Experiment 2

Student Name: Gourav Sharma **UID: 23BCS10857**
Branch: CSE **Section/Group: KRG 3-A**
Semester: 6th **Date of Performance:15/01/2026**
Subject Name: System Design **Subject Code: 23CSH-314**

1. **Aim**: To design and implement an E-commerce platform like Amazon/Flipkart that allows users to search products, view product details, add items to cart, checkout & payment, and track orders with proper inventory handling.

2. **Objective**:

- To understand E-commerce system workflow.
- To design functional and non-functional requirements.
- To create system architecture (HLD).
- To design modules/classes (LLD).
- To implement APIs for products, cart, checkout, payment, orders.
- To ensure stock consistency during flash-sale / concurrent orders.

3. **Tools Used:**

- **Python** – Backend logic implementation and URL generation algorithms.
- **Flask** – Lightweight web framework for developing RESTful APIs.
- **Draw.io** – Designing system architecture diagrams (HLD & LLD).

4. **System Requirements:**

A. Functional Requirements

User Module

1. User registration & login
2. Profile management (address, phone, email)

Product Module

3. Search products by title/name/category
4. Filter products (price, rating, brand)
5. View product details (image, description, price, available quantity, reviews)

Cart Module

6. Add item to cart (choose quantity)
7. Update quantity in cart
8. Remove item from cart

Checkout + Payment Module

9. Checkout cart (address selection)
10. Payment via UPI/Card/COD
11. Generate invoice/order confirmation

Order Module

12. Place order
13. Track order status (Placed/Shipped/Delivered/Cancelled)
14. Order history

Inventory Module

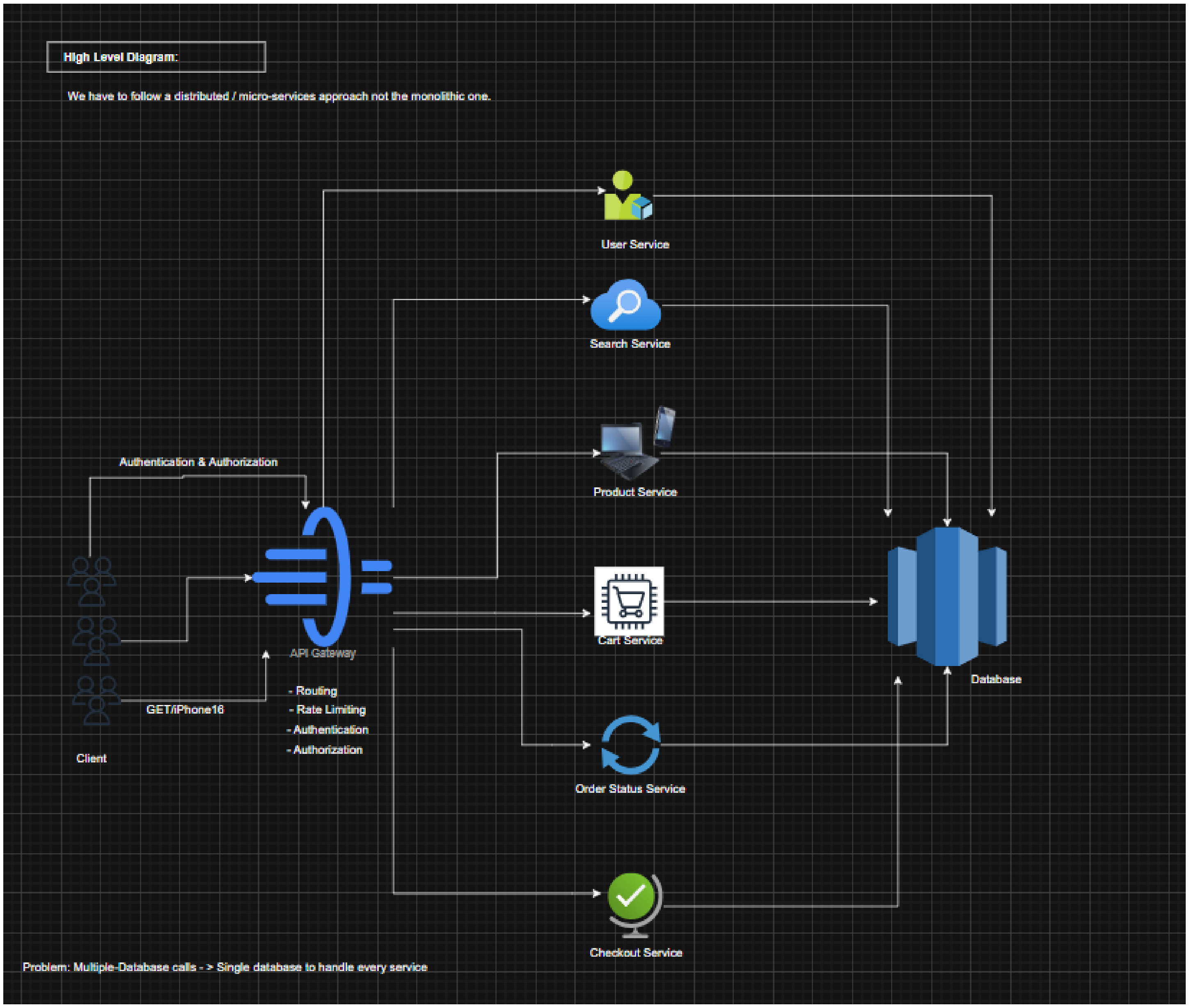
15. Maintain product stock count
16. Handle limited stock + flash-sale race condition

