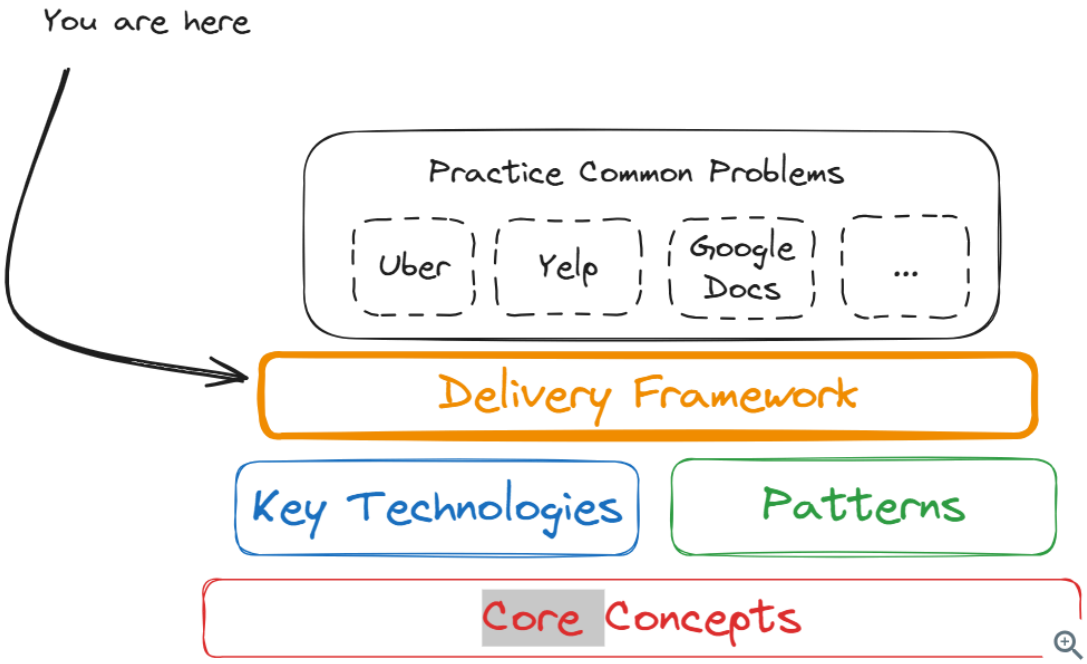
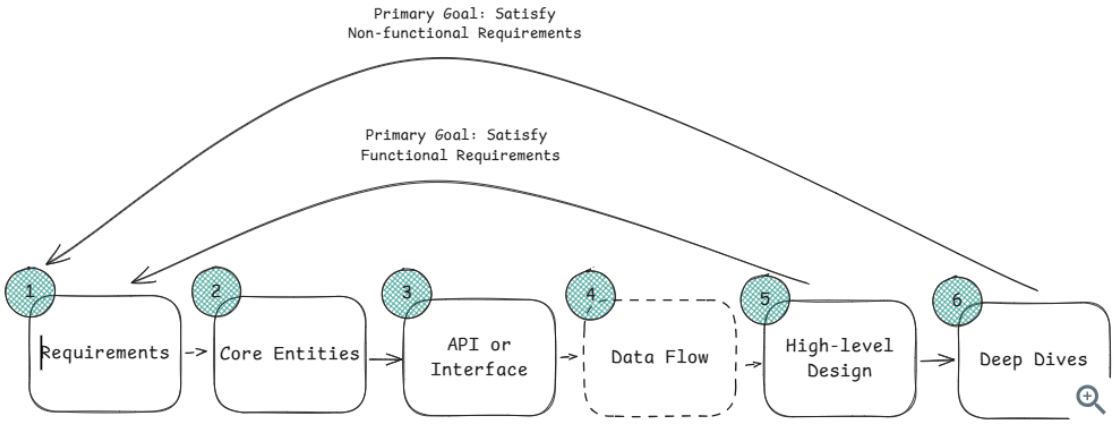
Delivery Framework



Overall Structure

Here's the framework!



Recommended system design interview structure

⭐ What This Diagram Represents

The diagram shows a **recommended structure** for answering a system design interview.
It breaks your entire thought process into **6 clean steps**, flowing left → right:

1️⃣ Requirements
2️⃣ Core Entities
3️⃣ API / Interface
4️⃣ Data Flow
5️⃣ High-Level Design
6️⃣ Deep Dives

Two overarching goals run across the top:

- **Satisfy Functional Requirements**
- **Satisfy Non-functional Requirements (Scalability, Reliability, Latency, etc.)**

This framework prevents you from jumping randomly between points and ensures you present a **logical, organized HLD**.

🟦 Step-by-Step Explanation of Each Block

1️⃣ Requirements (~5 minutes)

This is the foundation of your HLD.

You identify:

Functional Requirements

- What should the system *do*?
- Core features?
- Must-have vs optional?

Example: *Design Instagram*

- Upload posts
- View feed
- Like/comment
- Follow users

Non-Functional Requirements

- Scalability
- Availability
- Latency constraints
- Consistency vs availability choices (CAP)
- Fault tolerance

This sets the boundaries for your design.

2️⃣ Core Entities (~2 minutes)

Here you define the major **data models or objects** in the system.

For Instagram:

- User
- Post
- Comment
- FollowGraph

For Uber:

- Driver

- Rider
- Trip
- Location

Purpose:
➡️ Helps you keep your design structured
➡️ Ensures you don't forget key modules later

You don't go too deep into schema here—just identify the essential entities.

3️⃣ API or Interface (~5 minutes)

Now define how the system will be used from the outside.

Think of it as:

"What endpoints or API calls do we expose?"

Example: *Notification System HLD*

- `POST /sendNotification`
- `GET /fetchNotifications`
- `GET /notificationCount`

Example: *URL Shortener*

- `POST /shortenURL`
- `GET /{shortCode}`

Why this step matters:

- Helps you reason about inputs/outputs

- Forces clarity before designing internals
- Interviewers love candidates who treat the system like a real service with APIs

---

4️⃣ Data Flow

This block explains **how data moves through your system**.

This is extremely important in system design.

You walk through the lifecycle of a request:

Example: *Upload Post in Instagram*

1. Client uploads image
2. API Gateway receives request
3. Storage Service stores media in S3/HDFS
4. Metadata Service stores post record in DB
5. Feed Service fans-out post to followers' feeds
6. Cache (Redis) updated
7. CDN distributes image

This step ensures the interviewer KNOWS you understand how components interact—not just naming components.

5️⃣ High-Level Design (HLD)

This is the **main architecture diagram**—the heart of your system design answer.

Here you draw:

- Load balancers

- API Gateway
- Microservices
- Databases
- Caches
- Message queues
- Storage layers
- Replication
- Partitioning strategies

Interviewers judge your seniority mainly from this part.

Purpose:
✔️ Shows system structure
✔️ Shows how you satisfy scalability
✔️ Shows actual architecture choices

6️⃣ Deep Dives

After presenting the HLD, you choose **one or two components** to explore deeply.

This is where 90% of strong candidates shine.

Examples of deep-dives:

- Sharding algorithm for the database
- Cache invalidation strategy
- Consistency model (strong vs eventual)
- Designing the Feed Ranking algorithm
- Message Queue retry + dead-letter queues

- How to run distributed transactions

This shows that you can take one part of a large system and design it thoroughly.

🎯 Why This Framework Is Powerful

Most candidates answer system design in a chaotic order:

- They jump to databases too early
- They pick random technologies without requirements
- They describe components without explaining data flow

This framework forces structure:

- **Requirements first**
- **Data model next**
- **External interface next**
- **Then architecture**
- **Then deep dive**

It's clean, professional, and matches how real engineers design systems.

💡 How to Use This Framework in an Actual Interview

When asked "Design X", you follow this flow:

Step 1 → Clarify requirements

Step 2 → Identify core entities

Step 3 → Define APIs

Step 4 → Explain data flow

Step 5 → Draw high-level architecture

Step 6 → Deep dive into components

Your answer becomes confident, structured, and easy to follow.

🧭 Would you like a real example?

