Q1:-

Backend System for Uber's Ride Hailing Service

1. User Authentication and Authorization

Uber's ride-hailing service requires robust user authentication and authorization mechanisms to ensure secure access to the platform. Key components include:

- **Authentication**: Users can sign up and log in using email/password or social media accounts.
- **Authorization**: Different roles are assigned to users (drivers, riders, admins), each with specific permissions.
- **Token-Based Authentication:** JSON Web Tokens (JWT) can be used for stateless authentication, allowing users to access protected resources by presenting a token.

Code Snippets:-

- **Authentication:-**

```
// Middleware for JWT authentication
const jwt = require('jsonwebtoken');

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];
  if (token == null) return res.sendStatus(401);

  jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
```

```
};

module.exports = authenticateToken;
```

- **Authorization:-**

```
// Middleware for role-based access control
const checkRole = (role) => {
  return (req, res, next) => {
    if (req.user && req.user.role === role) {
      next();
    } else {
      res.status(403).send('Unauthorized');
    }
  };
};

module.exports = checkRole;
```

2. Geo location and Mapping

Uber uses a third-party map service provider to build the map in their application. Earlier Uber was using Mapbox services but later Uber switched to Google Maps API to track the location and calculate ETAs.

2.1 Real-Time Tracking of Drivers and Riders

Uber employs real-time tracking using a combination of GPS technology and mapping services. The system continuously updates the location of both drivers and riders, facilitating efficient matching and navigation.

- **GPS Integration:** Mobile apps use GPS to retrieve the current location of drivers and riders.
- **WebSocket Communication:** Real-time updates are sent to the server using WebSocket connections, allowing for instant location updates.
- **Map Integration:** Mapping services like Google Maps or Mapbox display the real-time location of drivers and riders on the user interface.

2.2 Efficient Calculation of ETA

ETA is an extremely important metric in Uber because it directly impacts ride-matching and earnings.

- ETA is calculated based on the road system (not geographically) and there are a lot of factors involved in computing the ETA (like heavy traffic or road construction).
- When a rider requests a cab from a location the app not only identifies the free/idle cabs but also includes the cabs which are about to finish a ride.
- We can represent the entire road network on a graph to calculate the ETAs. We can use AI-simulated algorithms or simple Dijikstra's Algorithm to find out the best route in this graph.
- In that graph, nodes represent intersections (available cabs), and edges represent road segments.
- We represent the road segment distance or the traveling time through the edge weight. We also represent and model some additional factors in our graph such as one-way streets, turn costs, turn restrictions, and speed limits.

3. Ride Matching and Dispatching

3.1 Algorithm for Matching Riders with Drivers

Uber's matching algorithm considers multiple factors to pair riders with available drivers efficiently:

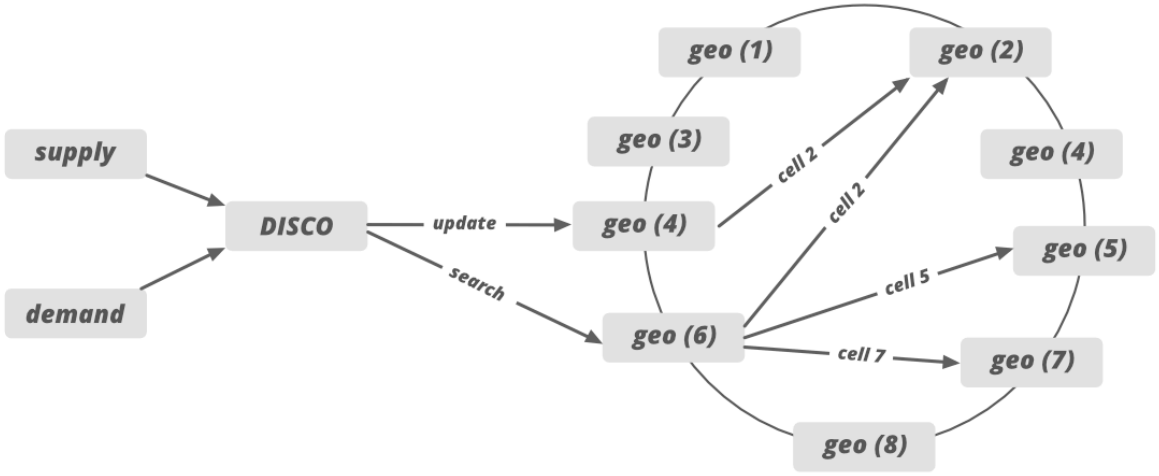
- **Proximity:** Riders are matched with the nearest available drivers to minimize wait times.
- **Traffic Conditions:** The algorithm accounts for traffic congestion and route efficiency when selecting drivers.
- **Driver Ratings:** Riders may be matched with drivers based on their ratings and feedback to ensure a positive experience.

```
// Sample code for ride matching algorithm
const findNearestDriver = (userLocation, availableDrivers) => {
  // Calculate distance between userLocation and each driver's location
  // Return the nearest driver
};
```

3.2 Handling Ride Requests During Peak Hours

During peak hours, when demand for rides is high, Uber employs strategies to manage ride requests effectively:

- **Surge Pricing:** Dynamic pricing encourages more drivers to be on the road during peak hours, balancing supply and demand.
- **Priority Dispatching:** The system prioritizes ride requests based on factors like distance and driver availability to minimize wait times.
- **Driver Incentives:** Incentives are offered to drivers to encourage them to accept rides during peak hours, increasing the availability of drivers.