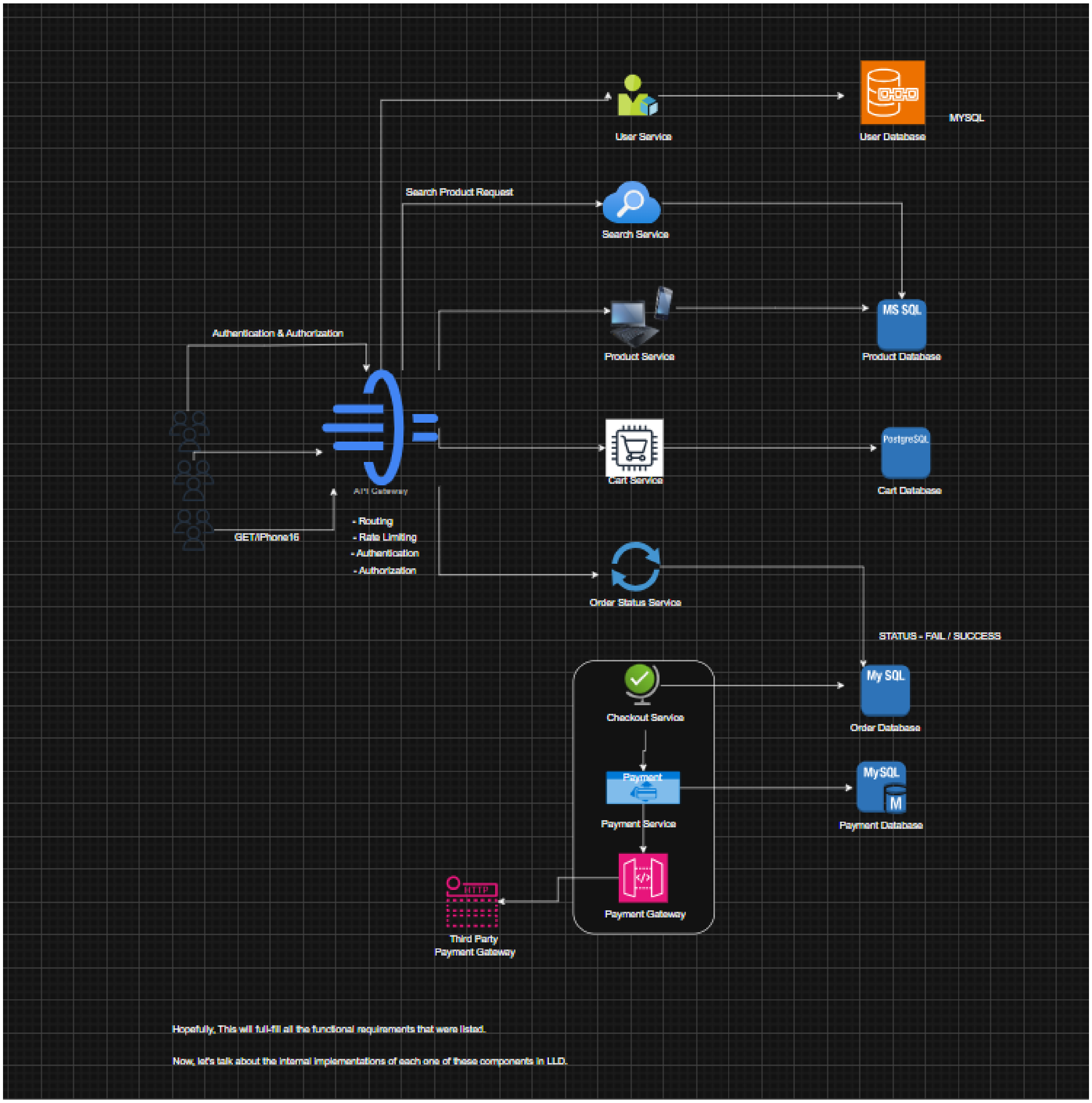
B. Non-Functional Requirements

- Scalability: 100M DAU, 10+ orders/sec
- Availability: 99.9% uptime
- Latency: search & product listing under ~200ms
- Consistency:
- Strong consistency for payment + inventory
- Eventual consistency acceptable for search indexing
- Security: JWT auth, encrypted passwords, HTTPS
- Reliability: rollback on payment failure
- Maintainability: modular services
- Logging & Monitoring: request logs + failure alerts