I can walk you through this exact 6-step framework for:

- **Instagram HLD**
- **Twitter HLD**
- **WhatsApp HLD**
- **Notification Service HLD**
- **URL Shortener HLD**
- **E-commerce HLD**
- **Uber / Ola HLD**
- **Food Delivery HLD (Zomato/Swiggy)**

🌐 E-Commerce System Design (HLD)

(Using the HelloInterview 6-step Framework)

1️⃣ Requirements (~5 minutes)

Functional Requirements

1. User Management
    - Sign up, login, user profile
    - Save addresses, payment methods
2. Product Catalog
    - View product list
    - Search & filter products
    - View product details
3. Cart Service
    - Add/remove/update items
    - Maintain user cart session
4. Order Processing
    - Place order
    - Payment processing
    - Order confirmation
5. Inventory Management
    - Track stock levels
    - Reserve inventory at order time
6. Delivery / Logistics
    - Estimated delivery time
    - Track order status
7. Notification Service

    - Emails, SMS, push notifications

Non-Functional Requirements

- **High availability** (0.999 or better)
- **Low latency** for search < 200 ms
- **Scalable catalog** (millions of products)
- **Strong consistency** for payments & inventory
- **Eventual consistency** for reads like product views
- **Fault tolerance** (multi-region optional)
- **Horizontal scalability** for traffic peaks

2️⃣ Core Entities (~2 minutes)

You identify the main data models of the system:

User

- userId
- name, email
- address[]
- paymentMethod[]

Product

- productId
- title, description
- category
- price
- images
- reviews

Inventory

- productId
- availableQty
- reservedQty

Cart

- cartId
- userId
- items[] (productId, qty)

Order

- orderId
- userId
- items
- totalPrice
- status (CREATED → PAID → SHIPPED → DELIVERED)

Payment

- paymentId
- orderId
- paymentStatus

Thinking in entities helps structure APIs and storage.

3️⃣ API / Interface (~5 minutes)

Here we define all user-facing endpoints.

User APIs

- `POST /signup`
- `POST /login`

Catalog APIs

- `GET /products?query=shoes&sort=price`
- `GET /products/{productId}`

Cart APIs

- `POST /cart/add`
- `POST /cart/remove`
- `GET /cart`

Order APIs

- `POST /order/create`
- `GET /order/{orderId}`

Payment APIs

- `POST /payment/initiate`
- `POST /payment/verify`

Tracking APIs

- `GET /order/{orderId}/tracking`

This step clarifies what the system exposes before diving into architecture.

4️⃣ Data Flow (~4 minutes)

Let's walk through **placing an order**, which is the core flow.

Flow: User Places an Order

1️⃣ User clicks "Place Order"

- Request hits **API Gateway** → **Order Service**

2️⃣ Order Service calls Cart Service

- Fetches all items + quantities

3️⃣ Order Service calls Inventory Service

- Try to **reserve** stock (atomic operation)
- If not enough stock → fail order

4️⃣ Order Service creates ORDER record with status = CREATED

5️⃣ Payment Service initiates payment

- Payment gateway handshake
- On success → send event “PAYMENT_SUCCESSFUL”

6️⃣ Order Service updates status → PAID

7️⃣ Shipping Service creates shipment

- Assigns delivery partner
- Updates tracking number

8️⃣ Notifications sent via Notification Service

- Order confirmed email
- Payment success SMS

This demonstrates the full lifecycle clearly.

5️⃣ High-Level Design (HLD)

Here is the architecture breakdown:

🟥 Client Layer

- Web app, mobile app
- Talks to API Gateway

🟦 API Gateway

- Request routing
- Rate limiting
- Authentication

🟩 Microservices Layer

1. User Service

- Manage accounts
- Authentication
- Stores user profile data

2. Product Catalog Service

- Handles search, categories, product browsing
- Uses Elasticsearch/Solr for fast search
- Product DB in SQL/NoSQL (Postgres, MongoDB, or DynamoDB)

3. Inventory Service

- Strong consistency required

- Reserve stock during checkout
- Avoids overselling
- DB: SQL with row-level locking or Redis + DB

4. Cart Service

- Session-based
- High read/write volume
- Store in Redis (fast, in-memory)

5. Order Service

- Central workflow engine
- Writes orders to SQL
- Produces Kafka events for asynchronous processing

6. Payment Service

- Gateway integrations (Stripe/Paytm/Razorpay)
- Idempotent operations
- Uses a secure store for tokens

7. Shipping/Logistics Service

- Assign delivery partner
- Track order status

8. Notification Service

- Sends SMS, email, push
- Uses event-based messaging

🟧 Databases & Storage

Product Catalog DB

- NoSQL (MongoDB / DynamoDB)
- Also indexed in Elasticsearch for search

User DB

- Relational or NoSQL

Inventory DB

- SQL for strong ACID updates
- Or Redis + DB double write

Order DB

- SQL (MySQL/Postgres) with partitioning
- Order is financial → strong consistency

Cart DB

- Redis (temporary storage)

🟨 Message Queue

(Kafka / RabbitMQ)

Used for:

- Payment events
- Order events
- Notification triggers
- Inventory adjustments

🟪 Caching Layer

- Redis / Memcached for
    - Product info cache
    - Price cache
    - User session cache

🟫 CDN

- Delivers product images
- Static content delivery

6️⃣ Deep Dives (choose 1–2)

This is where you show senior-level thinking.

Let's deep dive into **Inventory Consistency** because it's crucial for ecommerce.

🔍 Deep Dive: Inventory Service (Avoid Overselling)

Required properties:

- Atomic operations
- Strong consistency
- Very high throughput

Possible Designs:

Option A: SQL Row Locking

```
UPDATE inventory

SET availableQty = availableQty - 1

WHERE productId = X AND availableQty > 0;
```

- If 0 rows updated → Out of stock
- Simple, strong consistency
- Bottleneck: single DB instance must scale vertically or shard by productId

---

Option B: Redis + Lua Script (Atomic)

Use Redis to maintain counters.

Lua script for atomic decrement:

```
if redis.call("GET", key) > 0 then

   redis.call("DECR", key)
```

```
    return 1
else
    return 0
end
```

- Extremely fast
- Needs synchronization with persistent DB

Option C: Eventual Consistency with Reservation Token

1. Reserve stock
2. Create reservation record with TTL (like a lock)
3. If payment fails → release reservation

Used by Amazon-scale systems.

🔍 Deep Dive: Search System

- Product data stored in NoSQL
- Indexed in Elasticsearch
- Query goes to ES cluster
- Supports prefix matching, filters, facets

🔍 Deep Dive: Order Lifecycle Workflow

Using an orchestrator (like AWS Step Functions or custom state machine):

CREATED → PAYMENT_PENDING → PAID → PACKED → SHIPPED → DELIVERED

Each transition triggered by events (Kafka).

🎯 Final Summary (E-commerce HLD)

Step	What You Covered
Requirements	Functional + non-functional clarity
Entities	Product, user, cart, order, inventory
APIs	Catalog, cart, order, payment
Data Flow	Full order lifecycle
HLD	Microservices + DB + cache + queues
Deep Dives	Inventory consistency, search, order workflow

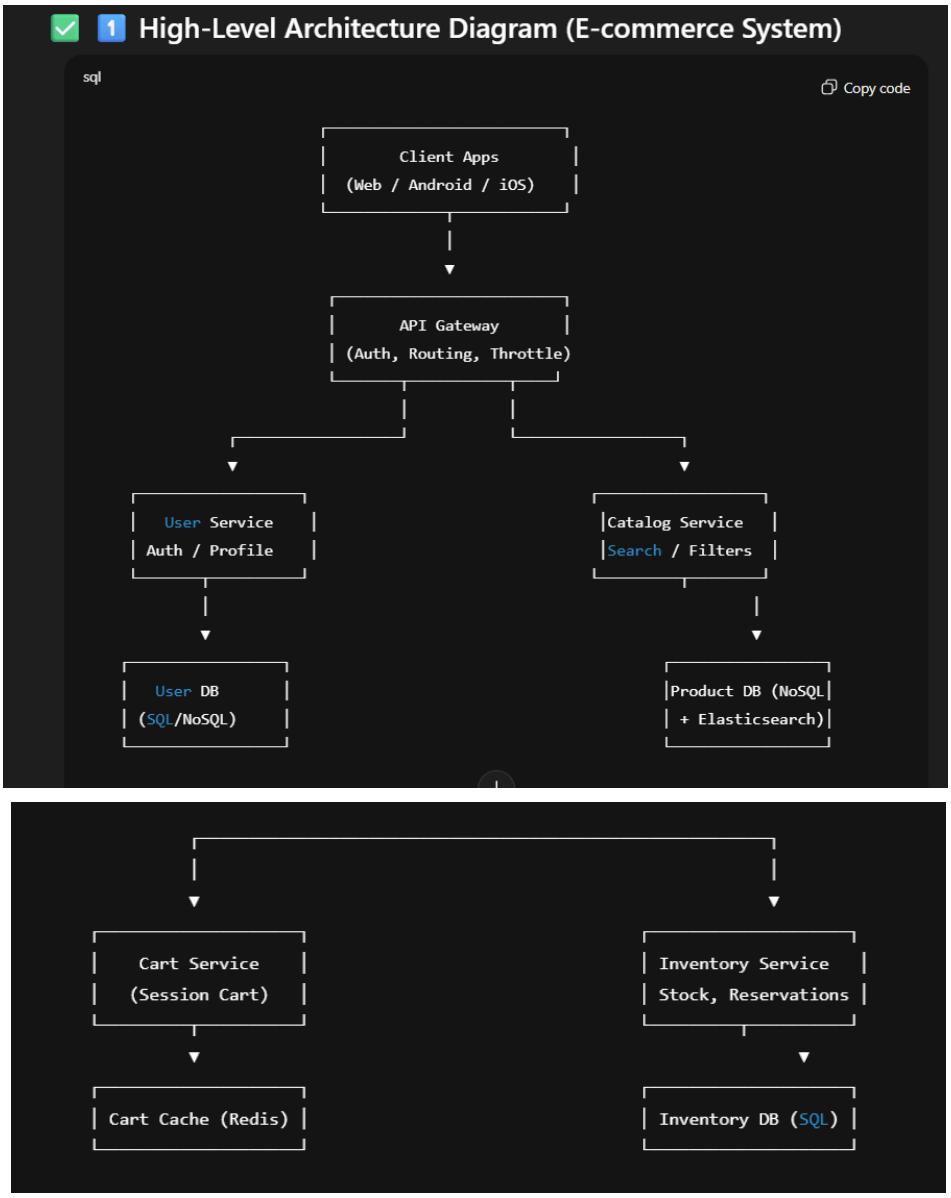
This framework produces a **strong, structured, professional HLD**, exactly what interviewers expect.

