Introduction to HLD

1.HLD (High-Level Design)

What It Means

It represents the overall architecture of a system — a bird's-eye view of how the complete application is organized and how its major parts connect with each other.
Think of it as the **blueprint** prepared before construction begins.

What It Focuses On

- Main components or services of the system
- How these components interact
- Communication patterns (REST APIs, message queues, etc.)
- Choice of databases and storage
- Use of load balancers, cache layers, CDN, etc.
- Overall flow of data through the system

HLD avoids low-level code details and instead emphasizes the structure and behavior of the entire system.

Example

Imagine you're designing a food-delivery application like **Swiggy**.

Your **High-Level Design** might include:

- **User Service**
- **Restaurant Service**
- **Order Service**
- **Delivery Partner Service**
- **Notification Service**
- **Databases** (SQL for orders, NoSQL for menus, Redis for caching)
- **API Gateway** for routing
- **Load Balancer** to distribute traffic
- **CDN** for images of food items
- **Auth Service** for login and security

HLD is like a **map of the entire city** that shows roads, areas, and how locations are connected.
It does not go into room-by-room details — that part belongs to **LLD (Low-Level Design)**.

2.LLD (Low-Level Design)

What It Means

LLD describes how each component of the system will work internally.
It moves closer to the implementation level — almost like preparing for coding.

What It Focuses On

- Classes, objects, methods, and attributes
- How classes relate to each other (inheritance, composition, association)
- Detailed flow of functions and algorithms
- Precise database table structures, fields, and constraints
- Validation rules, error handling, data flow inside modules

This is where developers decide *exactly how things will be built*.

Example (Continuing the Swiggy Case)

For the **Order Service**, the LLD might include:

Class: Order

- Attributes:
  - orderId
  - userId
  - restaurantId
  - items
  - amount
  - status
  - timestamp
- Methods:
  - placeOrder()
  - calculateTotal()
  - cancelOrder()
  - updateStatus()

Database Table: orders

Column	Type	Description
order_id	INT (PK)	Unique order ID
user_id	INT (FK)	User who placed the order
restaurant_id	INT (FK)	Restaurant selected
amount	DECIMAL	Total order amount
status	VARCHAR	Order state
created_at	TIMESTAMP	Time of order creation

It also includes:

- Flow of placing/cancelling an order
- How status updates propagate to delivery and notification services
- Validation rules like checking restaurant availability

Simple Analogy

LLD is like the **detailed plan of a single building** —
It shows the exact placement of every room, pipe, wire, and door.

HLD is the map of the entire city,
LLD is the construction plan of one specific building.

3.Client–Server Model

Meaning

It is a communication model where **one side requests** (client) and the **other side provides a service** (server).
For example: when you open a website, your browser (client) asks for the page, and the web server sends it back.

How It Works

1. **Client sends a request**
   (e.g., browser asking for a webpage)

2. **Server receives the request**
   (e.g., server checks data, processes logic)
3. **Server sends a response**
   (e.g., server sends the webpage data)
4. **Client displays the result**
   (browser shows you the website)

This cycle repeats whenever the client needs something.

Real-Life Example

Think of a **restaurant**:

- **You (Client)** → ask for food (request)
- **Waiter + Kitchen (Server)** → prepare food (process)
- **Food served to you (response)**

You request… they respond. That's the client–server model.

Advantages

- Centralized control
- Easy to maintain and update
- Better security (server controls the data)

Disadvantages

- If server crashes → all clients get affected
- Server can become overloaded if too many clients request at the same time

4. Scaling in System Design

When your application grows (more users, more traffic), you need to **scale** your system so it can handle the load.

There are **two classic ways** to scale:

- **Vertical Scaling (Scale Up)**
- **Horizontal Scaling (Scale Out)**

Both have their importance, just like old systems and modern cloud systems.

1. Vertical Scaling (Scale Up)

Meaning

Increase the **power** of the existing server.

You keep **one machine**, but make it stronger.

