



## Experiment No: 2

Student Name: Debanuj Kumar De

UID: 23BCS12048

Branch: BE CSE

Section/Group: 23BCS\_KRG-2\_B

Semester: 6<sup>th</sup>

Date of Performance: 14/01/2026

Subject Name: System Design

Subject Code: 23CSH-314

**1- Aim** - Design an Online shopping platform similar to Amazon / Flipkart that will allow users to purchase mobiles, laptops, cameras, clothes etc.

### 2- Requirements: Functional & Non-Functional

#### A- Functional Requirement

- User should be able to search and find the products based on product title or names.
- User should be able to view the details of the product like description, image, available quantity, review.
- User should be able to select the quantity and move the product/item into the cart.
- User should be able to make the payment and should be able to perform the check out.
- User should be able to check the status of the order.
- System should be able to manage purchase of items having limited stocks.

Race condition happens during a flash sale when inventory has limited stock, and the system must handle multiple transactions occurring at the same time.

#### B- Non-Functional Requirement

- The system is designed for 100 million daily active users with around 10 orders handled per second.
- Consistency & Availability: Based on the target scale, both are required, but at different system levels.
  - i. According to functional requirements, users must search products smoothly, so product search needs high availability.
  - ii. Strong consistency is required for critical components such as payment processing, order placement, and inventory management.
- Expected response time is approximately 200 ms.
- Scaling will be done either Horizontally or vertically wherever applicable.

### 3- Core-entities of System

- User/Client
- Products
- Cart
- Orders
- Checkout followed by Payment

### 4- API endpoint creation

#### a) GET API Call: Prod\_Search

Https://Local\_Host/products/search\_item = {Search\_keywords}

HTTP Req

```
{  
    GET: <iPhone 16>
```

```
}
```

HTTP Res

```
{  
    List<ProductID:iPhone>  
}
```

Now, on front-end if multiple data of respective product is coming in that case the FE becomes faulty thus ultimately increasing the Latency.

So we will be using Pagination (1,2,3,....next)

#### **b) GET API Call: View Product Details**

Https://Local\_Host/products/{product\_id}

HTTP Req

```
{  
    GET: <Product_id=17>
```

```
}
```

HTTP Res

```
{  
    Product_id=17,  
    Name: iPhone17,  
    Color: Navy Blue,  
    Price: $1009,  
    Image_URL: URL_image  
}
```

#### **c) POST API Call: Item add in cart**

Https://Local\_Host/cart/add\_products

HTTP Req

```
{  
    Product_id:17,  
    Product_id:16
```

```
}
```

HTTP Req Header

```
{  
    User_id: 04
```

```
}  
HTTP Res  
{  
    Cart_id: 101  
}
```

- d) PUT API Call: To update any order in the cart
- e) DELETE API Call: To remove any item from the cart
- f) **POST API Call: for check out & Payment**  
Https://Local\_Host/checkout -> {post body}

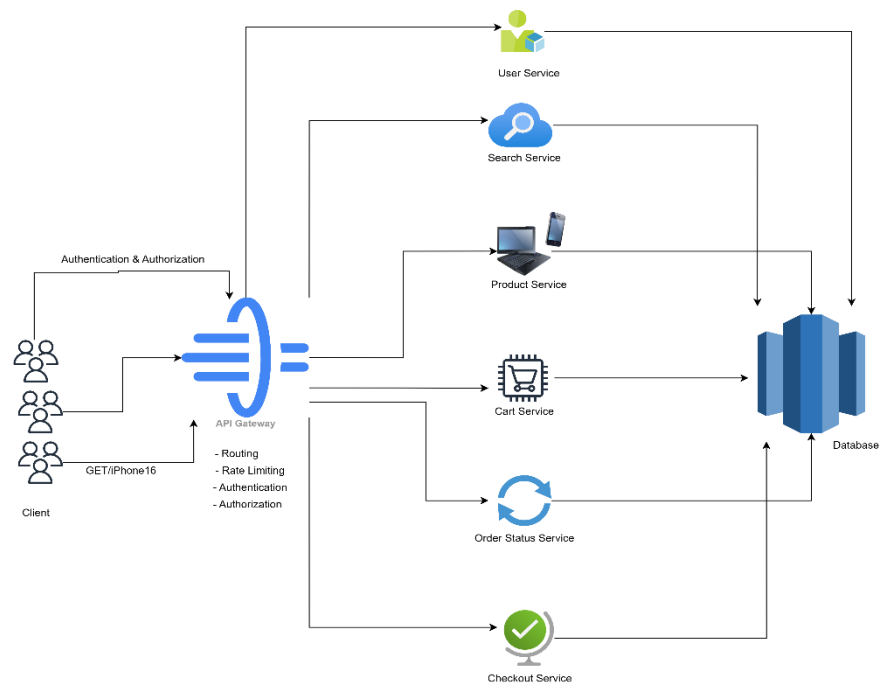
```
HTTP Req  
{  
    All Product Id's,  
    Total Quantity,  
    Total Price  
}  
HTTP Res  
{  
    Order_id  
}
```

```
Https://Local_Host/payment -> {post body}  
HTTP Req  
{  
    Order_id,  
    Payment Type,  
    Payment_Mode  
}  
HTTP Res  
{  
    Confirmation_Status: Succes / Fail  
}
```

