System Design Guide

Designing a system is like building a skyscraper: you need a blueprint for the overall structure (HLD) and specific schematics for the plumbing and electrical wiring (LLD).

This guide breaks down the core pillars of System Design to help you understand how to build scalable, reliable, and maintainable software.

1. High-Level Design (HLD) vs. Low-Level Design (LLD)

Before diving into the concepts, it is important to distinguish between the two layers of design.

Feature	**High-Level Design (HLD)**	**Low-Level Design (LLD)**
Focus	System Architecture & Components	Module Logic & Class Design
Target Audience	Stakeholders, Architects, Leads	Developers, Implementers
Components	Databases, Load Balancers, Microservices	Classes, Interfaces, Design Patterns
Goal	Scalability, Availability, Reliability	Code Reusability, Maintainability

2. High-Level Design (HLD) Core Concepts

HLD focuses on the "macro" view—how different services interact and how data flows through the system.

Key Architectural Patterns

- **Monolithic:** A single, unified unit. Simple to deploy but hard to scale.
- **Microservices:** Decoupling features into independent services. High scalability but increased complexity.
- **Serverless:** Running code without managing servers (e.g., AWS Lambda).
- **Event-Driven:** Systems that react to state changes (events) using message brokers.

Scalability and Performance

- **Vertical Scaling:** Adding more power (CPU/RAM) to an existing server.
- **Horizontal Scaling:** Adding more servers to the pool.
- **Load Balancing:** Distributing incoming traffic across multiple servers to prevent any single server from becoming a bottleneck.

Databases and Storage

- **SQL (Relational):** Structured data (e.g., PostgreSQL, MySQL). Best for ACID compliance.
- **NoSQL (Non-Relational):** Unstructured data (e.g., MongoDB, Cassandra). Best for high-volume, flexible schemas.
- **Database Sharding:** Horizontal partitioning of data across multiple databases.
- **Consistent Hashing**: A strategy used to minimize data reorganization when nodes (database shards or cache servers) are added or removed from a cluster. It ensures that only a small fraction of keys need to be remapped

during scaling. Crucial for distributed caching (e.g., Redis clusters) to prevent a 'cache storm' when servers are added/removed. It is used to map both **data (keys)** and **nodes** onto the same logical ring.

- **Replication:** Copying data across servers to ensure high availability.

Caching

Reducing latency by storing frequently accessed data in memory (e.g., Redis, Memcached).

- **Write-through cache:** Data is written to the cache and the DB simultaneously.
- **Cache-aside:** The application checks the cache first; if it's a "miss," it queries the DB and updates the cache.

---

3. Low-Level Design (LLD) Core Concepts

LLD is about the internal logic of a component. It ensures the code is clean, modular, and easy to extend.

SOLID Principles

These are the gold standard for object-oriented design:

1. **S - Single Responsibility:** A class should have one, and only one, reason to change.
2. **O - Open/Closed:** Software entities should be open for extension but closed for modification.
3. **L - Liskov Substitution:** Subtypes must be substitutable for their base types.
4. **I - Interface Segregation:** Don't force a class to implement interfaces it doesn't use.
5. **D - Dependency Inversion:** Depend on abstractions, not concretions.

Design Patterns

Common solutions to recurring software problems:

- **Creational:** Singleton, Factory, Builder, Prototype.
- **Structural:** Adapter, Decorator, Facade, Proxy.
- **Behavioural:** Observer, Strategy, Command, State.
- **Stability Pattern:** Circuit Breaker.

Data Modelling (LLD Perspective)

- **Class Diagrams:** Visualizing the attributes and methods of classes and their relationships (Inheritance, Composition, Aggregation).
- **Sequence Diagrams:** Mapping out how objects interact with each other over time for a specific use case.

---

4. Cross-Cutting Concerns (The "Vital" List)

These concepts apply to both HLD and LLD and are essential for any robust system.

- **CAP Theorem:** In a distributed system, you can only provide two of the following three guarantees: Consistency, Availability, and Partition Tolerance.
- **PACELC Theorem**: This is an extension of CAP that addresses how a system behaves when there is no partition (the "E" for "else," choosing between Latency and Consistency). PACELC highlights that even when the system is running normally (no partition), there is a trade-off between Latency and Consistency.
- **Consistency Models:** Understanding Strong vs. Eventual Consistency (e.g., how long it takes for a profile picture update to show for all users).
- **Rate Limiting:** Protecting your APIs from being overwhelmed by too many requests.
- **Security Fundamentals:** Security - Implementing **Authentication** (identity verification) vs. **Authorization** (permission control), and ensuring **Encryption** for data at rest and in transit.
- **Message Queues:** Using tools like Kafka or RabbitMQ for asynchronous processing and decoupling services.
- **Saga Pattern:** A way to manage distributed transactions across multiple microservices by using a sequence of local transactions and compensating transactions to maintain consistency.
- **Monitoring and Logging:** Implementing observability (Prometheus, ELK stack) to catch issues before users do.