4. Real-Time Updates and Notifications

Uber provides real-time updates and notifications to keep users informed throughout the ride-hailing process:

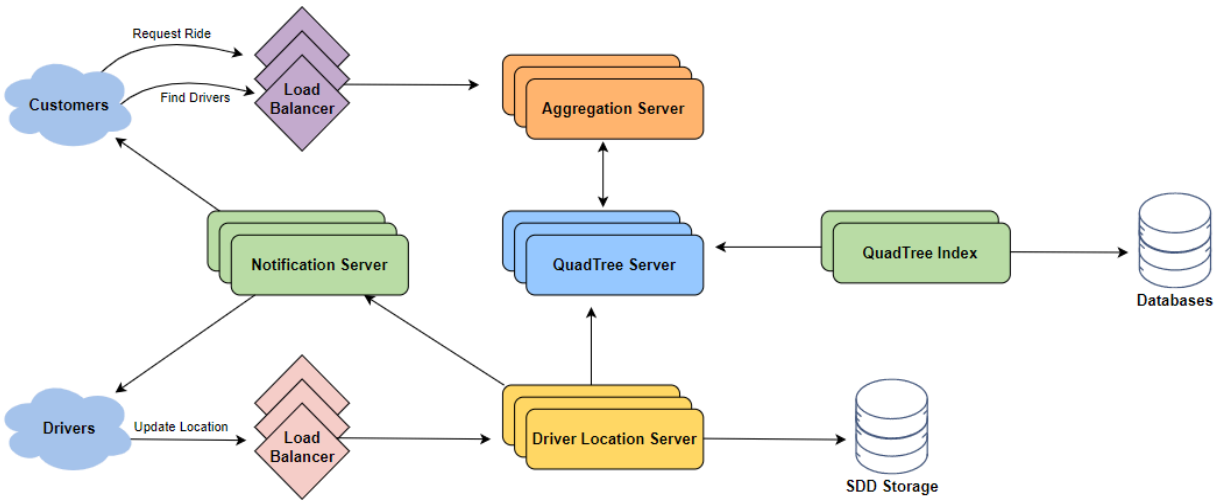
- **Push Notifications:** Mobile apps send push notifications to users to notify them of ride status updates, such as driver arrival and trip completion.
- **SMS Notifications:** Users without the mobile app receive SMS notifications with ride details and updates.
- **In-App Messaging:** Riders and drivers can communicate directly within the app for additional updates or instructions.

Notification Service:

To efficiently implement the Notification service, we can either use HTTP long polling or push notifications. Customers are subscribed to nearby drivers when they open the Uber app for the first time.

When new drivers enter their areas, we need to add a new customer/driver subscription dynamically. To do so, we track the area that a customer is watching. However, this would get extra complicated.

Instead of pushing this information, we can design the system so customers pull the information from the server. Customers will send their current location so that the server can find nearby drivers from our QuadTree. The customer can then update their screen to reflect drivers' current positions.



5. Payment Processing

5.1 Payment Processing System

Uber's payment processing system ensures secure and seamless transactions between riders and drivers:

- **Payment Gateways:** Integration with payment gateways like Stripe or Braintree enables secure processing of credit/debit card payments.
- **Fare Calculation:** Fares are calculated based on factors such as distance, time, and surge pricing adjustments.
- **Handling Refunds:** The system handles refunds for canceled rides or disputed transactions, ensuring timely resolution and customer satisfaction.

```
// Sample code for fare calculation
const calculateFare = (distance, duration, surgeMultiplier) => {
  // Apply surge pricing
  const baseFare = distance * BASE_RATE_PER_KM + duration *
BASE_RATE_PER_MINUTE;
  return baseFare * surgeMultiplier;
};
```

6. Scalability and Fault Tolerance

Uber's backend system is designed to scale horizontally and withstand failures gracefully:

- **Microservices Architecture:**
  Uber's adoption of a microservices architecture enables the decomposition of the system into smaller, independent services, each responsible for a specific business function. This architecture offers several benefits:

1. **Scalability:** Microservices can be scaled independently based on demand for specific functionalities. This granular scalability allows Uber to allocate resources efficiently and handle varying workloads effectively.
2. **Flexibility:** Each microservice can be developed, deployed, and maintained independently, fostering agility and flexibility in the development process. Teams can work autonomously on their respective services without dependencies on other teams.
3. **Fault Isolation:** Isolating services minimizes the impact of failures. If one microservice experiences an issue, it does not necessarily affect the entire system, reducing downtime and improving resilience.
4. **Technology Diversity:** Different microservices can be implemented using diverse technologies, chosen based on their suitability for specific tasks. This flexibility allows Uber to leverage the strengths of different technologies and frameworks across its ecosystem.

- **Load Balancing:**
  It plays a crucial role in distributing incoming requests across multiple instances of microservices to ensure optimal resource utilization and maintain high availability. Uber employs various load balancing strategies:

1. **Algorithmic Load Balancing:** Load balancers use algorithms (such as round-robin, least connections, or weighted round-robin) to distribute incoming requests evenly

across available instances. This prevents any single instance from becoming overwhelmed with requests, thereby improving overall system performance.