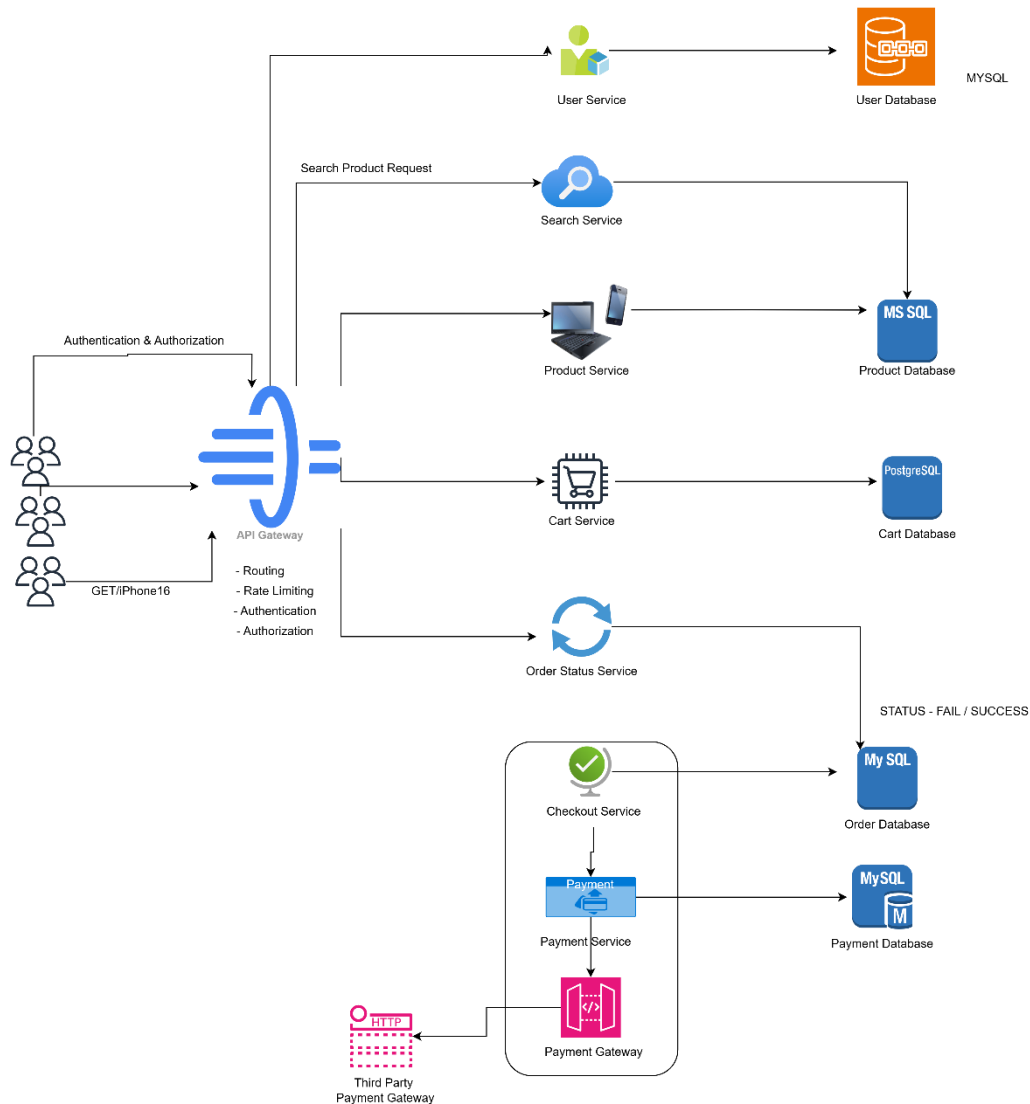
- g) **GET API Call: Order Status**  
Https://Local\_Host/orde\_status = {order\_id}

