Experiment 02:

Design a E-commerce platform similar to Amazon or Flipkart

Design Brief: We have to design an Online shopping platform similar to Amazon / Flipkart that will allow users to purchase mobiles, laptops, cameras, clothes etc.

Important Note: Collection of functional & Non-functional requirements is totally the resposibility of the candidate appearing in the interview

KAFKA, ELASTIC SEARCH, CDC PIPELINE

Functional Requirements:

1.User should be able to search and find the products based on product title or names.

2. User should be able to view the details of the product llike description, image, available quantity, review

3. User should be able to select the quantity and move the product/item into the cart.

4. User should be able to make the payment and should be able to perform the check out.

5. User shoulde be able to check the status of the order.

6. System should be able to manage purchase of items having limited stocks.

Race Condition: If there is flash sale, and in our inventory only limited items are available -> then how concurrent transactions will be processed.

Non-Functional Requirements:

1. Target Scale: 100Million DAU with 10 orders processed per second.

2. Consistency & Availability: Here for this system we need both as per the Target Scale.
Now we should specifiy that, which part of our system needs what?

Availability: As per the FR, we know that client should be able to search for the products efficiently means we need:
High Availability(on Products_search)

Consistency: The system should be highly consistence for our two important modules:

1. Payement and the Order Placing.

2. Inventory Management

3. Latency: Required: ~200 ms

4. Scaling: Horizontal / Vertical

Core-Entites of the System:

1. User / Client

2. Products

3. Cart

4. Orders

5.Checkout followed by Payment

API Designing:

1. 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 -> ultimately increasing the LATENCY.

For that: we can use Pagination 1, 2 ,3 ,4 ,----- SO ON

2. GET API Call: View Product Details

Https://Local_Host/products/{product_id}

```
HTTP Req
    {
        GET: <Product_id = 17>
    }

HTTP Res
    {
        Product_id:17,
        Name: iPhone 17,
        Color: Navy Blue,
        Price: $1099,
        Image Thumbnail: URL_Image
    }
```

3. 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
    }
```

4. PUT API Call: To update any order in the cart

Let's Suppose you want to add one more product into the cart.

5. DELETE API Call: To remove any item from the cart

Let's Suppose you want to delete one more product into the cart.

6. POST API Call: for check out & Payement

Https://Local_Host/checkout -> {post body}

```
HTTP REQ
    {
        All products ID's,
        Total Quantity,
        Total Price
    }
HTTP RES
    {
        Order_ID
    }
```

Https://Local_Host/payment -> {post body}

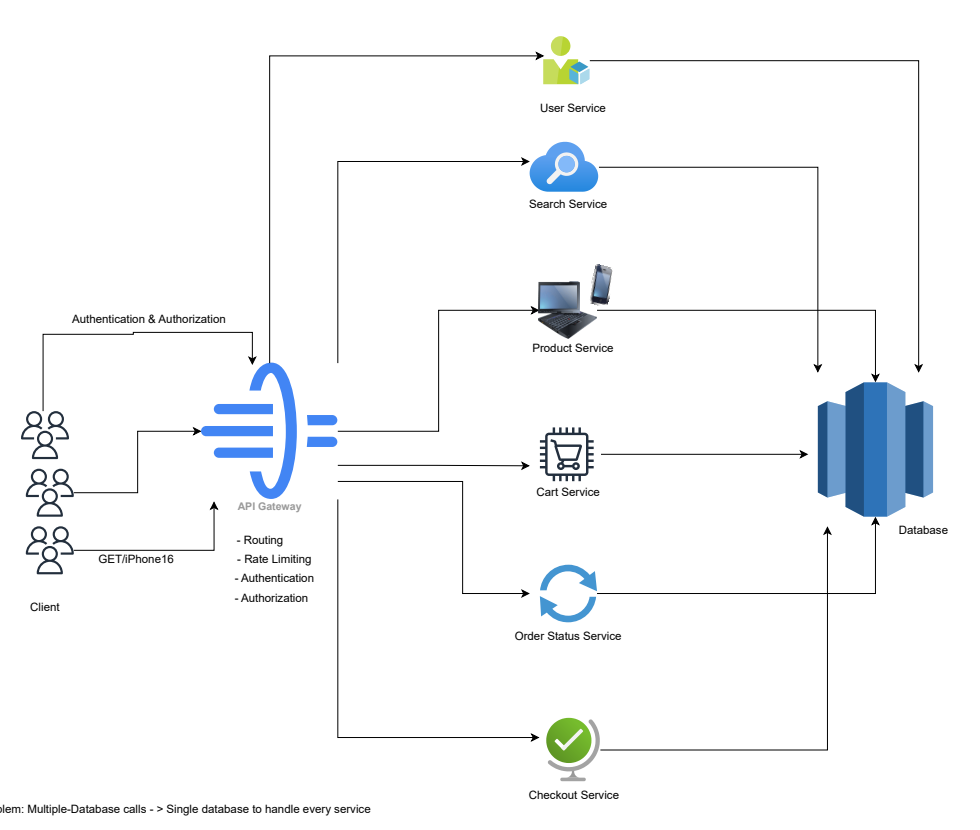
```
HTTP REQ
    {
        Order ID,
        Payment Type,
        Payment Mode
    }