2. **Health Checks:** Load balancers continuously monitor the health and availability of backend instances. If an instance becomes unhealthy or unresponsive, the load balancer automatically routes traffic away from it, ensuring that only healthy instances handle incoming requests.
3. **Session Persistence:** For stateful applications or services that require session persistence, load balancers can be configured to maintain session affinity, ensuring that subsequent requests from the same client are routed to the same backend instance. This is particularly important for maintaining user sessions and ensuring a seamless experience.

- **Fault Tolerance:**
  Uber's backend system incorporates fault tolerance mechanisms to minimize service disruptions and maintain availability, even in the face of failures:

1. **Redundancy:** Critical components of the system are replicated across multiple instances or data centers. This redundancy ensures that if one instance or data center fails, redundant instances can seamlessly take over, minimizing downtime and preventing service disruptions.
2. **Failover Mechanisms:** In the event of a failure, failover mechanisms automatically redirect traffic from the failed component to redundant instances or backup systems. This ensures continuity of service and minimizes the impact on end users.
3. **Circuit Breakers:** Circuit breakers are employed to prevent cascading failures. If a microservice experiences prolonged unresponsiveness or errors, the circuit breaker temporarily halts traffic to that service, allowing it to recover and preventing further degradation of performance.
4. **Graceful Degradation:** Uber's system is designed to gracefully degrade functionality in the face of failures. Non-essential features or functionalities may be temporarily disabled or scaled back to prioritize critical services and maintain overall system stability.

7. Driver and Rider Ratings

Uber's rating system allows riders and drivers to rate each other after each trip, providing valuable feedback for improving service quality:

- **Rating Calculation:** Ratings are calculated based on factors such as driver professionalism, vehicle cleanliness, and rider behavior.

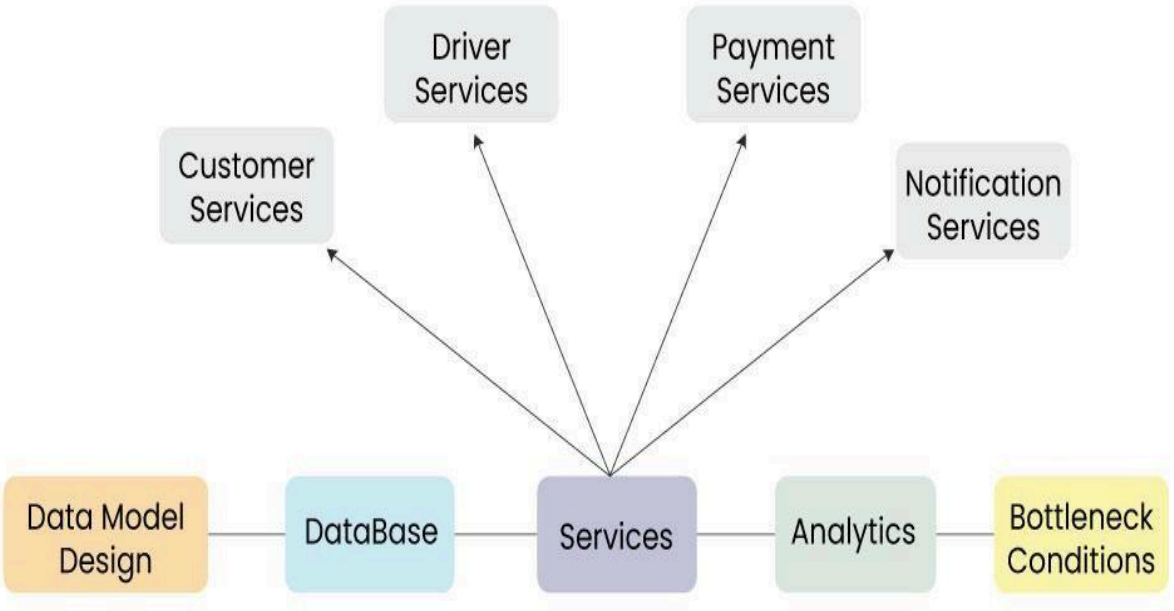
- **Feedback Mechanism:** Users can leave comments along with their ratings to provide specific feedback on their experience.
- **Impact on Service:** Drivers with consistently low ratings may be deactivated from the platform, while riders may be flagged for abusive behavior.

8. Data Security and Privacy

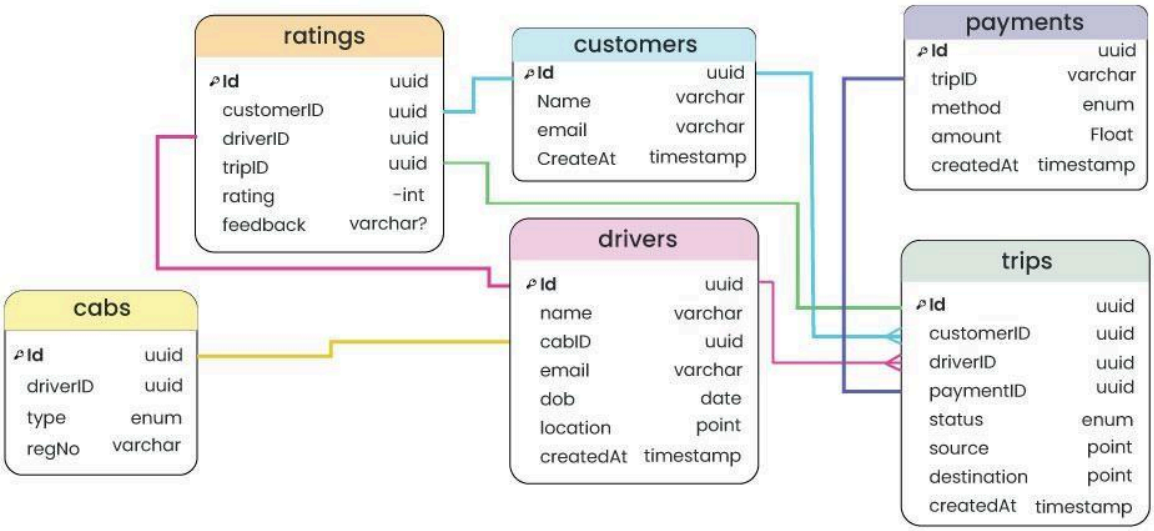
Uber prioritizes data security and privacy to protect user information and ensure compliance with regulations:

- **Encryption:** Sensitive data, such as user credentials and payment information, is encrypted both in transit and at rest to prevent unauthorized access.
- **Access Control:** Role-based access control (RBAC) mechanisms restrict access to data based on user roles and permissions.
- **Compliance:** The system complies with relevant data protection regulations, such as GDPR and CCPA, to safeguard user privacy rights.

9. High-Level Design



9.1 Data model design



9.2 Databases

Uber had to consider some of the requirements for the database for a better customer experience. These requirements are...

- The database should be horizontally scalable. You can linearly add capacity by adding more servers.
- It should be able to handle a lot of reads and writes because once every 4-second cabs will be sending the GPS location and that location will be updated in the database.
- The system should never give downtime for any operation. It should be highly available no matter what operation you perform (expanding storage, backup, when new nodes are added, etc).

Earlier Uber was using the RDBMS PostgreSQL database but due to scalability issues uber switched to various databases. Uber uses a NoSQL database (schemaless) built on top of the MySQL database.

- Redis for both caching and queuing. Some are behind Twemproxy (which provides scalability of the caching layer). Some are behind a custom clustering system.

- Uber uses Schemaless (built in-house on top of MySQL), Riak, and Cassandra. Schemaless is for long-term data storage. Riak and Cassandra meet high-availability, low-latency demands.
- MySQL database.
- Uber is building their own distributed column store that's orchestrating a bunch of MySQL instances.

9.3 Services

- Customer Service: This service handles concerns related to customers such as customer information and authentication.
- Driver Service: This service handles driver-related concerns such as authentication and driver information.
- Payment Service: This service will be responsible for handling payments in our system.
- Notification Service: This service will simply send push notifications to the users. It will be discussed in detail separately.

9.4 Analytics

To optimize the system, minimize the cost of the operation and for better customer experience uber does log collection and analysis. Uber uses different tools and frameworks for analytics. For log analysis, Uber uses multiple Kafka clusters.
Kafka takes historical data along with real-time data. Data is archived into Hadoop before it expires from Kafka. The data is also indexed into an Elastic search stack for searching and visualizations. Elastic search does some log analysis using Kibana/Graphana. Some of the analyses performed by Uber using different tools and frameworks are...

- Track HTTP APIs
- Manage profile
- Collect feedback and ratings
- Promotion and coupons etc
- Fraud detection
- Payment fraud
- Incentive abuse by a driver
- Compromised accounts by hackers. Uber uses historical data of the customer and some machine learning techniques to tackle this problem.

9.5 How To Handle The Data Center Failure?

Datacenter failure doesn't happen very often but Uber still maintains a backup data center to run the trip smoothly. This data center includes all the components but Uber never copies the existing data into the backup data center.
Then how does Uber tackle the data center failure?