## 5- High-Level Design

Now According to the functional requirement of the system, we can identify that : We have to follow a distributed / micro-services approach not the monolithic one.



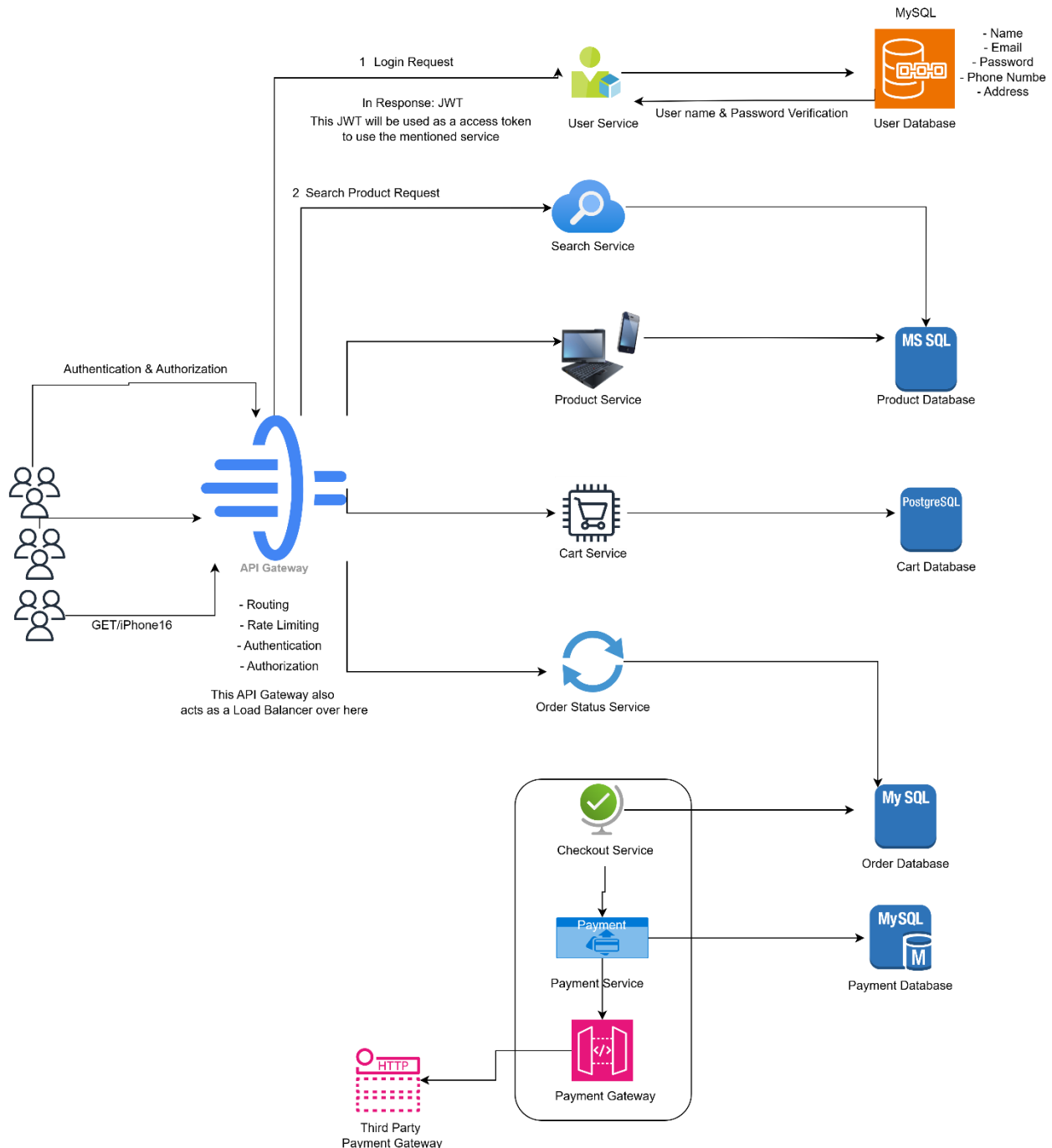
Drawbacks: Multiple-Database calls are being done and single database is being used to handle every service which increases the latency.