HTTP RES
    {
        Confirmation_Status: Succes / Fail
    }
```

7. GET API Call: Order Status

Https://Local_Host/orde_status = {order_id}

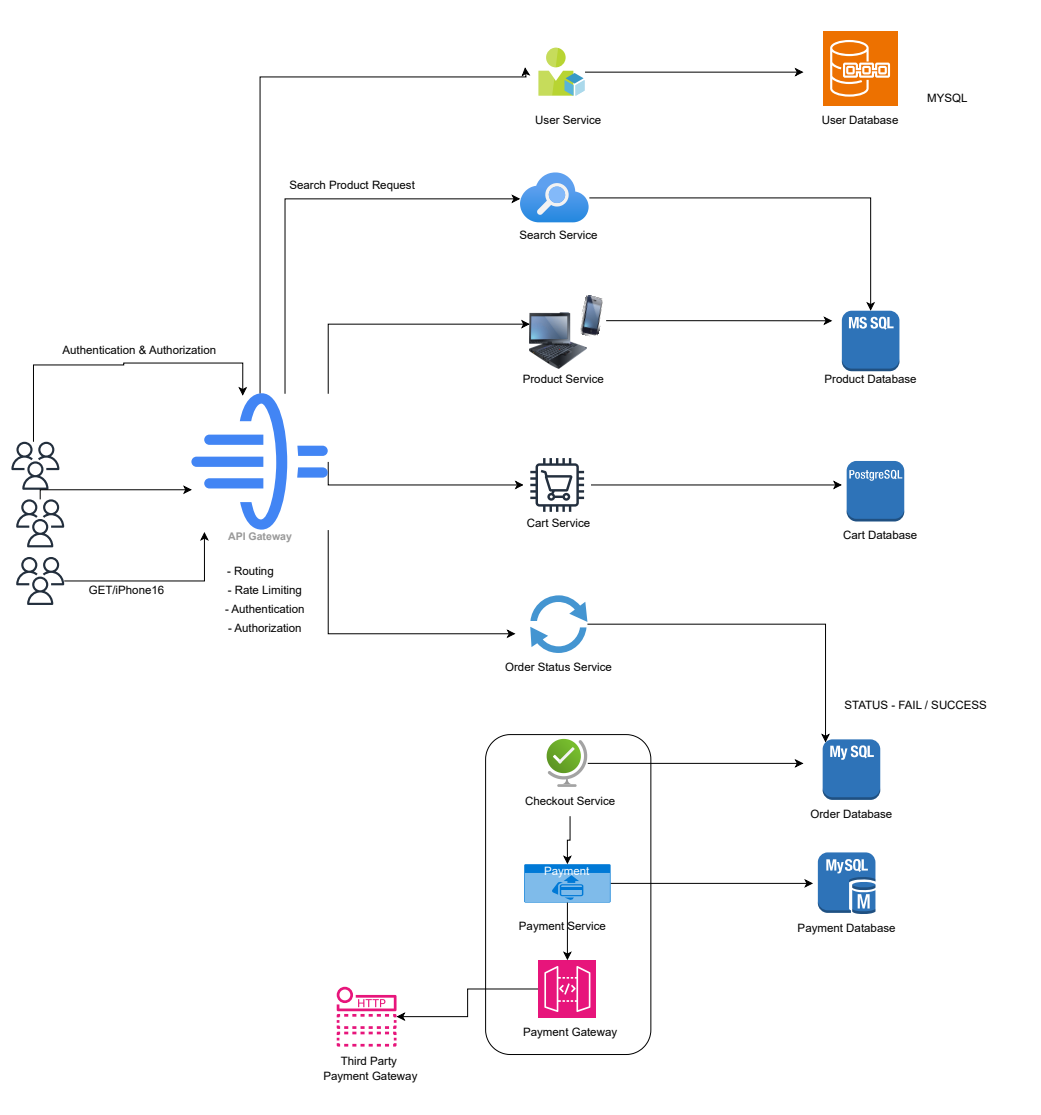
High Level Diagram:

We have to follow a distributed / micro-services approach not the monolithic one.



Problem: Multiple-Database calls - > Single database to handle every service

---



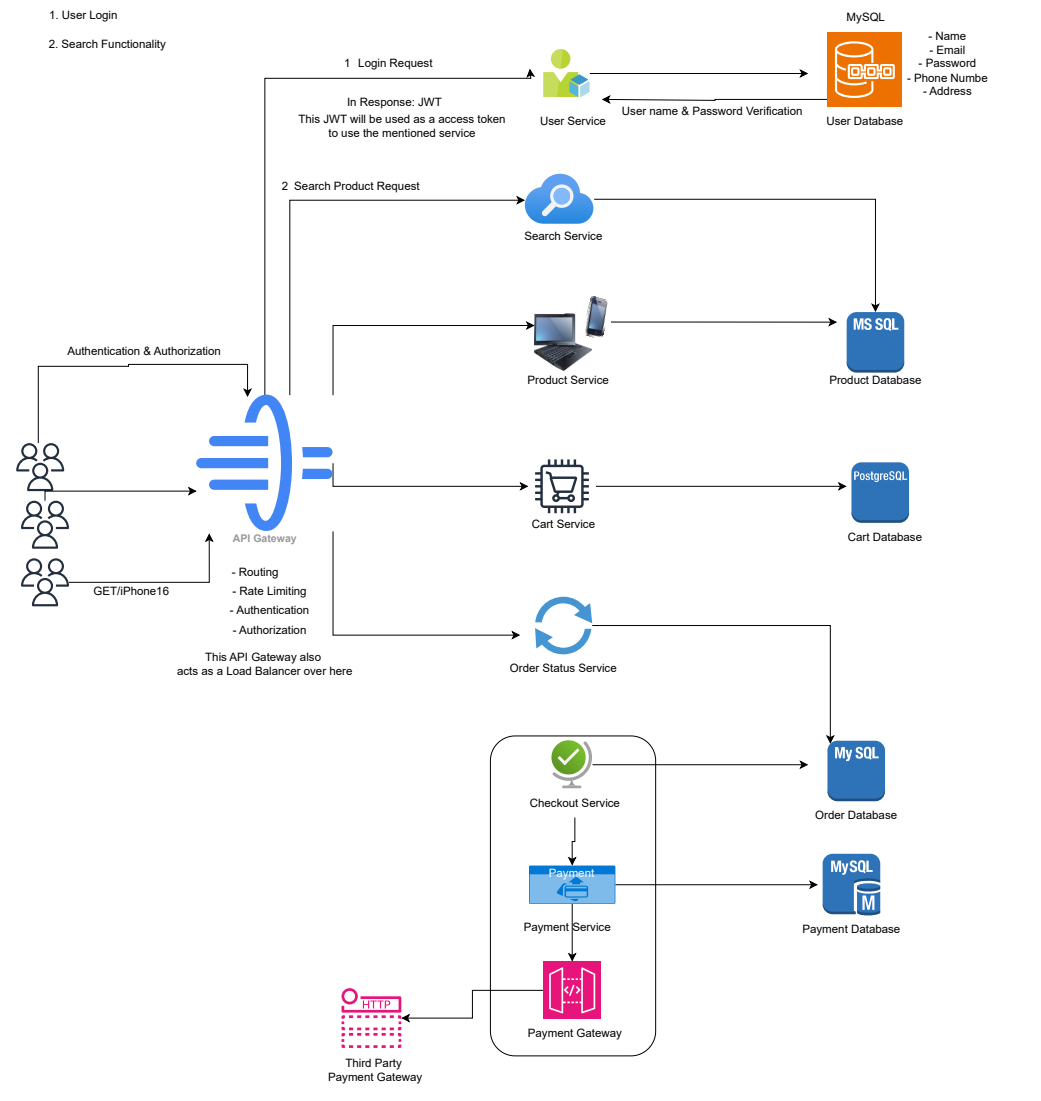
Hopefully, This will full-fill all the functional requirements that were listed.