Want me to create a diagram of this architecture?

I can create:

- High-Level Architecture diagram
- Sequence diagram for order placement
- Database schema diagrams
- Component breakdown

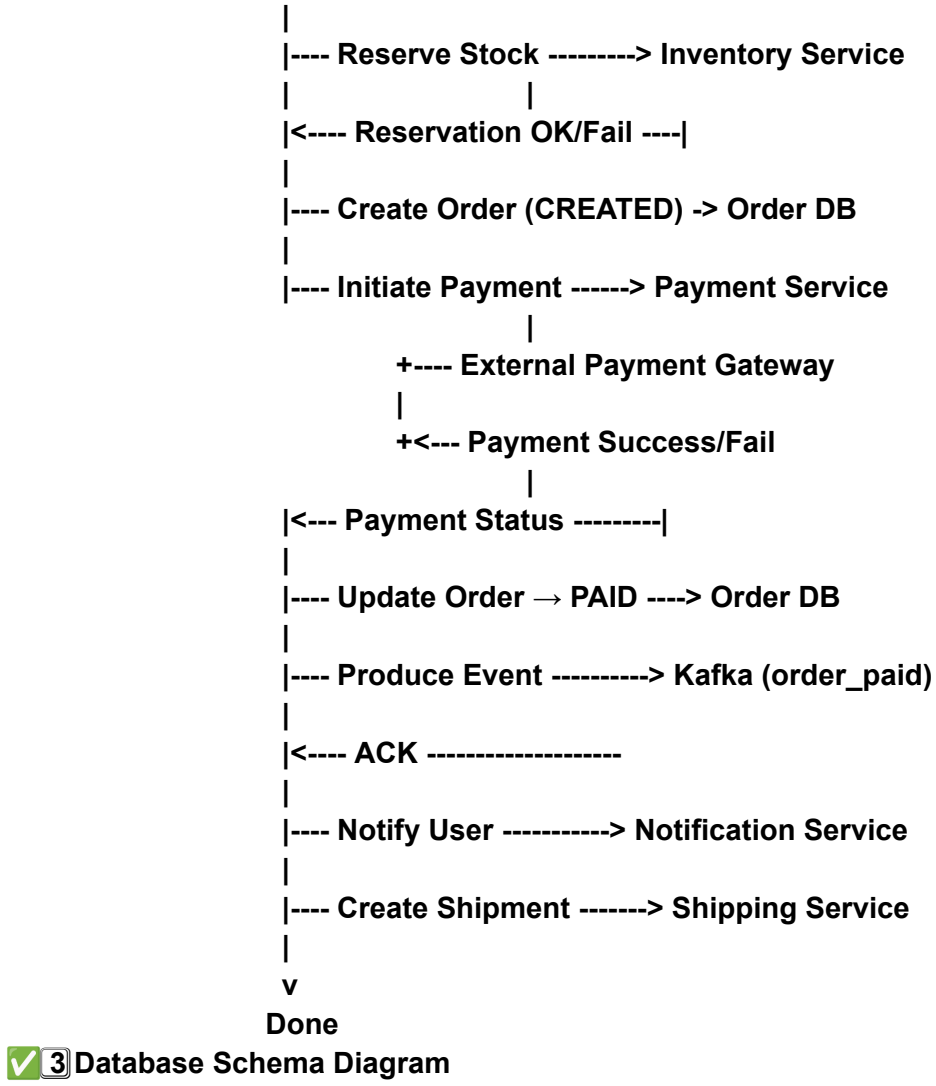
✅1️⃣ High-Level Architecture Diagram (E-commerce System)



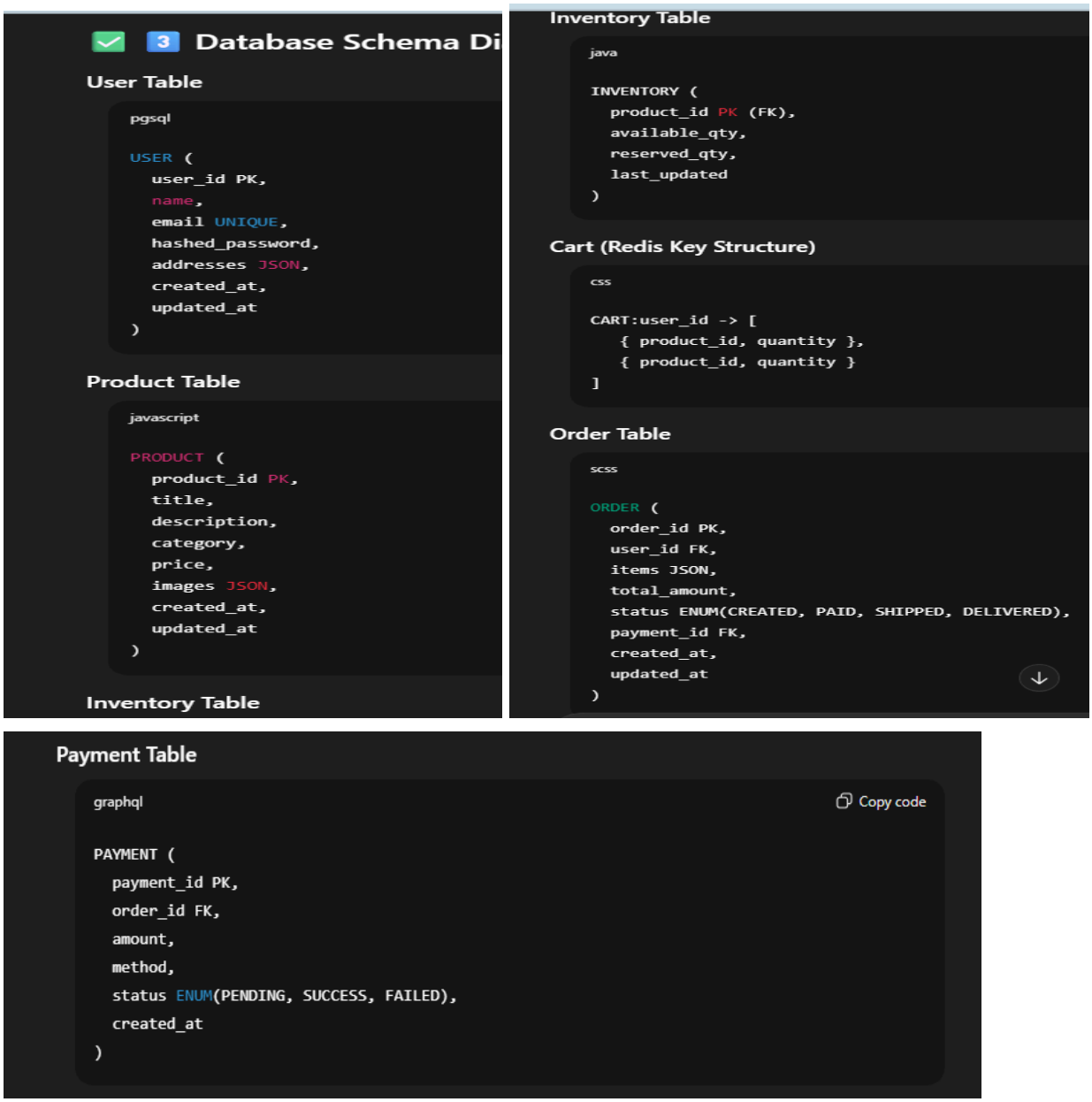
Copy code

✅2️⃣ Order Placement Sequence Diagram

```
User
 |                    API Gateway
 |                        |
 |--- Place Order ---> |
                          |
                          v
                    Order Service
                          |
                          |---- Get Cart ------------> Cart Service
                          |                              |
                          |<----- Cart Items -----------|
```