5. High Level Design (HLD):

The system follows a **Client–Server–Database architecture**:

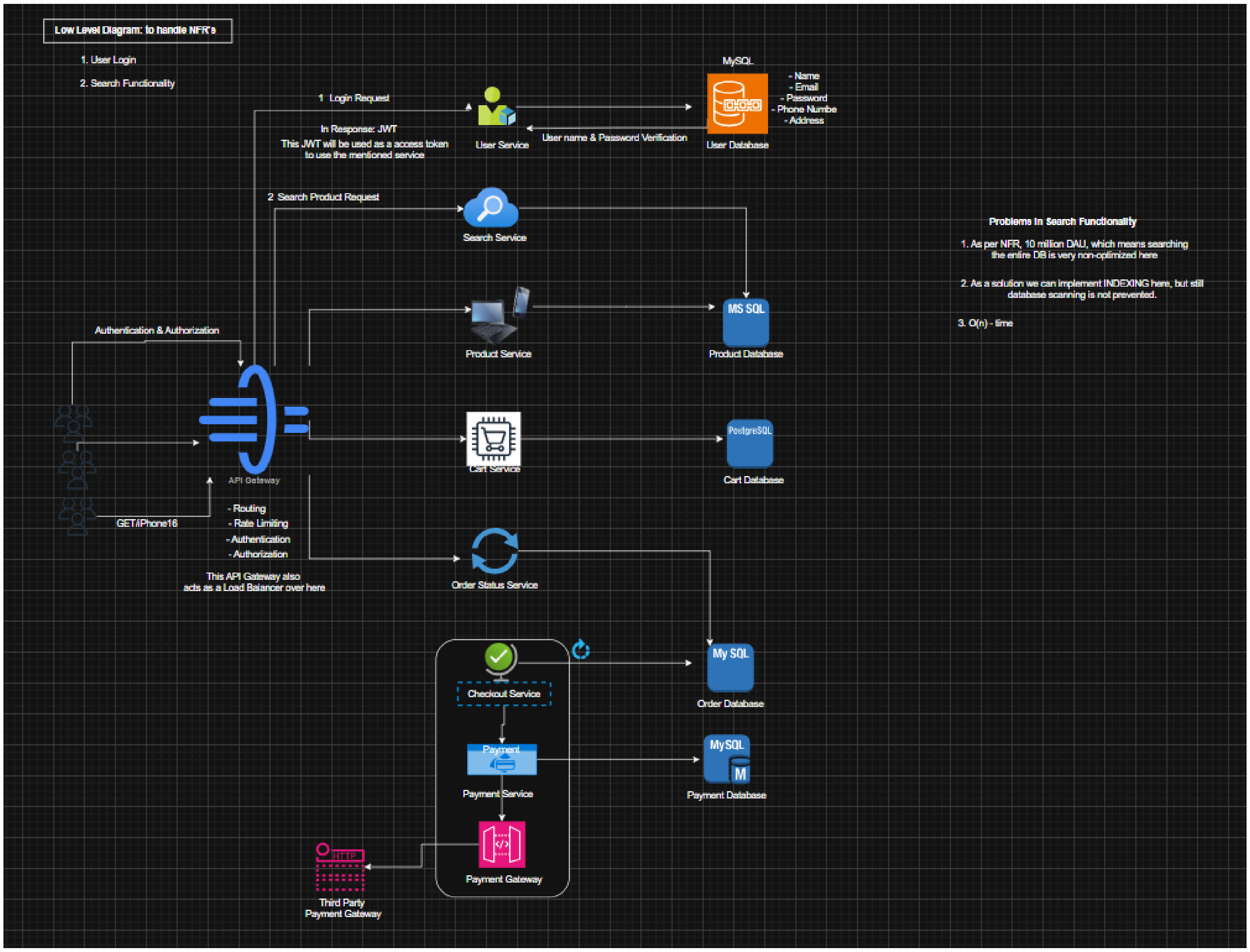


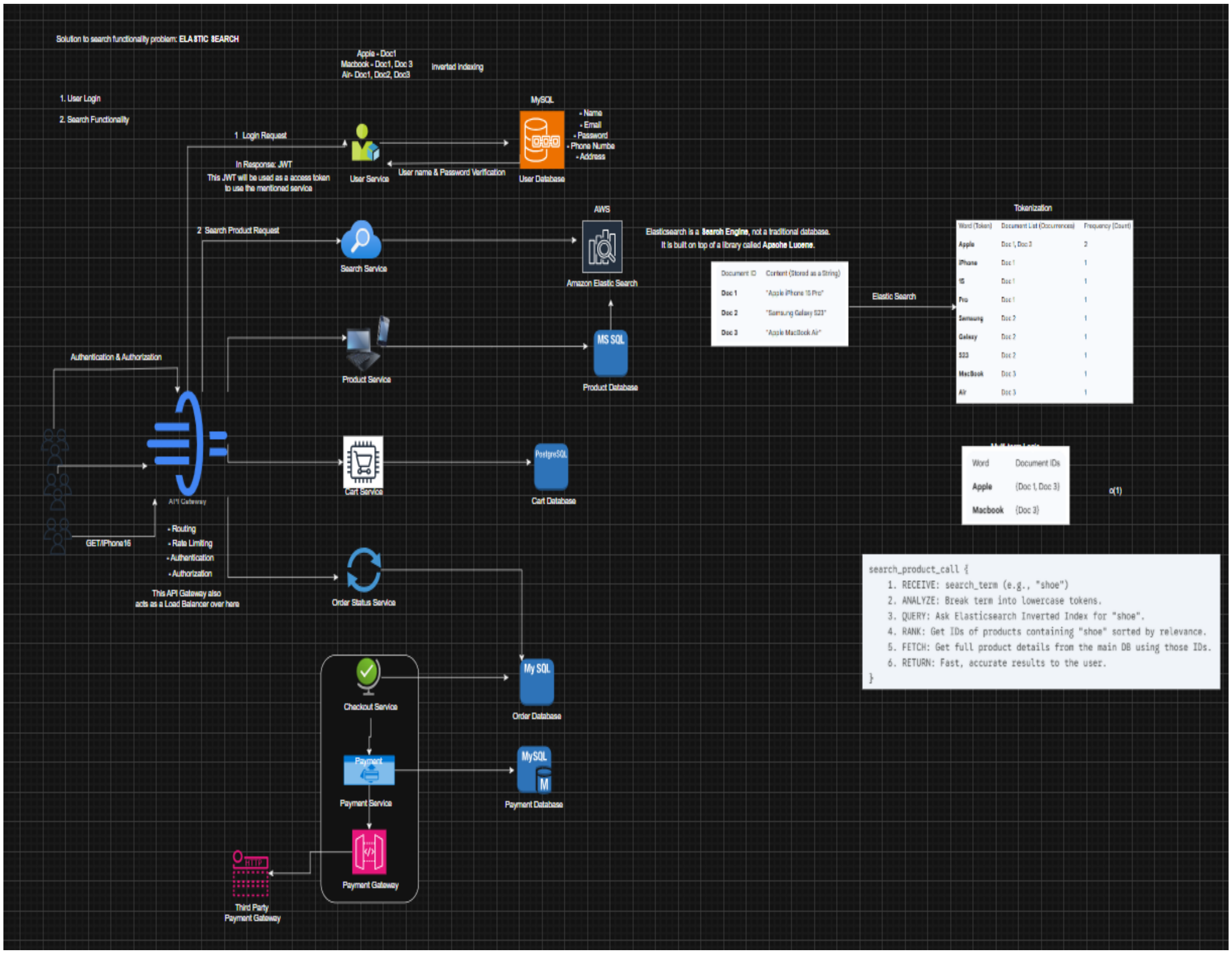


Hopefully, This will full-fill all the functional requirements that were listed.

Now, let's talk about the internal implementations of each one of these components in LLD.

6. Low Level Design (LLD):


Low Level Diagram: to handle NFR's
1. User Login
2. Search Functionality
1 Login Request
In Response: JWT
This JWT will be used as a access token to use the mentioned service
User Service
User name & Password Verification
MySQL
User Database
- Name
- Email
- Password
- Phone Numbe
- Address
2 Search Product Request
Search Service
Problems in Search Functionality
1. As per NFR, 10 million DAU, which means searching the entire DB is very non-optimized here
2. As a solution we can implement INDEXING here, but still database scanning is not prevented.
3. O(n) - time
Authentication & Authorization
Product Service
MS SQL
Product Database
Cart Service
PostgreSQL
Cart Database
API Gateway
GET/iPhone16
- Routing
- Rate Limiting
- Authentication
- Authorization
This API Gateway also acts as a Load Balancer over here
Order Status Service
My SQL
Order Database
Checkout Service
Payment
Payment Service
MySQL
Payment Database
Payment Gateway
HTTP
Third Party Payment Gateway