This will fulfill all the functional requirements that were listed. Now, we will see the internal implementations of each one of these components in LLD.

## 6- Low- Level Design

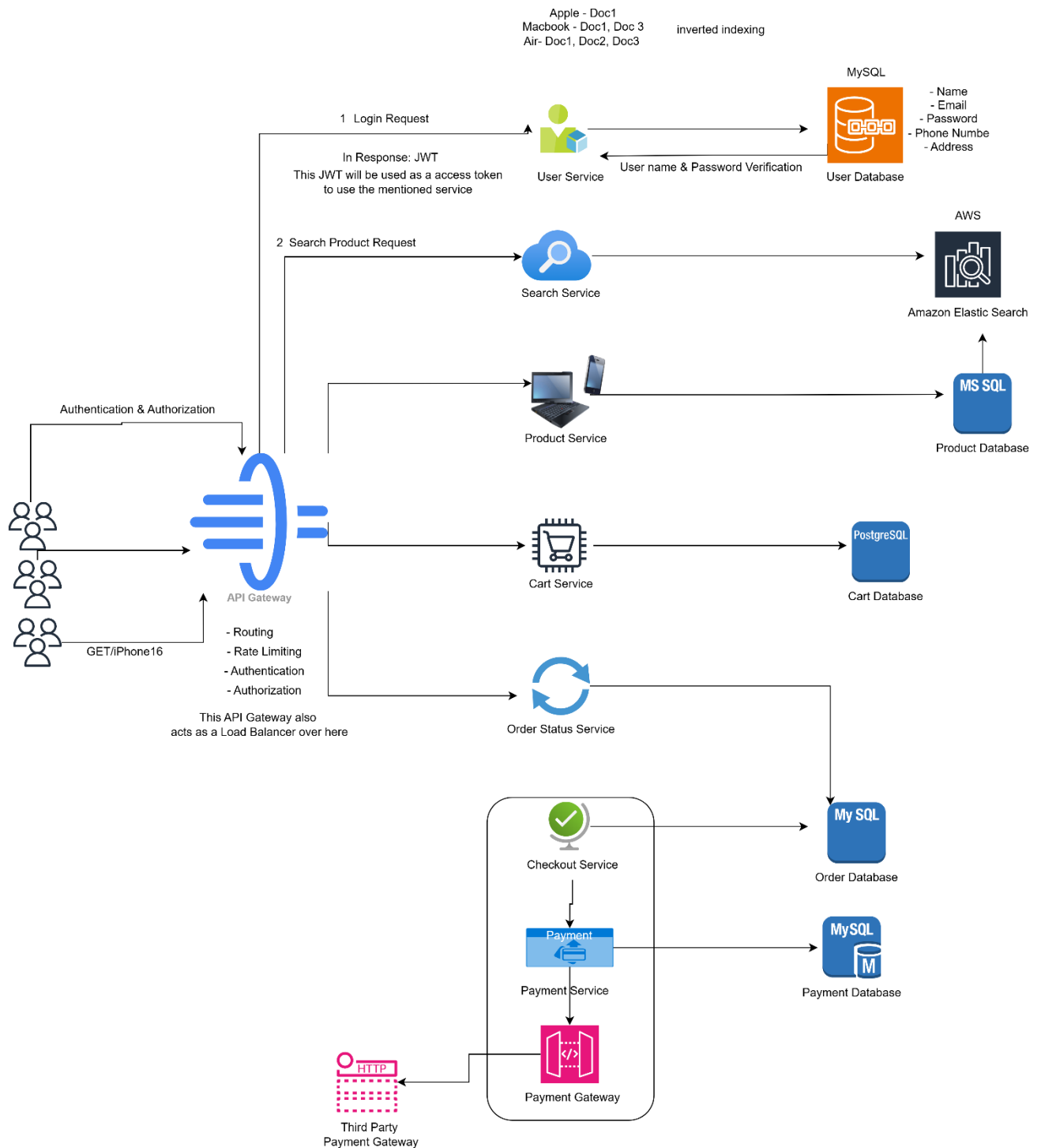
### 1- User Login and Search Functionality



#### Drawbacks-

- As per NFR, 10 million DAU, which means searching the entire DB is very non-optimized here
- As a solution we can implement INDEXING here, but still database scanning is not prevented.
- $O(n)$  – time

## Solution to search functionality problem: **ELASTIC SEARCH.**



Elasticsearch is a **Search Engine**, not a traditional database.  
It is built on top of a library called **Apache Lucene**.