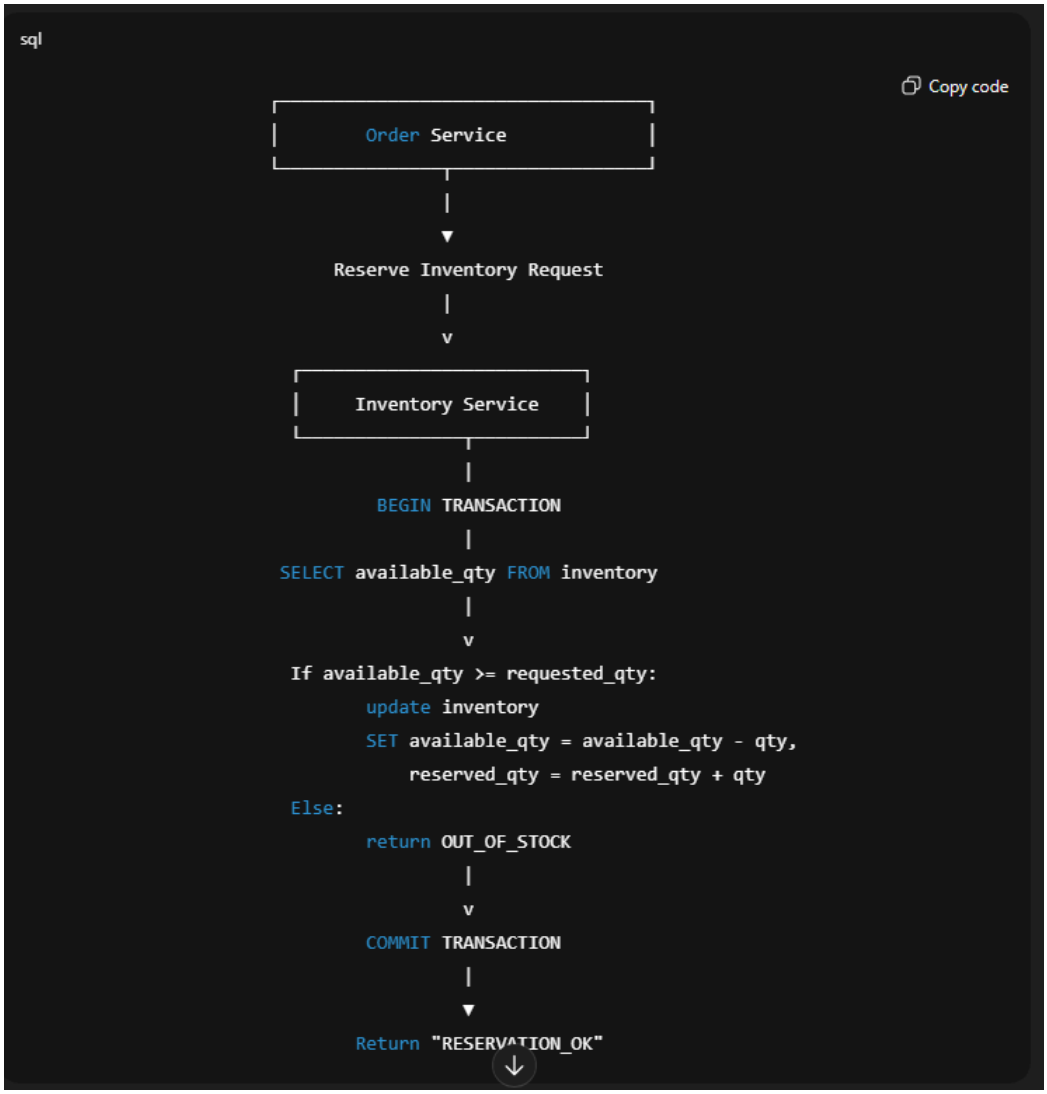


✅3️⃣ **Database Schema Diagram**



✅4️⃣ Deep Dive Diagram – Inventory Reservation Flow

This shows strong consistency for stock updates.



Guarantees:

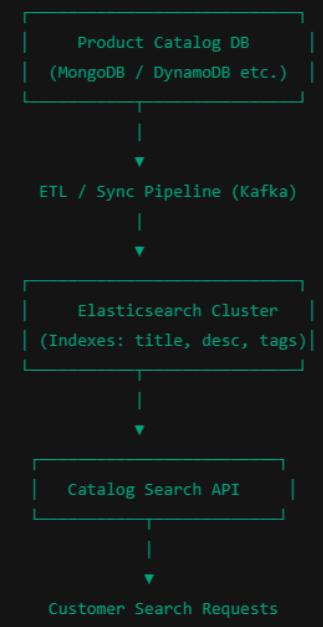
- **No overselling**
- **Atomic decrement**
- **Consistent stock state**

⚙️5️⃣ Deep Dive – Search Architecture Diagram (Elasticsearch)

5 Deep Dive – Search Architecture Diagram (Elasticsearch)

yaml

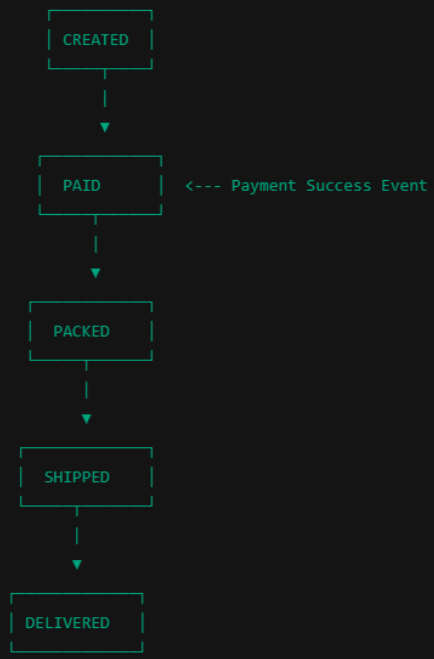
Copy code



6 Deep Dive – Order Lifecycle State Machine

yaml

Copy code



Events drive transitions:

- `payment_success`
- `shipment_created`
- `out_for_delivery`
- `delivered`

🎉 All diagrams are now complete!

These diagrams cover:
✔ High-level architecture
✔ Detailed order workflow
✔ DB schema
✔ Search architecture
✔ Inventory consistency
✔ Order lifecycle

You can directly use these in interviews, Google Docs, Notion notes, or whiteboard preparation.

🌟 Requirements Gathering in System Design

When you start a High-Level Design (HLD) interview, the first 5 minutes usually make or break your entire solution.

Your goal:
👉 *Understand exactly what system you're designing*
👉 *Clarify and prioritize what matters the most*
👉 *Convert vague problem statements into actionable requirements*

We break requirements into two major categories:

1️⃣ Functional Requirements (FR)

Functional requirements describe what the system should do — the core features.

Think of them as:

> **"Users/Clients should be able to …"**

Your job is to ask questions, refine the scope, and capture the top 3–4 essential features.

✔ Example 1: Twitter-like System

Suppose the interviewer says:
"Design Twitter."

You refine it by asking:

- **Should users be able to post text only? Images? Videos?**
- **Should they be able to follow each other?**
- **Do we need a recommendation timeline or just chronological?**
- **Should retweets/likes be included?**

Then you agree on the priority features, such as:

Core Functional Requirements

1. **Users should be able to post tweets.**
2. **Users should be able to follow/unfollow other users.**
3. **Users should be able to view a timeline containing tweets from the users they follow.**

That's enough to build a scalable Twitter.

✔ Example 2: E-Commerce System (Amazon-like)

🎯 **Problem:** ***Design an e-commerce system.***

You ask clarifying questions:

- **Do we need cart + checkout?**
- **Should we support product search?**
- **Is order tracking required in this stage?**
- **Do we allow sellers to list products?**

After discussion, you capture the core FRs:

Core Functional Requirements

1. **Users should be able to search and browse products.**
2. **Users should be able to add items to a cart and checkout.**
3. **Users should be able to place orders and view order history.**

Optional features you may explicitly deprioritize:

- **Wishlists**
- **Reviews**
- **Seller dashboards**
- **Offers and coupons**

This shows focus, something interviewers evaluate heavily.