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

Low Level Diagram: to handle NFR's

1. User Login

2. Search Functionality



Problems in Search Functionality

1. As per NFR, 10 million DAU, which means searching the entire DB is very non-optimized here

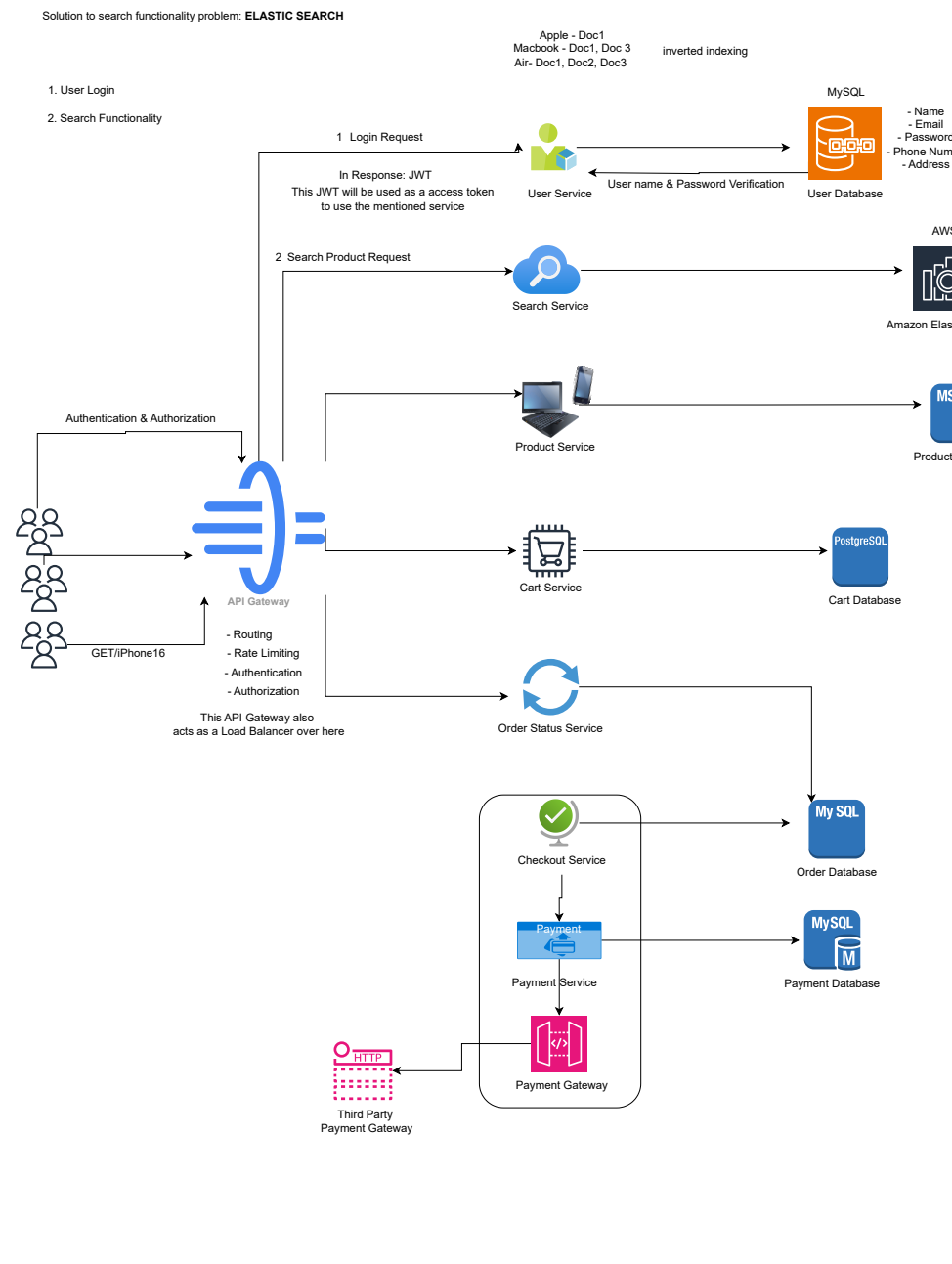
2. As a solution we can implement INDEXING here, but still database scanning is not prevented.

3. O(n) - time

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