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

Elastic Search

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

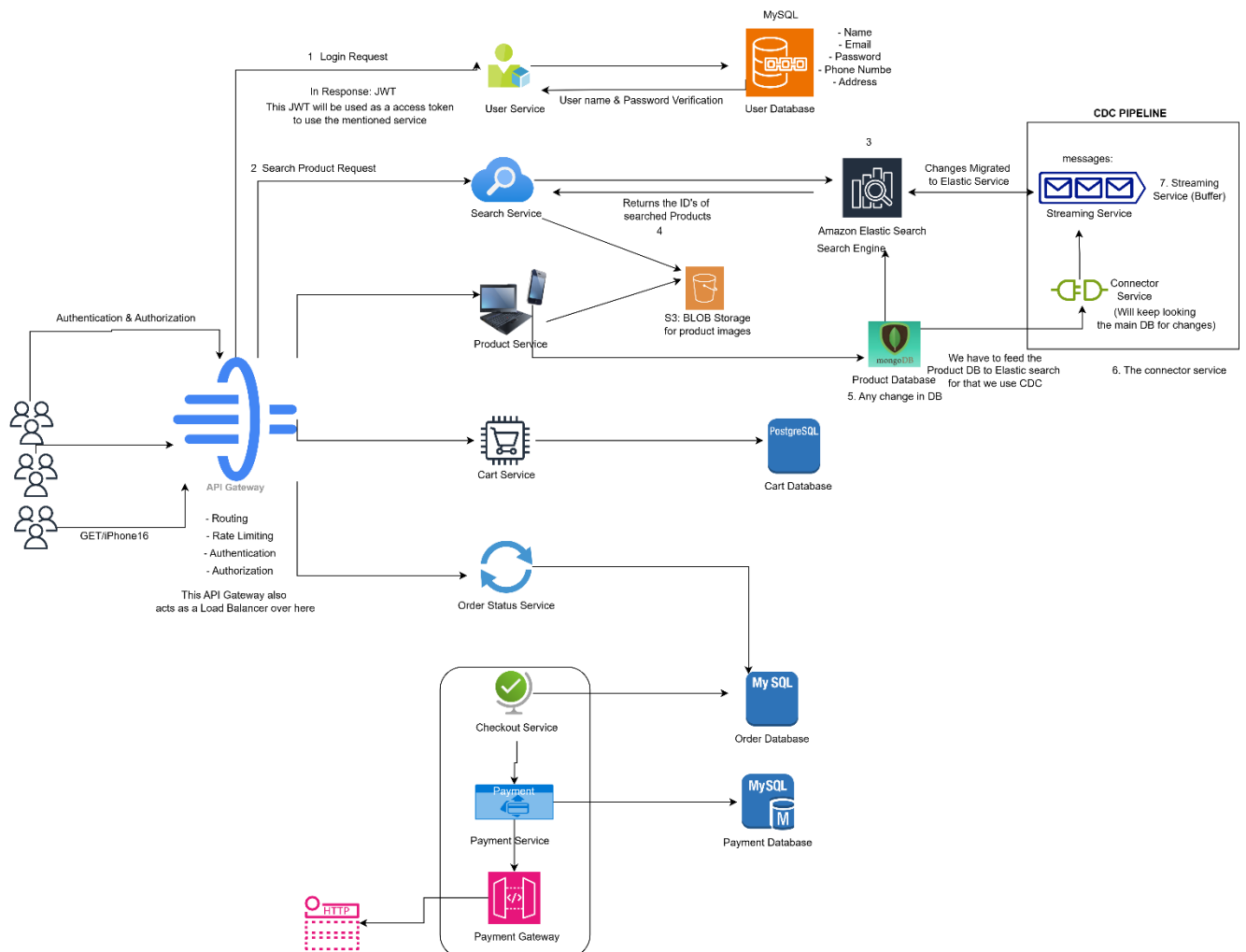
Multi-term Logic

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