✔ Example 3: Distributed Cache (Redis-like)

Problem: ***Design a distributed cache.***

Functional requirements may be:

1. **Clients should be able to insert key-value pairs.**
2. **Clients should be able to retrieve values by key.**
3. **Clients should be able to set expirations (TTL).**

That's enough to design a production-level cache.

⭐ Why Keeping FRs Short Helps

If you list 15 features, you won't have time to design the system.
FAANG interviewers judge you on:

- **Prioritization**
- **Ability to simplify**
- **Knowing what matters most**

Stick to 3–4 mandatory features, unless the interviewer expands scope.

2️⃣ Non-Functional Requirements (NFR)

Non-functional requirements describe qualities the system must have.

Think of them as:

> **"The system should be able to…"**
> **"The system should be…"**

These determine architecture choices like:

- **databases**

- **caching**
- **replication**
- **load balancing**
- **partitioning**

✔ Example 1: Twitter NFR

1. **Highly Available → Downtime is unacceptable**
2. **Horizontal Scalability → 100M+ daily active users**
3. **Low Latency Feed Rendering → < 200ms timeline load**

Quantifying it ("< 200 ms") makes it meaningful.

✔ Example 2: E-Commerce System NFR

When designing Amazon-type systems:

Key NFRs

1. **High Availability → People shop from everywhere, 24/7**
2. **Strong Consistency for Payment and Inventory**
    - **You cannot oversell a product**
3. **Scalable Search**
    - **Product search must return results in < 300 ms**
4. **Durability → Orders and payments *must never* be lost**

5. **Security & Compliance**
    - **Handle customer payment info securely**
    - **PCI-DSS compliance**

NFRs shape your architecture:

- **You will choose SQL+cache for accuracy**
- **You will use message queues for order flows**
- **You will replicate services for reliability**

✔ Example 3: Distributed Cache NFR

1. **Low latency (< 5ms reads)**
2. **High throughput (millions of ops/sec)**
3. **Best-effort consistency (Availability > Consistency)**
4. **Partition tolerance (cluster mode required)**

Here, CAP theorem influences your design.

🧠 Checklist for Choosing NFRs

Here are practical categories you should consider for any system:

🔸 CAP Theorem

- **Availability or Consistency — which is more important?**
- **Partition tolerance is always assumed in distributed systems.**

Examples:

- **Twitter feed → prioritize availability**
- **Banking → prioritize consistency**

🔸 Scalability

Questions to ask:

- **Are reads heavier than writes?**
- **Will there be spikes? (During sales, Black Friday, Diwali etc.)**

E-commerce usually:

- **Heavy reads (product search, browsing)**
- **Moderate writes (orders)**

🔸 Latency

Where do we need fast responses?

Examples:

- **Search (< 500ms)**
- **Payment confirmation (< 2s)**
- **Live chat (< 100ms)**

🔸 Durability

How critical is data safety?

Examples:

- **Banking → extremely high durability**
- **Twitter tweets → can tolerate minor loss (not ideal though)**

🔸 Security & Compliance

When important?

- **Handling personal data**
- **Handling payments**
- **Storing medical history**

Examples:

- **E-commerce → PCI-DSS**
- **Healthcare → HIPAA**

🔸 Fault Tolerance

Questions:

- **What happens if a server fails?**
- **Do we need automatic failover?**

- **Do we need replication?**

🔸 Environment Constraints

Examples:

- **Mobile networks with low bandwidth**
- **Edge devices with limited CPU/RAM**
- **IoT sensors with battery limitations**

🎯 Putting It All Together – Full Example (E-Commerce)

Below is a complete requirements section you'd present in an interview:

Functional Requirements

1. **Users should be able to search and browse products.**
2. **Users should be able to add items to a cart and checkout.**
3. **Users should be able to place orders and track order status.**

Non-Functional Requirements

1. **System should be highly available (99.9% uptime).**
2. **System should support scalable search with response time < 300ms.**

3. **Strong consistency for inventory & payments.**
4. **High durability for orders (data loss unacceptable).**
5. **System must follow payment security standards (PCI-DSS).**

✔ Why This Matters

These requirements directly shape your system architecture.

If inventory must be strongly consistent → use SQL with locking or distributed transactions.
If search must be fast → introduce Elasticsearch.
If availability > consistency → replicate data and use eventual consistency.

Interviewers WANT TO SEE THIS KIND OF THINKING.

🔹 What Capacity Estimation Really Means in System Design

Most system design guides make you do heavy “back-of-the-envelope” calculations before even starting the design.
But the idea behind the paragraph you shared is:

👉 Only do estimation when it meaningfully affects the architectural choices you make.

Not for showing off arithmetic.

This is important because many candidates waste 5–10 minutes calculating DAU, storage, bandwidth, etc., and then conclude:

> “So the system is large… okay.”

This does *not* help the interviewer assess your design thinking.

🔹 Why Over-Estimating Is Not Always Useful

If you're building something like:

- URL shortener
- Chat system
- Notification service
- Logging pipeline

You already know it must scale horizontally.

Saying “1 billion requests/day” doesn’t change your design much. Your design will *still* require:

- distributed storage
- load balancing
- sharding or partitioning
- replication
- queues or streams

So those extra calculations don't help the interviewer understand your thought process.

🔹 When Estimations *Actually Matter*

Estimation becomes important **only when the decision depends directly on the numbers**.

Here are clear examples:

✅ Example 1: Top-K Trending Topics System

If Facebook/Twitter wants:

- “Top 10 trending hashtags in last 10 minutes”

Estimation is needed because **your choice of data structure depends on expected size**.

Why estimation matters?

To decide if you can store all candidate hashtags in:

- A **single min-heap** (if tens or hundreds of thousands)
- Or a **sharded distributed system** (if millions or billions)

This **significantly** changes the design:

- A single machine heap? → simple design
- Multi-shard merge of heaps? → complex distributed design, faults, aggregation service, etc.

✅ Example 2: Feed Generation (Instagram/Twitter)

If you need to “fan-out” posts to users’ home timelines, you estimate:

- average number of followers
- peak posting rate

This helps you decide:

- **Fan-out on write** vs **fan-out on read**
- Need for message queues
- How many feed storage shards.

Here, numbers influence architectural strategy.

✅ Example 3: Logging / Metrics System

If logs are coming in at **10 TB/day**, HDFS storage, compression choices, retention policy all change.

Examples Where Estimation Does Not Change Anything

❌ URL Shortener

Even if you estimate 200M URLs per day:

- You'll still use a distributed KV store
- Still auto-increment or hash IDs
- Still replicate and shard

Numbers do not change the overall shape of the architecture.

❌ Messenger (e.g., WhatsApp architecture)

Even if messages are:

- 50M / hour or
- 100M / hour

You would still use:

- message queues
- sharding
- distributed storage
- async replication

No major decision changes.

🔹 How to Smartly Communicate This in an Interview

Here's a polished, professional way to express this:

> "I prefer to skip detailed capacity estimations upfront and instead calculate numbers only when they influence design choices. As I design components, I'll estimate relevant quantities to decide things like sharding strategy, storage choice, or data structure feasibility."

This shows:

- confidence
- time-efficiency
- senior-level reasoning
- awareness of trade-offs

🔹 Why Learning Estimation Is Still Useful

Even if you skip heavy math in interviews, being able to quickly estimate helps you reason about:

- bottlenecks
- throughput needs
- partitioning strategy
- when to use a distributed system vs single-node structure
- feasibility of caching layers
- latency and QPS limits

It's a thinking tool — **not an exam question**.

🔹 Summary (High-Level Takeaway)

What Most Candidates Do	What Good Systems Designers Do
Do storage/QPS math for everything	Do estimation **only when design needs it**
Impress interviewer with arithmetic	Impress interviewer with reasoning and design trade-offs
Waste time calculating obvious numbers	Focus time on main architecture decisions
Try to be perfect at math	Use rough numbers only when they drive decisions

.

🌟 Core Entities (with Proper Examples)

Once you finish requirements (Functional + Non-Functional), the **next step** is to identify the **core entities** of the system.

These are the **main nouns / objects / data units** your system will store, process, or exchange via APIs.

⭐ Why Identify Core Entities?

- They help you understand what your system fundamentally *is about*.
- They give you a starting point for your *data model*, *APIs*, and *storage design*.
- They keep the conversation organized and prevent overthinking too early.

- They allow you to iterate: you will add or refine them later as requirements become clearer.