1. User Login

2. Search Functionality



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

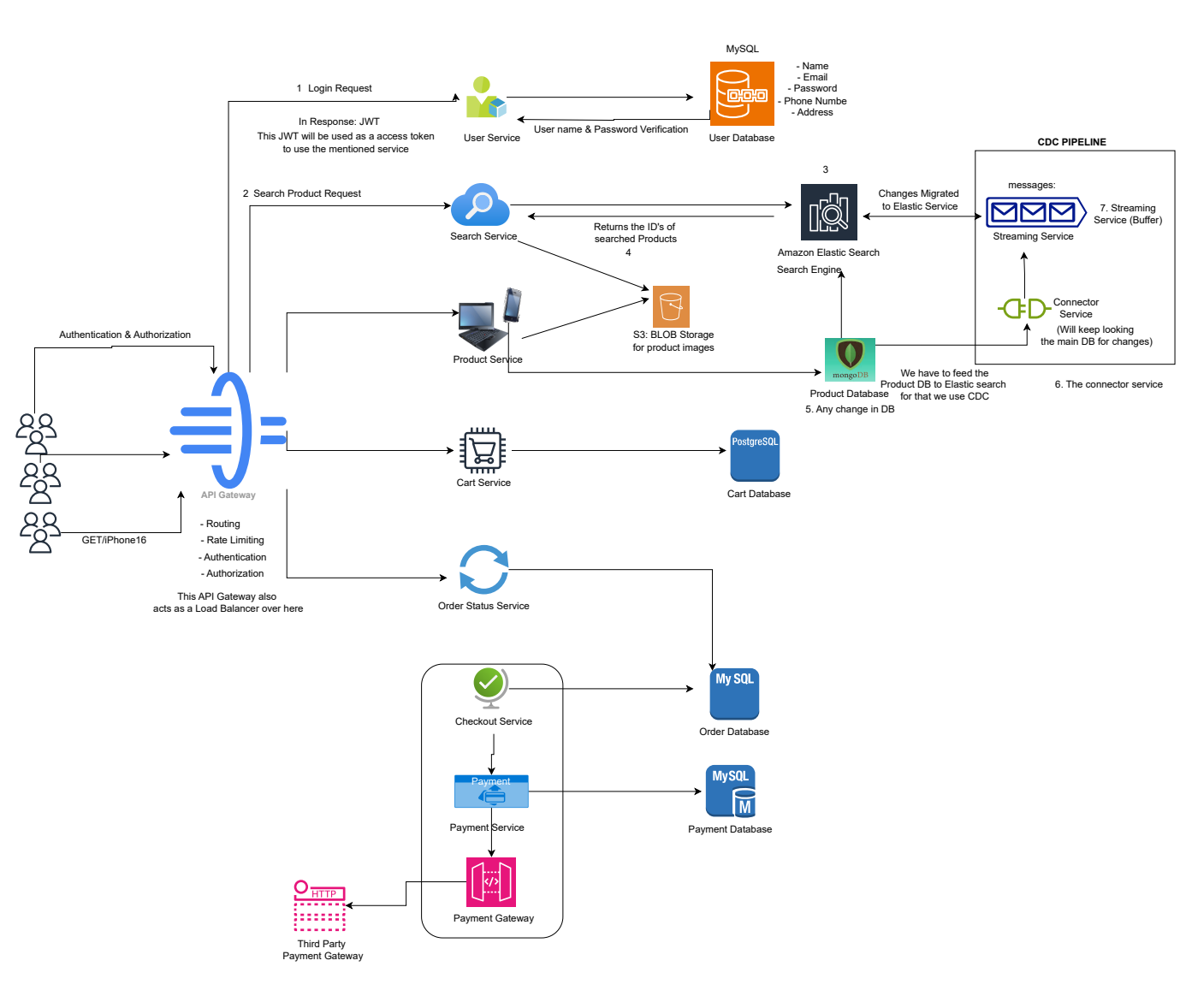
Word	Document IDs
Apple	{Doc 1, Doc 3}
Macbook	{Doc 3}

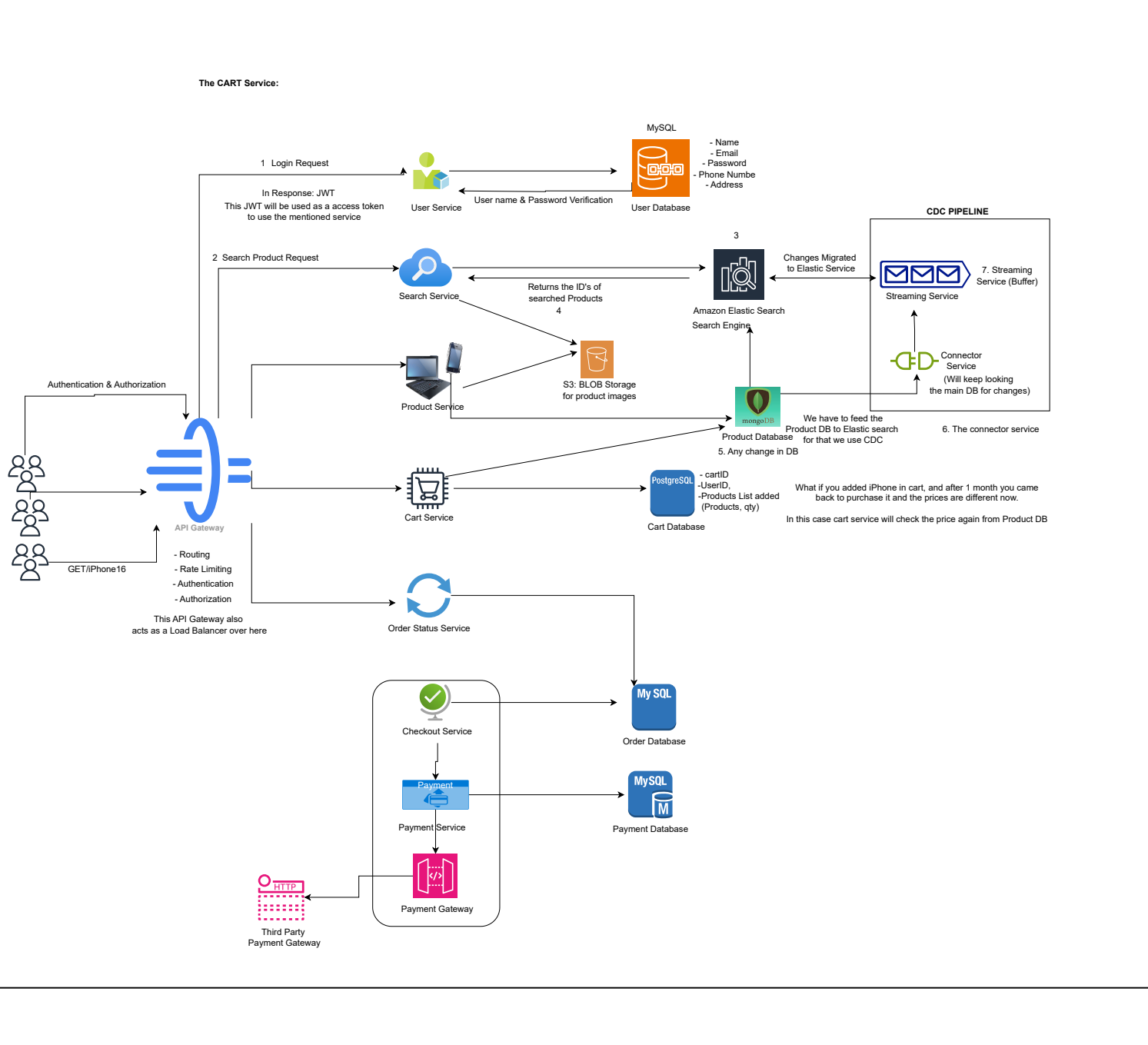
o(1)