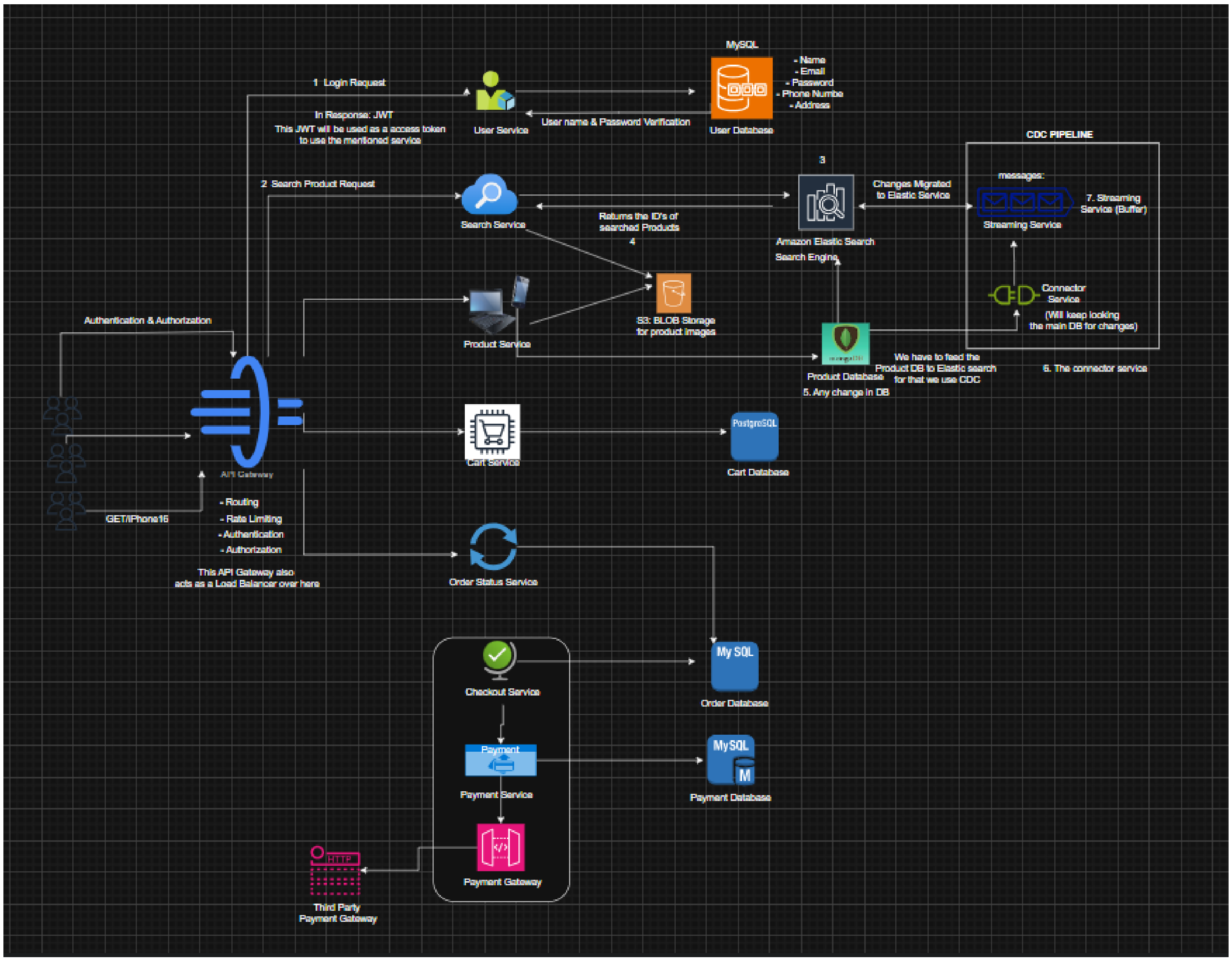
Document ID	Content (Stored as a String)
Doc 1	"Apple iPhone 15 Pro"
Doc 2	"Samsung Galaxy S23"
Doc 3	"Apple MacBook Air"