You **do NOT** define tables, columns, schemas yet — just the *names and purpose* of each entity.

🧠 How to Identify Core Entities?

Ask two quick questions:

1. Who are the actors in the system?

- User?
- Seller?
- Admin?
- Driver?
- Courier?

2. What are the important nouns required to satisfy the functional requirements?

Examples:

- Tweet
- Order
- Product
- Payment
- Cart
- ChatMessage

These nouns become *entities*.

✔ Example 1: Twitter

Functional requirements:

- Post tweets
- Follow users
- View a timeline

From this, we derive:

Core Entities

1. **User**
   Represents a Twitter user.
2. **Tweet**
   The content posted by a user.
3. **Follow**
   Represents a relationship: User A follows User B.

That's enough to start designing Twitter.

✔ Example 2: E-Commerce System (Amazon-like)

This is the most important example for interviews.
Requirements:

- Search products
- Add to cart

- Place orders
- Track order status

🔍 Step-by-step thinking

Actor?

- Customer
- Seller (optional in basic version)

Key nouns from FRs:

- Product
- Cart
- CartItem
- Order
- OrderItem
- Payment
- Inventory

Now we list them cleanly:

Core Entities for E-Commerce

1. User

The customer using the platform.

2. Product

Represents a purchasable item on the platform.

3. Inventory

Tracks available stock for each product.
Critical because consistency matters.

4. Cart

A temporary shopping basket linked to a user.

5. CartItem

Each entry in the cart representing (product, quantity).

6. Order

A confirmed purchase.

7. OrderItem

Products included in an order.

8. Payment

Tracks payment status, amount, method.

🎯 Final list (this is what you show to the interviewer):

E-Commerce Core Entities

- User
- Product
- Inventory
- Cart
- CartItem
- Order

- OrderItem
- Payment

This is clean, structured, and shows complete understanding.

✔ Example 3: Instagram

Requirements:

- Users post photos
- Users follow each other
- Users like/comment on posts

Core Entities

1. **User**
2. **Post (Photo)**
3. **Follow**
4. **Like**
5. **Comment**

✔ Example 4: Uber / Ride-Hailing App

Requirements:

- Book a ride

- Match driver to rider
- Track trip

Core Entities

1. **Passenger**
2. **Driver**
3. **Vehicle**
4. **RideRequest**
5. **Trip**
6. **Location**

✔ Example 5: Distributed Cache (Redis)

Requirements:

- Put key-value
- Get key-value
- TTL expiration

Core Entities

1. **CacheNode**
2. **Key**
3. **Value**
4. **TTL Metadata**

🌟 Why Not Build Full Data Model Now?

Because:

- You don't yet know all relationships
- You may revise your functional requirements
- You might add new entities later (e.g., "Review", "Notification")
- You save time by iterating **only when needed**

Interviewers love when candidates:

- Start small
- Expand as the design evolves
- Avoid premature optimization

⚡ Mini Exercise (I can evaluate your answer if you want)

Try to list core entities for:

"Design a Food Delivery App (like Zomato/Swiggy)"

Hint: List entities like "User", "Restaurant", "Order", etc.

🌟 API / System Interface (~5 minutes)

After identifying **core entities**, your next step is to define **how external clients interact with your system**.

This is where you design the **API contract**.

This contract guides the ENTIRE HLD:

- How clients call your system
- What data flows in/out
- What operations are supported
- How your backend processes requests

Before designing internal architecture, define **what the system exposes**.

1️⃣ Choose an API Style (Simple Rule)

You have 3 primary options:

✔ REST (Default)

Use REST **by default** in interviews unless given a special use case.

- Simple
- Standard HTTP verbs
- Great for CRUD
- Easy to scale
- Most interviewers expect REST

Example verbs:

- `GET` → read
- `POST` → create

- `PUT` → update
- `DELETE` → delete

✔ GraphQL (When Needed)

Choose GraphQL if:

- many types of clients (mobile, web, admin, internal dashboards)
- clients need *different subsets* of data
- want to avoid under-fetching/over-fetching

Example: A product page in e-commerce where different clients need different levels of product detail.

✔ RPC / gRPC (Internal Services)

Use RPC when:

- service-to-service communication
- extremely low latency required
- performance > human readability
- heavy internal calls (microservices)

Example: PaymentService ↔ OrderService in e-commerce.

2️⃣ Design the REST API (based on core entities)

Your API resources should be:

- **Plural nouns**
- **Based on the core entities** you listed earlier

✔ Example 1: Twitter API (based on User, Tweet, Follow)

Create a tweet

```
POST /v1/tweets
Body: {
  "text": "hello world!"
}
```

Get a tweet

```
GET /v1/tweets/{tweetId}
```

Follow a user

```
POST /v1/follows
Body: {
  "followee_id": "12345"
}
```

Get feed

```
GET /v1/feed
```

⚠ Important rule:

> The logged-in user (follower) is **never** taken from the request body.
> It is taken from the **auth token** (JWT, OAuth token).

✔ Example 2: E-commerce API (based on User, Product, Cart, Order, Payment)

Let's design APIs for the core features:

🛍️ Product APIs

Search products

```
GET /v1/products?query=iphone&category=electronics&page=1
```

Get product by ID

```
GET /v1/products/{productId}
```

🛒 Cart APIs

Add item to cart

```
POST /v1/cart/items
Body: {
  "product_id": "P123",
  "quantity": 2
}
```

Get current user's cart

```
GET /v1/cart
```

Update quantity

```
PUT /v1/cart/items/{cartItemId}
Body: {
  "quantity": 3
}
```

Remove item

```
DELETE /v1/cart/items/{cartItemId}
```

📦 Order APIs

Create order (checkout)

```
POST /v1/orders
Body: {
  "payment_method": "UPI"
}
```

The backend figures out:

- which user is ordering
- cart contents
- current inventory
- total amount

Get order details

```
GET /v1/orders/{orderId}
```

List user's orders

```
GET /v1/orders
```

💳 Payment API

Initiate payment

```
POST /v1/payments
Body: {
  "order_id": "O89123",
  "method": "CARD"
}
```

✔ Example 3: Ride-Hailing App (Uber)

Request a ride

```
POST /v1/rides
Body: {
  "pickup_location": {lat, long},
  "drop_location": {lat, long}
}
```

Driver accepts ride

```
POST /v1/rides/{rideId}/accept
```

Track the ride

```
GET /v1/rides/{rideId}/status
```

Complete ride

```
POST /v1/rides/{rideId}/complete
```

🔐 Why Do We *Never* Pass user_id in Request Bodies?

Because:

- User identity must come from the **authentication token**
- Passing userId allows spoofing/hacking
- Security best practice: user context comes from headers

Example header:

```
Authorization: Bearer <JWT token>
```

Backend decodes token → identifies user.
Client cannot lie about user identity.

This is a **must-know point** for SDE interviews.

⭐ How to Present APIs in Interviews (Pro-Level Style)

You say:

> "I will use REST as the primary API protocol.
> These endpoints are based on the core entities discussed earlier.
> All sensitive identity information (user, session) is derived from the auth token, never from request payload."

Then show 5–10 clean endpoints.

🎯 What This Section Achieves

Once API is clear, the interviewer knows:

- What your system must support
- How clients interact with it
- What data flows look like

This makes the high-level design much easier.

🌟 What is Data Flow?

Data Flow answers:

> "What *sequence of steps* does the system perform to turn input → output?"

It is NOT the system architecture yet.
It is like a high-level *story* of how your system behaves.

Best used when your system:

- processes data in multiple stages
- transforms data before storing
- uses asynchronous steps
- involves background workers

If your system is simple CRUD (e.g., ToDo App), you can skip this.

🌟 Format

Use a **simple numbered list** that shows the sequence of operations.

Example (given):
Web Crawler

1. Fetch seed URLs
2. Parse HTML
3. Extract URLs
4. Store data
5. Repeat

✔ Example 1: Twitter – Data Flow for "Publish Tweet"

(Useful because tweet delivery is multi-step)

Tweet Publish Data Flow

1. User sends `POST /tweets`
2. Service validates tweet size, checks auth
3. Store tweet in DB
4. Push tweet ID into Fanout Service
5. Fanout Service distributes tweet IDs to followers' timelines
    - For high-scale users, store separately ("pull from heavy users")
6. Update caches (user timeline, tweet cache)
7. Return success response

This helps justify later:

- Why you use Fanout

- Why write path includes queueing

✔ Example 2: E-Commerce – Data Flow for “Place Order”

This is an excellent interview demonstration because ordering is a multi-step workflow.