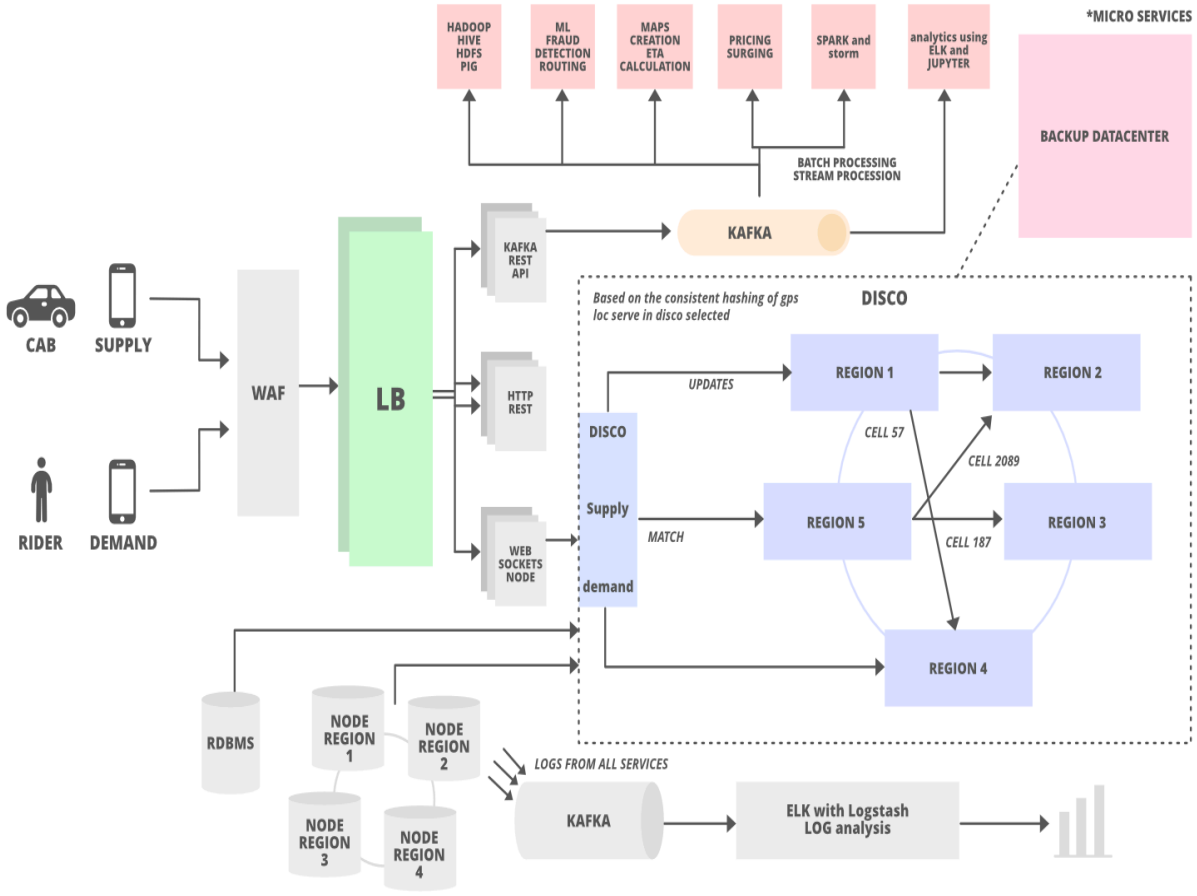
- It actually uses driver phones as a source of trip data to tackle the problem of data center failure.
- When The driver's phone app communicates with the dispatch system or the API call is happening between them, the dispatch system sends the encrypted state digest (to keep track of the latest information/data) to the driver's phone app.
- Every time this state digest will be received by the driver's phone app. In case of a data center failure, the backup data center (backup DISCO) doesn't know anything about the trip so it will ask for the state digest from the driver's phone app and it will update itself with the state digest information received by the driver's phone app.

Additional Features

- **Driver Scheduling:** Allow drivers to set their availability and schedule in advance, improving resource planning and flexibility.
- **Emergency Assistance:** Integration with emergency services and panic buttons within the app for immediate assistance in case of emergencies during rides.
- **Accessibility Features:** Provide options for users with disabilities, such as wheelchair-accessible vehicles and visual impairment support.

Conclusion

The backend system for Uber's ride-hailing service encompasses various components and features to provide a seamless and reliable experience for both riders and drivers. By prioritizing factors such as user authentication, real-time tracking, efficient ride matching, and secure payment processing, Uber ensures a high level of service quality while addressing scalability and fault tolerance challenges. With continuous innovation and improvement, Uber remains at the forefront of the transportation industry, delivering convenience and safety to millions of users worldwide.


HADOOP HIVE HDFS PIG
ML FRAUD DETECTION ROUTING
MAPS CREATION ETA CALCULATION
PRICING SURGING
SPARK and storm
analytics using ELK and JUPYTER
*MICRO SERVICES
BACKUP DATACENTER
BATCH PROCESSING STREAM PROCESSION
KAFKA
KAFKA REST API
CAB
SUPPLY
RIDER
DEMAND
WAF
LB
HTTP REST
WEB SOCKETS NODE
Based on the consistent hashing of gps loc serve in disco selected
DISCO
UPDATES
REGION 1
REGION 2
DISCO
Supply
demand
MATCH
REGION 5
REGION 3
REGION 4
CELL 57
CELL 2089
CELL 187
RDBMS
NODE REGION 1
NODE REGION 2
NODE REGION 3
NODE REGION 4
LOGS FROM ALL SERVICES
KAFKA
ELK with Logstash LOG analysis

Q2:-

Backend System for an E-commerce store

1. Product Catalog and Inventory Management

1.1 Database Schema and Inventory Management

A relational database schema can be designed to manage the product catalog and inventory efficiently:

- **Product Table**: Stores information about each product, including ID, name, description, price, and category.

```
CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    category_id INT,
    FOREIGN KEY (category_id) REFERENCES categories(id)
);
```

- **Inventory Table:** Tracks the quantity of each product in stock and updates inventory levels upon purchases.

```
CREATE TABLE inventory (
    product_id INT PRIMARY KEY,
    quantity INT NOT NULL,
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

1.2 Handling Product Variations and Categories

To handle product variations (e.g., size, color) and categories, a flexible schema can be adopted:

- **Product Variants:** Variants table linked to the main product table via foreign key, storing attributes such as size and color.

```
CREATE TABLE variants (
    id INT PRIMARY KEY AUTO_INCREMENT,
    product_id INT NOT NULL,
    size VARCHAR(50),
    color VARCHAR(50),
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

- **Categories**: Categories table with a hierarchical structure, allowing products to be classified into multiple categories.

```
CREATE TABLE categories (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    parent_category_id INT,
    FOREIGN KEY (parent_category_id) REFERENCES categories(id)
);
```

2. User Authentication and Authorization

Secure user authentication and authorization mechanisms are essential for protecting user accounts and data:

- **Authentication**: Users can sign up, log in, and reset passwords securely using techniques like bcrypt for password hashing.

```
// Sample code for password hashing using bcrypt
const bcrypt = require('bcrypt');

const hashedPassword = await bcrypt.hash(password, 10);
```