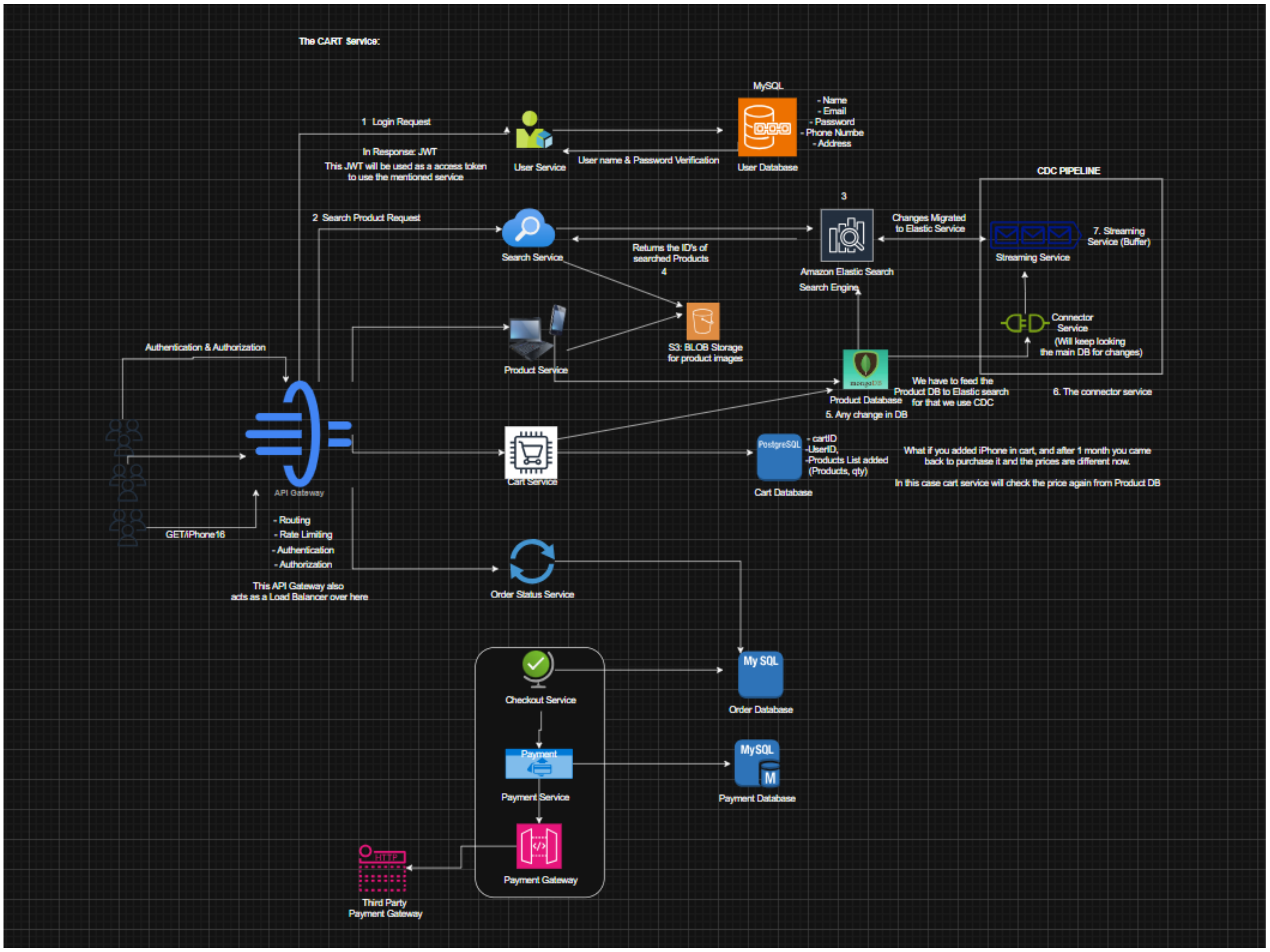
Tokenization

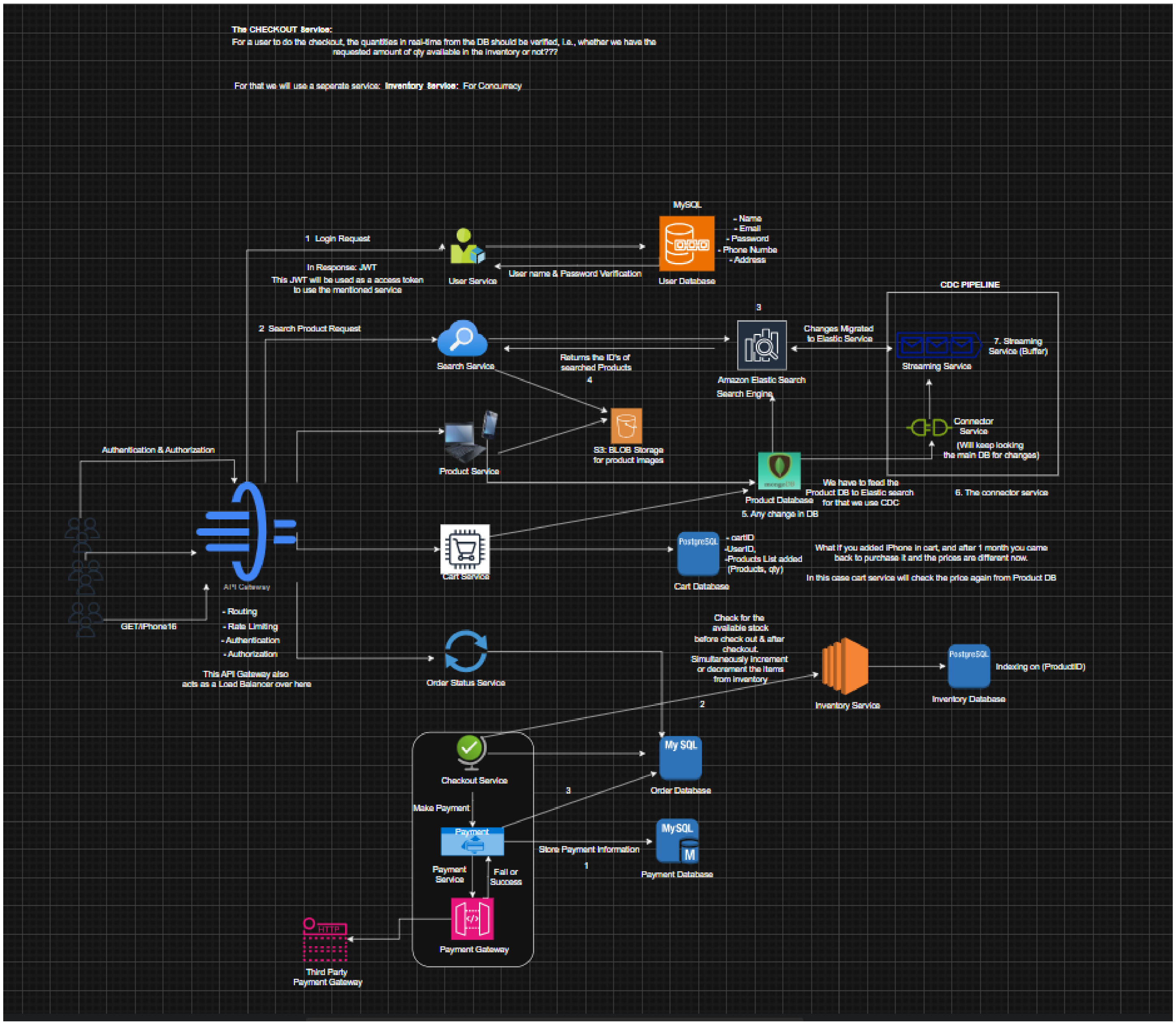
Word (Token)	Document List (Occurrences)	Frequency (Count)
Apple	Doc 1, Doc 3	2
iPhone	Doc 1	1
15	Doc 1	1
Pro	Doc 1	1
Samsung	Doc 2	1
Galaxy	Doc 2	1
S23	Doc 2	1
MacBook	Doc 3	1
Air	Doc 3	1