Order Placement Data Flow

1. User clicks “Place Order” → `POST /orders`
2. Backend validates cart contents
3. Check inventory service for stock availability
4. Reserve inventory (lock stock temporarily)
5. Calculate final price, tax, shipping
6. Initiate Payment Service
7. Payment Gateway returns success/failure
8. If success:
    - Convert cart → order
    - Deduct inventory permanently
    - Notify warehouse
    - Send order confirmation
9. If payment fails:
    - Release reserved inventory

    - Keep cart unchanged

This flow sets up your need for:

- Order Service
- Payment Service
- Inventory Service
- Message queues
- Event-driven processing

✔ Example 3: Instagram – Data Flow for "Upload Photo"

Photo Upload Data Flow

1. Client uploads image to Upload Service
2. Service validates image (size, format)
3. Store original in Object Storage (S3/GCS)
4. Trigger async worker to generate thumbnails
5. Store metadata in DB (userId, caption, URL)
6. Update followers' feeds asynchronously
7. Return post ID to user

Why this is useful:
You justify storage choices (object store), async processing (thumbnails), fanout approach.

✔ Example 4: Uber – Data Flow for “Book a Ride”

Ride Request Data Flow

1. User sends ride request with pickup/drop
2. Matching Service receives request
3. Fetch nearby drivers from Location Service
4. Choose best driver (distance, rating, ETA)
5. Send request to driver app
6. Driver accepts/declines
7. If accepted → create trip
8. Track live location during trip
9. Trip completes → calculate fare → payment

This prepares the interviewer for:

- Location service
- Real-time updates
- Matching algorithm
- Push notification system

✔ Example 5: Message Queue Processing System

(Like Log Processor, Analytics Pipeline)

Analytics Event Pipeline Data Flow

1. Client app sends event (click, view, purchase)
2. Ingestion Service receives event
3. Writes events to queue (Kafka)
4. Stream Processor aggregates (count, sum, group-by)
5. Writes processed data to data warehouse
6. Dashboards query warehouse

This flow tells the interviewer:

- You understand event streaming
- You differentiate hot path vs cold path

⭐ What NOT to Do

❌ Do NOT draw architecture here
❌ Do NOT describe every microservice
❌ Do NOT list dozens of steps

Keep it **high-level, sequential, clean**.

⭐ When Should You Include Data Flow?

Include it when your system has:

- Background workers

- Queues
- Multiple transformations of data
- Event processing
- Long workflows (checkout, uploads, etc.)

Skip it when your system is:

- CRUD
- Simple API → DB → Response pattern

I'll walk through everything step-by-step and connect it to **two examples**:

- mini-Twitter (like in the notes)
- a simple **e-commerce app** (Amazon-lite)

I'll also explain what's happening in the **HLD diagram** you shared.

---

1. Requirements (Functional + Non-functional)

1.1 Functional Requirements (FR)

These are **"what the system must do"**.

Example A: Mini-Twitter

Top 3 functional requirements:

1. Users should be able to **post tweets**.
2. Users should be able to **follow / unfollow** other users.
3. Users should be able to **see a feed** of tweets from users they follow.

You say this in the interview like:

> "Let me clarify the core user flows:
> – Post a tweet
> – Follow someone
> – See my home feed from people I follow.
> I'll focus mainly on these three."

Example B: E-commerce

Imagine an Amazon-lite system.

Top 3 functional requirements:

1. Users should be able to **browse/search products**.
2. Users should be able to **add/remove items in cart**.
3. Users should be able to **place an order and pay**.

You could add more (reviews, wish-list, etc.) but in interview you usually say:

> "The real system has many more features, but I'll focus on search/browse, cart, and checkout since they're core."

1.2 Non-functional Requirements (NFR)

These are **"how the system should behave"** (qualities).

Example A: Mini-Twitter

For Twitter-like:

1. **High availability** over strong consistency for feed reads
   – It's okay if a new tweet appears with a small delay.
2. **Scale** to 100M+ daily active users.
3. **Low latency** for:
   – posting tweet (< 200ms)
   – loading feed (< 200ms)

You might say:

> “I’ll prioritize availability over consistency, and aim for sub-200ms feed reads at 100M+ DAU.”

Example B: E-commerce

For an e-commerce site:

1. **High availability for browsing & checkout**
   – If site is down during sale, huge loss.

2. **Strong durability & consistency for payments/orders**
   – An order should never be charged twice or disappear.

3. **Elastic scalability**
   – Handle big spikes (Diwali sale, Black Friday).

In interview:

> “For orders and payments, I’ll favor strong consistency and durability. For catalog browsing, I can tolerate eventual consistency for better availability and performance.”

---

2. Core Entities

Here you just list the **main nouns / objects** in the system.
Not full schema, only the big pieces.

2.1 Mini-Twitter Example

From the FRs:

- `User`
- `Tweet`
- `Follow` (a relation: who follows whom)

Later, during design, you might add entities like `Feed`, `Like`, `Notification`, etc.

2.2 E-commerce Example

From our FRs:

- `User`
- `Product`
- `Cart` (or `CartItem`)
- `Order`
- `Payment`

You can say:

> "I'll start with User, Product, Cart, Order, and Payment as my core entities. I can refine fields later when we design the DB."

---

3. API / System Interface

Now you define the **contract**: what endpoints clients call.
Usually **REST** with plural resources.

3.1 Mini-Twitter (from your notes)

Using REST:

```
POST /v1/tweets
body: { "text": "hello world" }

GET /v1/tweets/{tweetId}

POST /v1/follows
body: { "followee_id": "123" }

GET /v1/feed
```

Important points:

- **Plural resource names**: `/tweets`, `/follows`.
- **Current user** is taken from **auth token** (e.g., JWT in headers), not from body.
    - Don't trust `userId` sent in request body.

3.2 E-commerce Example

A simple set of APIs:

```
# Product browsing
GET /v1/products?query=iphone&page=1

GET /v1/products/{productId}

# Cart
GET  /v1/cart              -> current user's cart
POST /v1/cart/items
body: { "product_id": "p123", "quantity": 2 }

DELETE /v1/cart/items/{itemId}

# Order + payment
POST /v1/orders
body: { "address_id": "a10", "payment_method_id": "pm5" }

GET /v1/orders/{orderId}
```

Again, **user identity** is from the auth token; the APIs assume "current user".

---

4. [Optional] Data Flow

This is a simple **step-by-step story** of what happens for a key operation.

4.1 Web Crawler example (from your text)

- Fetch seed URLs
- Parse HTML
- Extract URLs
- Store data
- Repeat

4.2 E-commerce Example: “Place Order” Flow

For one important flow:

1. User clicks **“Place Order”** in the UI.
2. Client sends `POST /v1/orders`.
3. API gateway routes to **Order Service**.
4. Order Service:
    - fetches **cart items** from Cart Service / DB
    - calculates **total price**
    - calls **Payment Service** to charge the card
5. On success:
    - Order Service writes **Order** record to DB
    - clears cart
    - sends message to **Inventory Service** to reduce stock
6. Returns `orderId` to user.

You’ll use this story to decide what services, DBs, and queues you need in HLD.

---

5. High Level Design (HLD) & The Diagram

Now let's decode the **Twitter HLD diagram** you shared.
It has 4 quadrants, each for one endpoint:

1. `POST /v1/tweet`
2. `GET /v1/tweet/:tweetId`
3. `POST /v1/follows`
4. `GET /v1/feed`

Common Components in the Diagram

- **Client**: mobile app / web browser.
- **API Gateway**: single entry point that forwards API requests to internal services.
- **Tweet CRUD Service**: business logic for creating/reading tweets.
- **Follower Service**: manages follows (user A follows user B).
- **Feed Service**: builds the home feed.
- **Tweet DB (Postgres)**: stores tweets.
- **Following DB (Postgres)**: stores `Follow` relationships (`userId1`, `userId2`).

Now each quadrant.

5.1 Top-left: `POST /v1/tweet` (Create Tweet)

Flow:

1. Client calls `POST /v1/tweets` with `{"text": "hello"}`.
2. Request goes into **API Gateway**.

3. Gateway forwards to **Tweet CRUD Service**.
4. Tweet CRUD Service:
    - Finds current user from auth token.
    - Validates text, maybe extracts media links.
    - Writes a record into **Tweet DB**:
        - `id`
        - `authorId`
        - `text`
        - `media: s3Links[]` (links to S3 or any object storage).
5. Service returns success to API Gateway → Client.

This satisfies: **"Users should be able to post tweets."**

5.2 Top-right: `GET /v1/tweet/:tweetId` (Read Single Tweet)

Flow:

1. Client calls `GET /v1/tweets/{tweetId}`.
2. API Gateway → **Tweet CRUD Service**.
3. Service reads the tweet from **Tweet DB** by `tweetId`.
4. Returns tweet details to client.

