & Processing

- The **Ride Service** or an **Event-Driven** workflow (e.g., a message queue event "RideCompleted") triggers the **Payment Service** to handle the financial transaction.
- The Payment Service looks up the rider's **default payment method** (credit card, digital wallet, etc.) in the **User Service** or a secure vault.
- If no valid payment method exists, the system may prompt the rider to update payment information.
- The Payment Service **initiates a charge** via an **external payment gateway** Stripe, PayPal).

- The payment gateway processes the charge asynchronously. Once the payment successful (or fails), it **triggers a webhook event** to notify Payment Service.
- Payment Service updates the **Payment Status** in the database (`PENDING → SUCCESS/FAILED`).

Thank you for reading!

If you found it valuable, hit a like ❤️ and if you have any questions or suggestions, leave a comment.

I hope you have a lovely day!

See you soon,

Ashish

59 Likes · 7 Restacks

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...