Word	Document IDs
Apple	{Doc 1, Doc 3}
Macbook	{Doc 3}

```
search_product_call {
    1. RECEIVE: search_term (e.g., "shoe")
    2. ANALYZE: Break term into lowercase tokens.
    3. QUERY: Ask Elasticsearch Inverted Index for "shoe".
    4. RANK: Get IDs of products containing "shoe" sorted by relevance.
    5. FETCH: Get full product details from the main DB using those IDs.
    6. RETURN: Fast, accurate results to the user.
}
```


1 Login Request
In Response: JWT
This JWT will be used as a access token
to use the mentioned service
User Service
MySQL
User name & Password Verification
User Database
- Name
- Email
- Password
- Phone Numbe
- Address
CDC PIPELINE
2 Search Product Request
Search Service
Returns the ID's of
searched Products
4
3
Amazon Elastic Search
Search Engine
Changes Migrated
to Elastic Service
messages:
7. Streaming
Service (Buffer)
Streaming Service
Connector
Service
(Will keep looking
the main DB for changes)
Authentication & Authorization
Product Service
S3: BLOB Storage
for product images
Product Database
5. Any change in DB
We have to feed the
Product DB to Elastic search
for that we use CDC
6. The connector service
API Gateway
Cart Service
PostgreSQL
Cart Database
GET/iPhone16
- Routing
- Rate Limiting
- Authentication
- Authorization
This API Gateway also
acts as a Load Balancer over here
Order Status Service
My SQL
Order Database
Checkout Service
Payment
Payment Service
MySQL
Payment Database
Payment Gateway
HTTP
Third Party
Payment Gateway


The CART Service:
MySQL
- Name
- Email
- Password
- Phone Numbe
- Address
1 Login Request
In Response: JWT
This JWT will be used as a access token
to use the mentioned service
User Service
User name & Password Verification
User Database
CDC PIPELINE
3
2 Search Product Request
Changes Migrated
to Elastic Service
7. Streaming
Service (Buffer)
Search Service
Returns the ID's of
searched Products
4
Amazon Elastic Search
Search Engine
Streaming Service
Connector
Service
(Will keep looking
the main DB for changes)
Authentication & Authorization
Product Service
S3: BLOB Storage
for product images
We have to feed the
Product DB to Elastic search
for that we use CDC
Product Database
6. The connector service
5. Any change in DB
- cartID
-UserID,
-Products List added
(Products, qty)
What if you added iPhone in cart, and after 1 month you came
back to purchase it and the prices are different now.
In this case cart service will check the price again from Product DB
API Gateway
Cart Service
Cart Database
GET/iPhone16
- Routing
- Rate Limiting
- Authentication
- Authorization
This API Gateway also
acts as a Load Balancer over here
Order Status Service
My SQL
Checkout Service
Order Database
Payment
Payment Service
MySQL
Payment Database
Payment Gateway
HTTP
Third Party
Payment Gateway



The CHECKOUT Service:
For a user to do the checkout, the quantities in real-time from the DB should be verified, i.e., whether we have the requested amount of qty available in the inventory or not???