Satisfies: **"Users should be able to see tweets."**

5.3 Bottom-left: `POST /v1/follows` (Follow Someone)

Flow:

1. Client calls `POST /v1/follows` with `{ "followee_id": "B" }`.
2. API Gateway → **Follower Service**.
3. Follower Service:
    - Gets current user from token (say user A).
    - Validates that both users exist.
    - Writes a row in **Following DB**:
        - `userId1` = A (follower)
        - `userId2` = B (followee)
4. Returns success.

This satisfies: **"Users should be able to follow other users."**

5.4 Bottom-right: `GET /v1/feed` (Get Home Feed)

This is the interesting part – multiple services talk to each other.

Flow:

1. Client calls `GET /v1/feed`.
2. API Gateway → **Feed Service**.
3. Feed Service:
    - Calls **Follower Service** (via an internal API like `/v1/followers`):
        - `getFollowers()` or better `getFollowees(currentUser)` (naming in diagram is a bit loose).

    - Follower Service reads **Following DB** to get list of users that current user follows.

4. With this list of followees, Feed Service calls **Tweet CRUD Service**:
    - `getTweets(followers)` – fetch latest N tweets from those users.
    - Tweet CRUD Service reads **Tweet DB**.
5. Feed Service may:
    - merge, rank, sort tweets (e.g., by time)
    - maybe cache results
6. Feed Service returns **array of tweets** to client.

This satisfies: **"Users should be able to see tweets from users they follow (feed)."**

Notice how this HLD:

- maps **directly** from the APIs,
- uses the **core entities** (`Tweet`, `Follow`),
- and supports the **functional requirements**.

---

6. Mapping This Style to E-commerce HLD

If you draw a similar HLD for e-commerce, it might look like:

Example: `POST /v1/orders` (Place Order)

- Client
  → **API Gateway**
  → **Order Service**
- Order Service:

    - reads user's **Cart** from Cart DB or Cart Service
    - calls **Payment Service** (maybe over gRPC)
    - on success writes **Order** to Order DB
    - sends a message to **Inventory Service** via **Message Queue** (Kafka/RabbitMQ) to reduce stock
- Order DB stores:
    - `orderId`, `userId`, `orderItems[]`, `totalAmount`, `status`, timestamps etc.

If you drew this as boxes and arrows, it would look very similar to the Twitter diagram: client → gateway → services → DBs, plus possibly queues.

How to Use All This in an Interview

1. **Start**: Clarify **functional requirements** (top 3) and then **non-functional** (3–5).
2. **List core entities**: just the big nouns.
3. **Define APIs**: REST, plural resource names, user from auth token.
4. **(Optional) Data flow** for one or two key operations.
5. **Draw HLD**:
    - client → API gateway → services → DBs/caches/queues
    - walk through each API endpoint and show how request flows and which state changes.

🌟 What is the Deep Dives Section? (~10 minutes)

After you finish:

- Requirements
- Core entities
- API
- High-level design diagram

You now move to a **deep analysis** where you:

🔥 1. Strengthen your design

Ensure the system meets all non-functional requirements (scale, performance, latency, availability).

🔥 2. Explore bottlenecks

Identify weak points and propose improvements.

🔥 3. Solve hard problems

Talk about caching, sharding, replication, concurrency, queues, etc.

🔥 4. Discuss tradeoffs

Especially important for senior engineers:

- availability vs consistency
- latency vs cost
- freshness vs fanout cost

🔥 5. Iterate on your design with interviewer feedback

This is a *conversation*, not a monologue.

🧠 Why Interviewers Care About Deep Dives

Because the high-level design is **easy** — most candidates can do it.

Deep dives test your:

- **ability to diagnose performance bottlenecks**
- **knowledge of distributed systems**
- **skill in scaling beyond millions of users**
- **handling of NFRs (latency, availability, durability)**
- **communication style**
- **tradeoff decision making**

This determines if you are **L4, L5, L6** or beyond.

🔍 Example Deep Dive for Twitter

(From the note you shared)

Your high-level design for fetching feed was:

```
Client → Feed Service → Follower Service → Tweet CRUD → DB
```

But this design becomes **too slow** for large scale.

Suppose a user follows 3000 accounts.
Feed Service has to:

1. Fetch list of followees
2. Fetch latest tweets of those 3000 users

3. Sort & merge
4. Return feed

This is **fanout-on-read**, and it is expensive.

⭐ Deep Dive Topic 1: Feed Scalability (Fanout)

Because NFR says:

> "Feed should load with low latency."
> "System must scale to 100M+ DAU."

Now you proactively say:

🟦 Option A — Fanout-on-read (current design)

Pros:

- Writes (tweet creation) are cheap
- No overhead at write time

Cons:

- Reads become extremely expensive for high-follower-count users
- Every feed load becomes slow
- Hot users (celebrities) cause massive read amplification

Thus, fanout-on-read becomes a bottleneck.

🟦 Option B — Fanout-on-write (what Twitter actually uses)

When a tweet is created:

- Push tweet ID into followers' individual timeline caches.
- Most users have few followers → cheap.
- For users with millions of followers:
    - Put their tweets into a "heavy user" list (stored separately)
    - Followers pull tweets from this list as needed.

Pros:

- Very fast feed reads (< 100ms)
- Easy to cache per-user feeds

Cons:

- Write path becomes more expensive
- Requires background workers and queues
- Precision tuning for celebrity users

This is the **classic Twitter deep dive**.

🟦 Add caching

You say:

> "I'll store each user's feed in Redis, updated on write. Feed reads will hit Redis, falling back to DB if needed."

Caches greatly reduce latency.

🟦 Shard the Tweet DB

Once number of tweets becomes huge:

- Vertical scaling fails (Postgres too big)
- Horizontal sharding is needed

Shard by:

- tweetId range
- authorId
- hash(authorId)

🟦 Handle high availability

Add:

- replicated DB nodes
- multi-region read replicas
- queue retry logic
- circuit breakers

This ensures uptime even under failures.

🔍 Example Deep Dive for E-commerce (Amazon-like)

Suppose requirements include:

- strong consistency for inventory
- low latency browsing
- high availability during sale

You lead a deep dive like this:

⭐ Topic 1: Inventory Consistency

Problem:
Two users buy the last remaining item at the same time.

Approach:

- Use **inventory reservation**
- Use **atomic DB updates** or **distributed locks**
- Or use **queues** to serialize inventory updates
- Or use **optimistic locking** with version numbers

Tell the interviewer the tradeoff:

- Strong consistency → lower availability
- Eventual consistency → faster but risk overselling

⭐ Topic 2: Scaling Search

Product search is expensive.

Solutions:

- Introduce **Elasticsearch** for full-text search.
- Use **caching** for popular queries.

- Use **async indexing** workers.

⭐ Topic 3: Handling High Traffic (Sales events)

Introduce:

- Load balancers
- CDN for images
- Auto-scaling groups
- Cache product pages
- Rate limiting

🟦 What Senior Candidates Do During Deep Dives

✔ Identify bottlenecks *themselves*

Example:

> “My current feed approach will not scale for users who follow thousands of accounts. I want to redesign using fanout-on-write.”

✔ Raise availability + consistency tradeoffs

Example:

> “Inventory updates must be strongly consistent, while browsing can tolerate eventual consistency.”

✔ Introduce technologies step-by-step

Queues, caches, sharding, CDN, load balancers, message brokers.

✔ Explain tradeoffs

Not just naming technologies — explaining **why**.

✔ Let interviewer interrupt / guide

You don’t monologue nonstop.

🔥 What Mistakes Candidates Make

❌ Talking nonstop

They never let interviewer ask questions.

❌ Adding premature complexity

They use Kafka, Redis, Elastic, Sharding… too early, before even identifying if needed.

❌ Panicking when interviewer challenges design

Challenges are a GOOD sign — they are testing depth.

❌ Not tying decisions back to NFRs

Every solution must map to:

- scalability
- latency
- availability
- durability

🎯 How YOU Should Conduct the Deep Dive (Script)

Here's exactly what you say:

Step 1: Identify 1–2 biggest bottlenecks

> "Given our scale and latency requirements, the biggest bottleneck will be generating feeds on read. Let's deep dive into optimizing feed delivery."

Step 2: Propose improved design

> "I'll introduce fanout-on-write with Redis caching to precompute feeds."

Step 3: Discuss tradeoffs

> "Write path becomes heavier, but read latency improves drastically which matches our NFR: fast feeds and high read traffic."

Step 4: Stop talking and wait

Interviewer will now probe deeper.