- **Authorization**: Role-based access control (RBAC) restricts access to certain features based on user roles (e.g., customer, admin).

```
// Sample code for role-based access control middleware
const checkRole = (role) => {
  return (req, res, next) => {
    if (req.user && req.user.role === role) {
      next();
    } else {
      res.status(403).send('Unauthorized');
    }
  };
};
```

3. Shopping Cart and Checkout

3.1 Shopping Cart System

Designing a shopping cart system involves managing the addition/removal of items and updating quantities:

- **Session-Based Carts:** Each user's cart is stored in their session, allowing them to add/remove items and update quantities before checkout.

```
// Sample code for managing session-based shopping cart
app.post('/cart/add', (req, res) => {
  const { productId, quantity } = req.body;
  // Add product to user's session cart
});

app.post('/cart/remove', (req, res) => {
  const { productId } = req.body;
  // Remove product from user's session cart
});

app.post('/cart/update', (req, res) => {
  const { productId, quantity } = req.body;
  // Update product quantity in user's session cart
});
```

3.2 Checkout Process

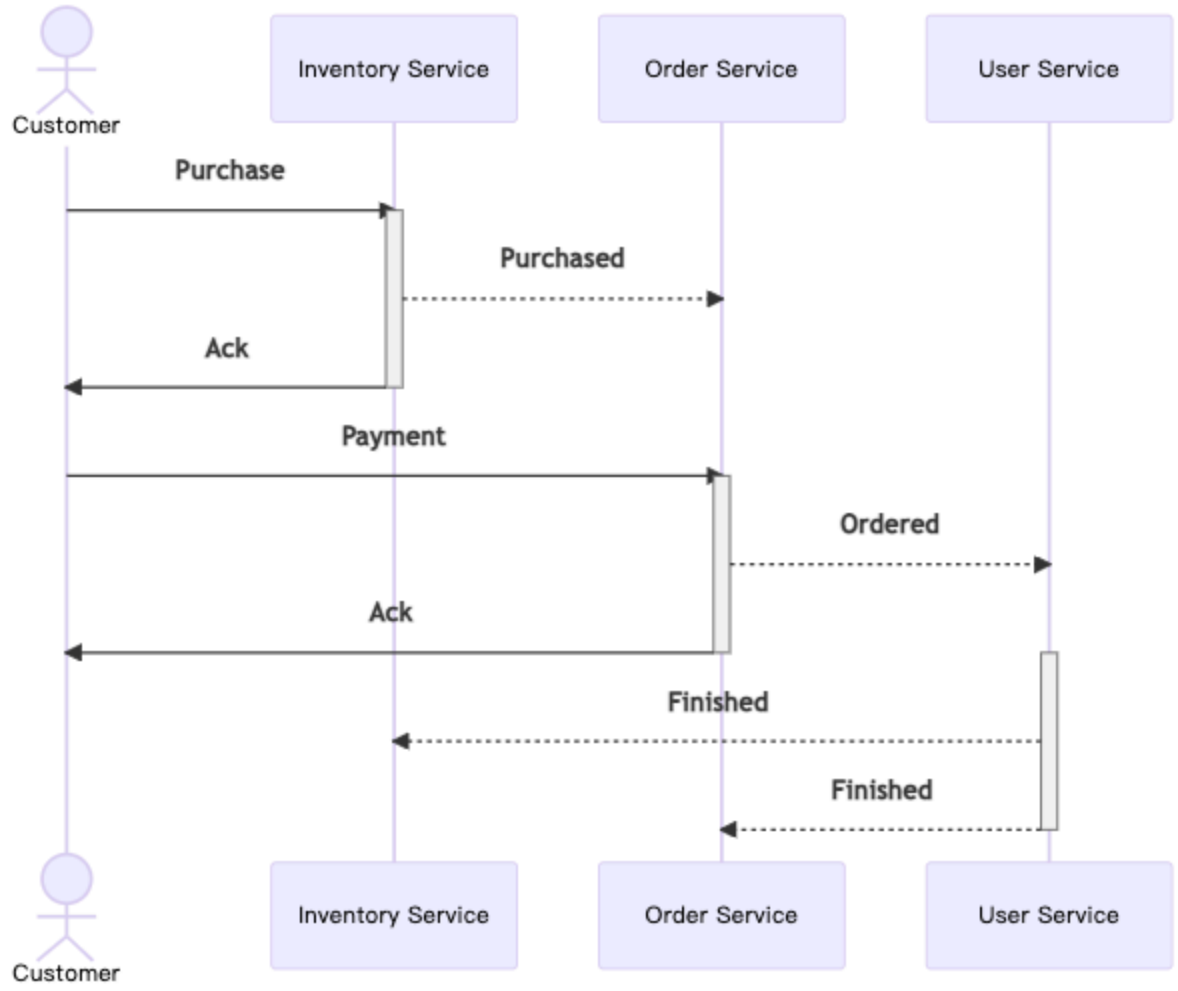
The checkout process involves several steps, including selecting shipping/payment methods and order confirmation:

- **Payment Gateways:** Integration with payment gateways like PayPal or Stripe for secure payment processing.

```
// Sample code for processing payments using Stripe
const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY);

app.post('/checkout', async (req, res) => {
  const { amount, currency, paymentMethodId } = req.body;

  const paymentIntent = await stripe.paymentIntents.create({
    amount,
    currency,
    payment_method: paymentMethodId,
    confirm: true,
  });

  // Process payment and place order
});
```