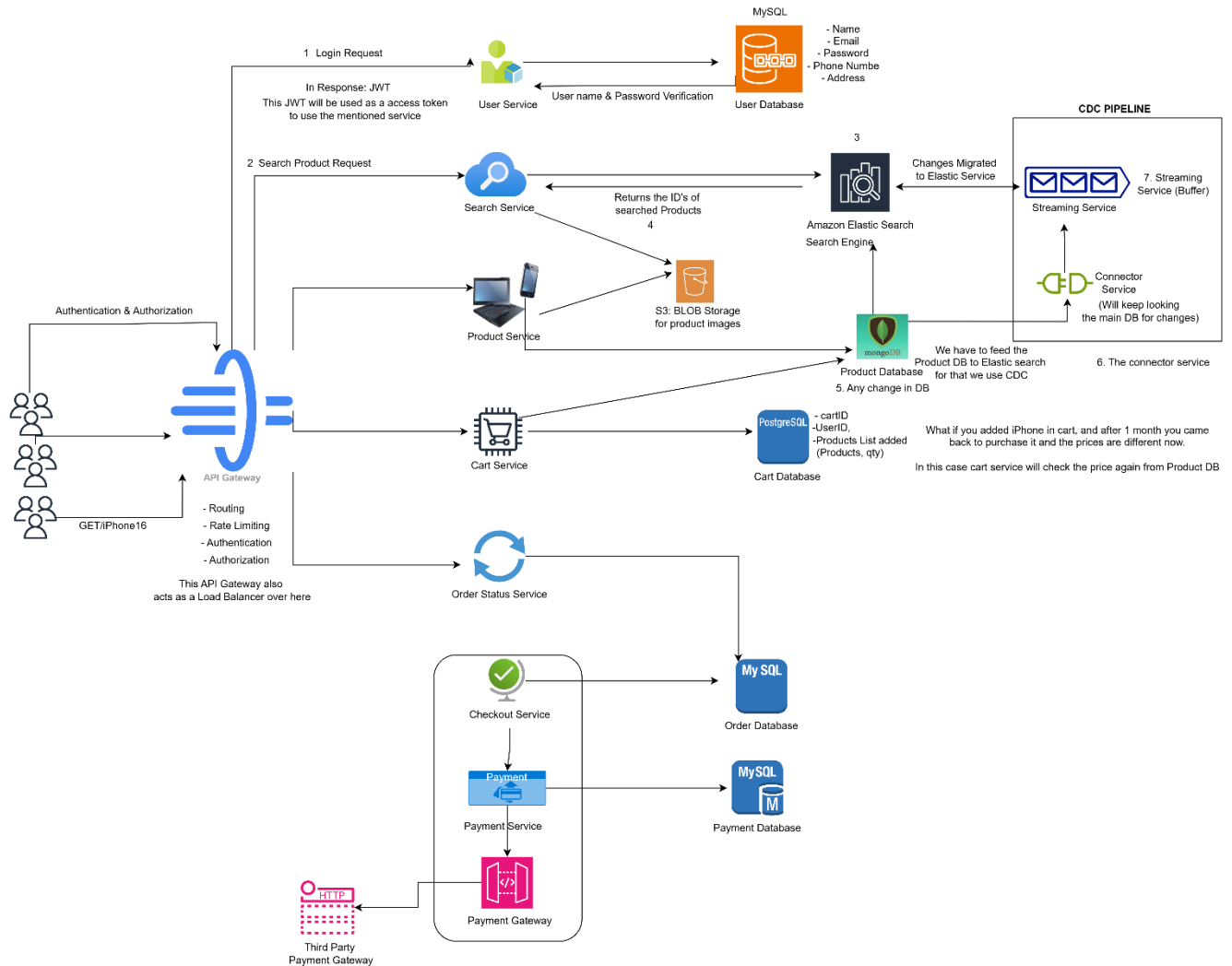
$O(1)$

```
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.
}
```

We will use CDC (Change Data Capture) to send data from original database to ES in real time.