```
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.
}
```

But the question is, How the data from Original database goes into Elastic Search in real time ????

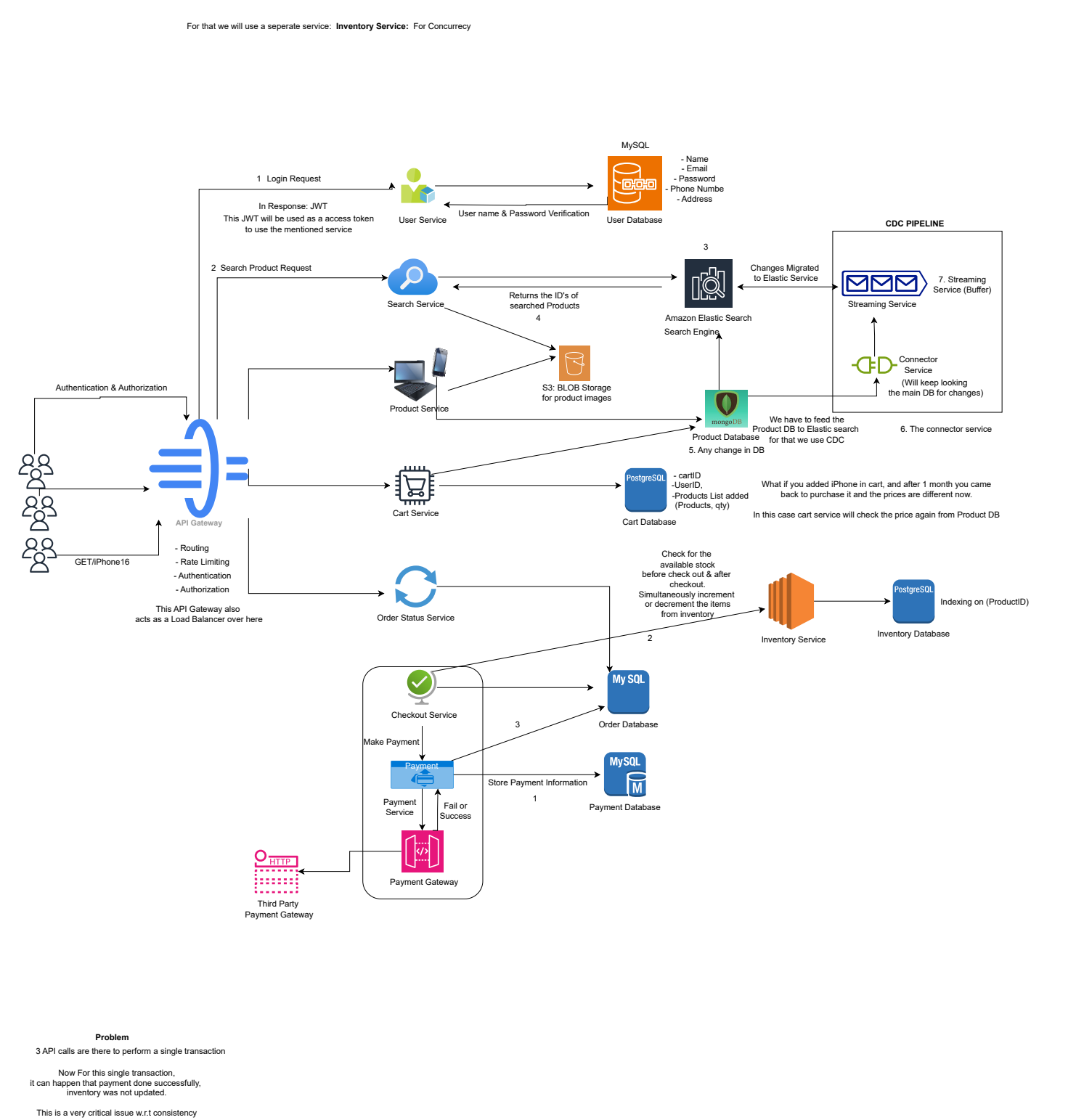
We use a technique known as : CDC (Change Data Capture)





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 Concurrecy