For that we will use a seperate service: **Inventory Service:** For Concurrency

In Response: JWT
This JWT will be used as a access token to use the mentioned service

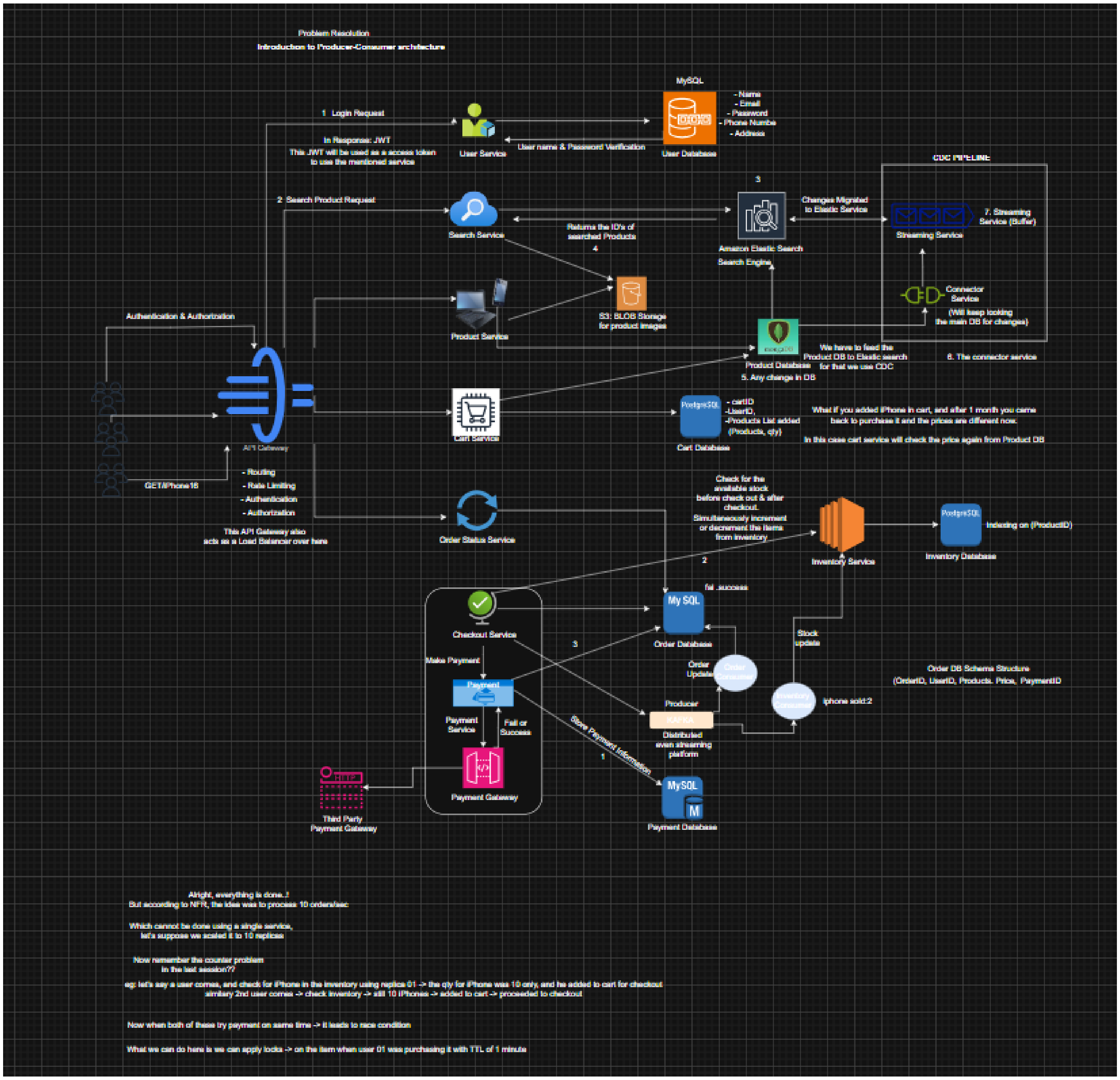
We have to feed the Product DB to Elastic search for that we use CDC

What if you added iPhone in cart, and after 1 month you came back to purchase it and the prices are different now.

In this case cart service will check the price again from Product DB

Check for the available stock before check out & after checkout.
Simultaneously increment or decrement the items from inventory

This API Gateway also acts as a Load Balancer over here



Alright, everything is done..!
But according to NFR, the idea was to process 10 orders/sec

Which cannot be done using a single service,
let's suppose we scaled it to 10 replicas

Now remember the counter problem
in the last session??

eg: let's say a user comes, and check for iPhone in the inventory using replica 01 -> the qty for iPhone was 10 only, and he added to cart for checkout
similary 2nd user comes -> check inventory -> still 10 iPhones -> added to cart -> proceeded to checkout

Now when both of these try payment on same time -> it leads to race condition

What we can do here is we can apply locks -> on the item when user 01 was purchasing it with TTL of 1 minute

7. Scalability Solution

- Use horizontal scaling + auto-scaling to handle high traffic.
- Apply load balancer to distribute user requests across servers.
- Use Redis caching + CDN to reduce database load and speed up responses.

- Implement DB read replicas + sharding to avoid database bottlenecks.
- Use Kafka/RabbitMQ queues for asynchronous processing of heavy tasks.

8. Learning Outcomes (What I Have Learnt)

- Understood the complete E-commerce purchase flow from search to delivery.
- Learned to identify functional and non-functional requirements clearly.
- Designed HLD architecture using services/modules for the system.
- Created LLD entities/tables and relationships for database design.
- Learned scalability + race condition handling for flash-sale inventory.