How?

- Add more RAM
- Add more CPU cores
- Use a faster SSD
- Switch to a bigger EC2 instance (like t2.small → t2.large)

Example

Imagine you have one shop counter.
When more customers come, you:

- Replace the worker with a faster one
- Give them better tools
- Add more billing machines at the *same* counter

Still only one counter, but more powerful.

Advantages

✓ Easy to implement
✓ No change in code or architecture
✓ Good performance for small to medium scale
✓ Less complexity

Disadvantages

✗ Machine has a **limit** (you can't upgrade forever)
✗ Expensive to upgrade high-end hardware
✗ Single point of failure — if this server goes down, everything stops
✗ Not suitable for large-scale apps

Perfect for: small apps, early-stage startups, simple backend.

2. Horizontal Scaling (Scale Out)

Meaning

Increase the **number of servers**.

Instead of making one machine powerful, you use **many machines**, working together.

How?

- Add multiple servers

- Add load balancer
- Split traffic among them
- Add more machines when traffic increases

Example

Instead of one shop counter, you open **10 counters**.
Customers automatically go to available counters.

This is modern distributed system scaling.

Advantages

✓ No limit — you can keep adding machines
✓ Supports huge traffic
✓ If one machine fails → others handle the load
✓ More reliable and fault-tolerant
✓ Used by big companies (Google, Amazon, Netflix)

Disadvantages

✗ More complex
✗ Needs load balancers
✗ Data consistency issues (multiple DB replicas, caching, etc.)
✗ Requires distributed system design

Perfect for: production systems, large-scale apps, high-traffic platforms.

5. Where Networking Fits in HLD

Whenever two systems communicate (client → server, service → service), the communication follows the **OSI Model** (7 layers):

1. Application
2. Presentation
3. Session
4. Transport
5. Network
6. Data Link
7. Physical

But for HLD, the important layers are:

- **Application Layer**
- **Transport Layer**

6. Why HLD Focuses on Application Layer & Transport Layer

A. Application Layer → "WHAT is being communicated?"

This defines the *logical communication* between services.

HLD decides:

- **Which protocol to use (HTTP, HTTPS, WebSocket, gRPC, SMTP, FTP)**
- **API structure (REST, GraphQL, RPC)**
- **Request/response format (JSON, XML)**
- **Client–server interactions**
- **Authentication and authorization flow**

Example

In a Swiggy-like system:

User Service → Restaurant Service
Communication: HTTP/HTTPS REST API
Data format: JSON

HLD specifies:
"Services communicate over HTTPS for security."

This is pure Application Layer.

B. Transport Layer → "HOW is it delivered?"

Once the application decides what to send, the transport layer decides how to deliver it reliably.

It deals with:

- **TCP (reliable, ordered)**
- **UDP (fast, real-time)**
- **Ports (HTTP → 80, HTTPS → 443)**
- **Connection type**

Example

- **APIs use TCP for reliable communication**
- **Real-time chat or live tracking may use UDP**

So HLD mentions which transport protocol your system uses and why.

7. Why Not Lower Layers (Network, Data Link, Physical)?

Because these layers deal with:

- **Routing**
- **Switching**
- **MAC addressing**
- **Hardware**
- **Cables & signals**

These are handled by infrastructure teams and cloud providers (AWS, GCP, Azure).

HLD only focuses on logical communication, not hardware-level design.

8.DNS in HLD

What is DNS?

DNS (Domain Name System) is the Internet's phonebook.

It converts a domain name like

www.swiggy.com

into an IP address like

13.234.56.10

Computers communicate using IPs, so DNS helps your client locate the server.

DNS in High-Level Design

In HLD, DNS explains how clients reach your system.

Flow:

1. **User enters www.swiggy.com**
2. **DNS resolves it to an IP**
3. **Request goes to load balancer**
4. **Load balancer routes it to backend servers**

HLD point:
"DNS directs the client to the correct IP address before the request reaches the application."