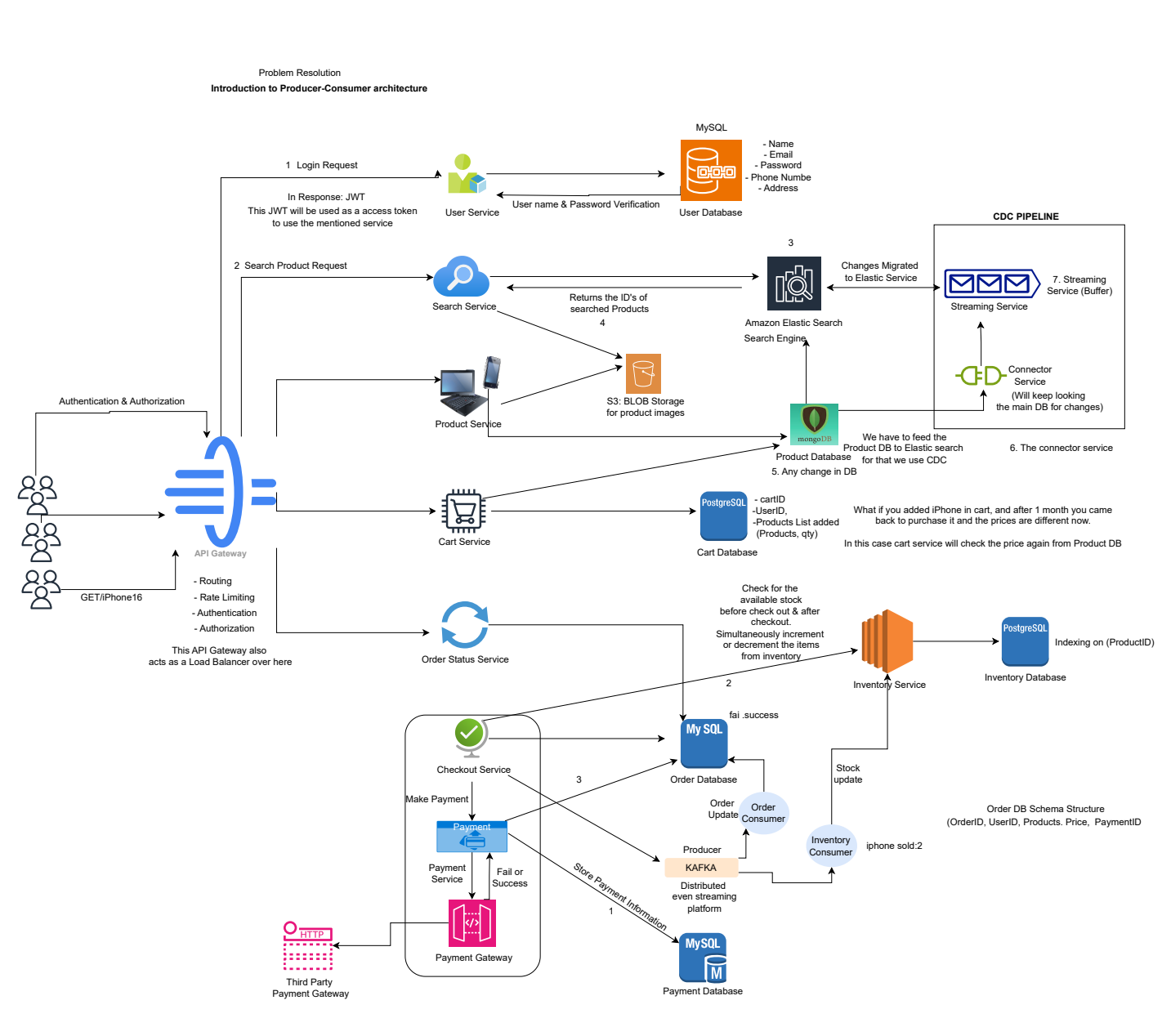
Problem
3 API calls are there to perform a single transaction

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

Problem Resolution
Introduction to Producer-Consumer architecture



Order DB Schema Structure
(OrderID, UserID, Products. Price, PaymentID

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