- **Order Confirmation:** Upon successful payment, generate an order confirmation with a unique order ID and email notification to the customer.

```
// Sample code for sending order confirmation email
const sendOrderConfirmationEmail = (email, orderId) => {
  // Send email to customer with order confirmation details
};
```



4. Order Processing and Fulfillment

4.1 Order Workflow and Fulfillment

The system's workflow for processing orders and coordinating order fulfillment can be designed as follows:

- **Order Status:** Orders progress through various statuses (e.g., pending, processing, shipped, delivered), with updates sent to customers via email.

```
// Sample code for updating order status
const updateOrderStatus = (orderId, newStatus) => {
  // Update order status in the database
};
```

- **Inventory Management:** Upon order placement, decrement product quantities in the inventory to reflect the purchased items.

```
// Sample code for updating inventory levels after order placement
const updateInventory = (productId, quantity) => {
  // Decrement product quantity in the inventory
};
```

4.2 Returns and Refunds

Handling returns and refunds requires clear policies and efficient processes:

- **Return Authorization:** Customers initiate return requests, which are reviewed and approved by customer service representatives.

```
// Sample code for processing return requests
app.post('/returns', (req, res) => {
  const { orderId, reason } = req.body;
  // Process return request and update order status
});
```

- **Refund Processing:** Upon approval, refunds are processed securely through the payment gateway, with funds returned to the customer's original payment method.

```
// Sample code for processing refunds using Stripe
const refundPayment = (paymentIntentId) => {
  // Refund payment using paymentIntentId
};
```

5. Search, Analytics, and Recommendations

5.1 Search Functionality

Implementing a search functionality enables users to find products efficiently:

- **Full-Text Search:** Utilize full-text search capabilities of the database for fast and accurate product searches.

- **Filters and Sorting**: Allow users to filter and sort search results by attributes such as price, category, and relevance.

5.2 Product Recommendations

Providing product recommendations based on user behavior enhances the shopping experience:

- **Collaborative Filtering:** Analyze user interactions (e.g., purchases, views) to generate personalized product recommendations.
- **Content-Based Filtering:** Recommend products based on similarities with previously viewed or purchased items.

6. Scalability and High Availability

6.1 Handling Concurrent Users and Transactions

Horizontal Scaling

Horizontal scaling involves adding more servers or instances to the application to distribute the incoming workload across multiple machines. This approach allows the system to handle a larger number of concurrent users and transactions by spreading the load.

- **Benefits:** Horizontal scaling provides elasticity, allowing the system to dynamically adjust its capacity based on demand. It also improves fault tolerance since the failure of one server does not impact the entire system.

Database Sharding

Database sharding is a technique used to partition the database horizontally into smaller, more manageable shards. Each shard contains a subset of the data, distributing the database workload across multiple servers.

- **Benefits**: Database sharding improves scalability by allowing the system to distribute the data processing load across multiple database servers. It also enhances performance by reducing the number of records each database server needs to handle.

6.2 Load Balancing and High Availability

Load Balancers

Load balancers distribute incoming traffic across multiple servers or instances to prevent overload on any single server and improve responsiveness. Load balancers can employ

various algorithms to determine how to distribute the traffic, such as round-robin, least connections, or weighted round-robin.

- **Benefits**: Load balancers improve scalability by evenly distributing the workload across multiple servers, preventing any single server from becoming a bottleneck. They also enhance fault tolerance by automatically rerouting traffic away from failed or overloaded servers.

Redundancy and Failover

Redundancy involves maintaining duplicate systems or components to provide backup in case of failure. Failover mechanisms automatically redirect traffic to redundant systems or backup servers in the event of a failure, ensuring continuous operation and minimizing downtime.

- **Benefits**: Redundancy and failover mechanisms improve high availability by ensuring that there are backup systems or servers available to take over in case of failure. This reduces the risk of service disruptions and improves reliability for users.

7. Security Measures

7.1 Protecting User Data During Transactions

SSL Encryption

SSL (Secure Sockets Layer) encryption ensures that data transmitted between the client (e.g., web browser) and the server is encrypted, preventing unauthorized interception or tampering.

- **Encryption**: SSL/TLS protocols encrypt the data, making it unreadable to anyone without the proper decryption key.
- **Data Integrity:** SSL ensures the integrity of the data by detecting any unauthorized modifications during transmission.
- **Authentication:** SSL certificates verify the identity of the server, ensuring that the client is communicating with the intended recipient.

PCI Compliance

Adhering to the Payment Card Industry Data Security Standard (PCI DSS) requirements ensures that sensitive payment information, such as credit card details, is handled securely.

- **Data Protection:** PCI DSS mandates measures to protect cardholder data, including encryption, access controls, and secure transmission protocols.

- **Regular Audits:** Compliance with PCI DSS requires regular audits and assessments to validate adherence to security standards and identify areas for improvement.
- **Risk Management:** PCI DSS compliance helps mitigate the risk of data breaches and financial fraud, protecting both merchants and consumers.

7.2 Guarding Against E-commerce Security Threats

DDoS Protection

Distributed Denial of Service (DDoS) attacks aim to disrupt services by overwhelming them with a flood of traffic. Implementing DDoS protection measures helps defend against these attacks and ensure service availability.

- **Traffic Filtering:** DDoS mitigation services filter incoming traffic to identify and block malicious requests, preventing them from reaching the target server.
- **Scalability:** DDoS protection services are scalable and capable of handling large volumes of traffic, allowing the system to remain responsive even during attacks.
- **Anomaly Detection:** Advanced DDoS protection solutions employ anomaly detection algorithms to identify and mitigate abnormal traffic patterns indicative of DDoS attacks.