## 2- Cart Service



## 3- Checkout Service

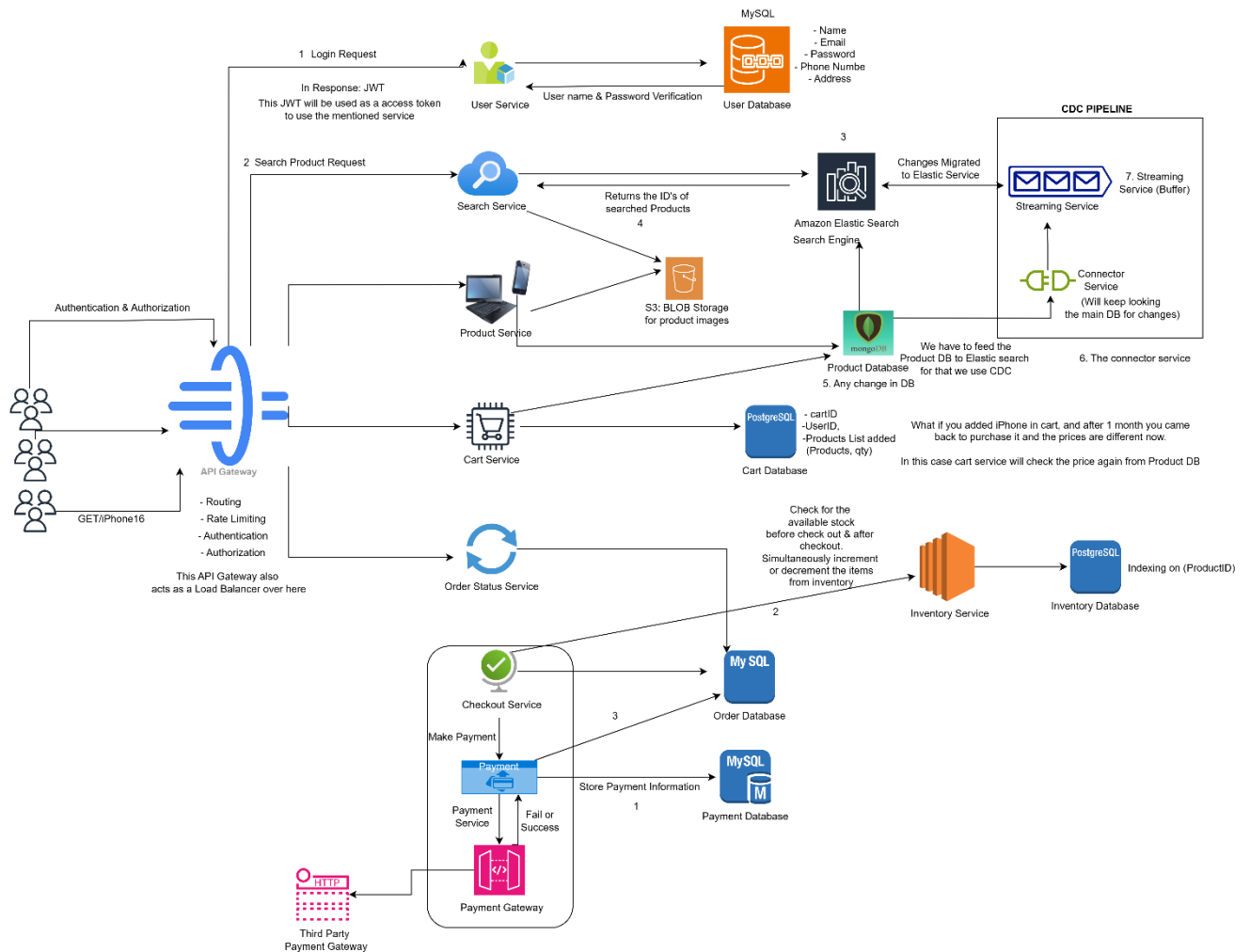
For a user to do the checkout, the quantities in real-time from the DB should be verified, i.e., whether we have the requested amount of qty available in the inventory or not?

For that we will use a separate service: Inventory Service: For Concurrency

### DRAWBACKS –

- 3 API calls are there to perform a single transaction
- Now For this single transaction, it can happen that payment done successfully, inventory was not updated.  
This is a very critical issue w.r.t consistency
- Rate of failure is very high over here





Solutions for above drawbacks is to introduce **Producer-Consumer architecture** using KAFKA.