Regular Security Audits

Regular security audits and vulnerability assessments help identify and address potential security vulnerabilities before they can be exploited by attackers.

- **Penetration Testing:** Penetration tests simulate real-world attacks to identify vulnerabilities in the system, including software flaws, misconfigurations, and access control issues.
- **Code Reviews**: Thorough code reviews help identify security vulnerabilities in the application code, such as injection flaws, cross-site scripting (XSS), and insecure direct object references.
- **Patch Management:** Regularly applying security patches and updates to software and systems helps remediate known vulnerabilities and reduce the attack surface.

8. Customer Reviews and Ratings

8.1 Review and Rating System

Enable customers to leave reviews and ratings for products:

- **Review Submission:** Customers can submit reviews with ratings and optional comments for products they have purchased.
- **Moderation**: Implement moderation mechanisms to filter out inappropriate content and ensure authenticity of reviews.

8.2 Preventing Fake Reviews and Managing Ratings

To prevent fake reviews and maintain trustworthiness:

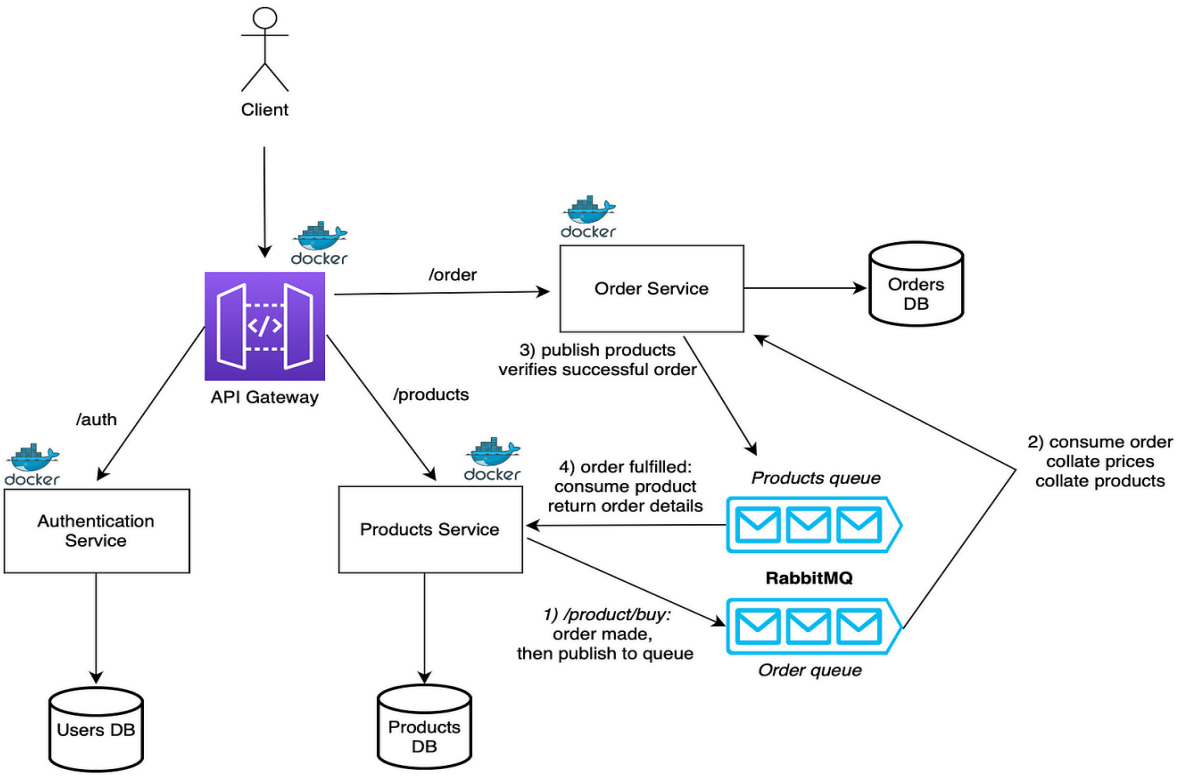
- **Verification:** Implement measures to verify the authenticity of user accounts and reviews, such as email verification and CAPTCHA.
- **Rating Aggregation:** Aggregate ratings from verified purchasers to provide a more accurate representation of product quality.

Additional Features

- **Wishlist**: Allow users to save products for future purchase by adding them to a wishlist.
- **Product Comparison**: Enable users to compare features and prices of multiple products side by side.
- **Order Tracking:** Provide real-time order tracking and status updates to customers via the website or mobile app.

Conclusion

Designing the backend system for an e-commerce platform involves implementing various features and functionalities to provide a seamless shopping experience for users. By focusing on product catalog management, user authentication, shopping cart and checkout processes, order processing and fulfillment, search and recommendation systems, scalability and high availability, security measures, and customer reviews and ratings, the system can deliver a secure, reliable, and user-friendly platform for buying and selling products online.


Client
docker
API Gateway
/order
docker
Order Service
Orders DB
3) publish products verifies successful order
/auth
/products
docker
Authentication Service
docker
Products Service
4) order fulfilled: consume product return order details
Products queue
RabbitMQ
2) consume order collate prices collate products
1) /product/buy: order made, then publish to queue
Order queue
Users DB
Products DB