- **The Four Golden Signals:** Latency, Traffic, Errors, and Saturation. Latency (time per request), Traffic (demand), Errors (rate of failure), and Saturation (how 'full' the service is). **Saturation** is often a leading indicator of failure, as latency typically spikes exponentially once a resource hits 100% utilization.
- **SLIs, SLOs, and SLAs:** Service Level Indicators, Objectives, and Agreements—how you define "up-time" and performance targets.
- **Observability & Tracing:** Distributed Tracing is essential for identifying which specific service in a chain is causing a bottleneck. Distributed Tracing Using tools like Jaeger or Zipkin to track the path of a single request as it moves across multiple microservices to identify latency bottlenecks.
- **Circuit Breaker Pattern:** A structural/behavioural pattern that prevents a failing service from causing a cascading failure across the entire system. This pattern prevents cascading failures by failing fast when a downstream service is known to be down.
- **Security Fundamentals**
  - **Authentication vs. Authorization:** The difference between *who* someone is (OAuth2, OIDC) and *what* they are allowed to do (RBAC, ABAC).
  - **Encryption at Rest and in Transit:** Using TLS/SSL for data moving through the system and AES-256 for data stored in databases.
  - **Secret Management:** How to store API keys and DB passwords securely (e.g., HashiCorp Vault, AWS Secrets Manager) instead of hardcoding them. Always prefer dynamic secrets (short-lived credentials) over static API keys to minimize the blast radius of a leak.

---

5. Deployment & Infrastructure

Modern system design is often "DevOps-aware":

- **Blue-Green & Canary Deployments:** Strategies for rolling out updates without downtime.
- **Infrastructure as Code (IaC):** Using tools like Terraform or CloudFormation to define your HLD architecture in code

---

How to Approach a System Design Interview

If you are preparing for a technical interview, follow this framework:

1. **Requirement Clarification:** Functional (what it does) vs. Non-functional (latency, scale).
2. **Back-of-the-envelope estimation:** Calculate throughput (QPS) and storage requirements.
3. **API Design:** Define the main endpoints.
4. **Database Schema:** Choose the right DB and define tables/collections.
5. **High-Level Design:** Draw the core components and their connections.
6. **Deep Dive:** Focus on specific challenges (e.g., "How do we handle million-user surges?").

Pro Tip: Always prioritize **Scalability** and **Reliability**. In the real world, a system that works perfectly but can't handle growth is a failure.

High Level Design (HLD)

*To deep dive into **High-Level Design (HLD)**, we need to move beyond just listing components and focus on **Architectural Patterns, Data Flow, and System Trade-offs**. HLD is about how you organize the "macro" components of your system to meet specific non-functional requirements (Availability, Scalability, Latency).*

1. Architectural Patterns: Choosing the Skeleton

The architecture you choose determines how easy it is to scale and deploy your system.

Microservices vs. Monolith

- **Monolithic:** All components (Auth, Payments, Catalog) share a single codebase and database.
  - *Best for:* Small teams, MVP stage.
- **Microservices:** Each service is a separate entity with its own database. They communicate via APIs or Message Brokers.
  - *Best for:* Large-scale systems where teams need to deploy independently.

Service-Oriented Architecture (SOA) vs. Microservices

While similar, SOA usually involves an **Enterprise Service Bus (ESB)** for communication, whereas Microservices prefer "dumb pipes" (like Kafka or RabbitMQ) and "smart endpoints" (the services themselves).

2. Load Balancing Strategies

A Load Balancer (LB) is the entry point of your system. It prevents any single server from being overwhelmed.

- **Layer 4 (L4) Load Balancing:** Operates at the Transport level (TCP/UDP). It's fast because it doesn't look at the content of the packets—it just routes based on IP and Port.
- **Layer 7 (L7) Load Balancing:** Operates at the Application level (HTTP/HTTPS). It is "smarter"—it can route traffic based on the URL path (e.g., /images goes to the Image Server, /api goes to the App Server).

3. The CAP Theorem & Distributed Databases

In HLD, you must decide how your system behaves during a network failure (a "partition"). You can only pick two:

1. **Consistency (C):** Every read receives the most recent write.
2. **Availability (A):** Every request receives a response (even if it's not the most recent data).
3. **Partition Tolerance (P):** The system continues to operate despite network failures.

Note: Since network failures are inevitable in distributed systems, we effectively choose between **CP** (Consistency and Partition Tolerance) or **AP** (Availability and Partition Tolerance).

4. Scalability: Horizontal vs. Vertical

- **Vertical (Scaling Up):** Upgrading your server (e.g., moving from 16GB to 64GB RAM). There are a hard hardware limit and a single point of failure. Primarily used for systems that are difficult to distribute, such as legacy relational database master nodes, as it hits hardware ceilings quickly.
- **Horizontal (Scaling Out):** Adding more machines. This is the heart of HLD. It requires a **stateless** application tier so that any request can be handled by any server.

5. Caching and Content Delivery (CDN)

To reduce latency, you want to move data as close to the user as possible.

- **CDN (Content Delivery Network):** Stores static assets (images, JS, CSS) on edge servers globally (e.g., Cloudflare, Akamai).
- **Application Cache:** Using Redis or Memcached to store computed results or DB queries.
  - **Cache Invalidation:** One of the "two hard things in computer science." Strategies include **TTL (Time to Live)**, **LRU (Least Recently Used)**, or **Write-around vs. Write-back vs. Write-through.** Use Write-around to avoid flooding the cache with data that is written but rarely read, preserving space for frequently accessed keys.

---

6. Communication Protocols

How do your services talk to each other?

- **REST:** Standard, human-readable (JSON). Good for public APIs.
- **gRPC:** Built on HTTP/2, uses Protocol Buffers. Much faster and smaller than REST. Great for internal microservice communication.
- **WebSockets:** For real-time, bi-directional communication (e.g., Chat apps).
- **Webhooks:** For "push" notifications from one system to another.

Service Discovery

The mechanism that allows microservices to find each other dynamically without hardcoded IP addresses.
- In a microservices architecture, you cannot rely on static IP addresses because services scale horizontally, fail, or move to different nodes. Service Discovery acts as a central registry that allows services to find and communicate with each other dynamically.

Key Components

- **Service Registry**: A database containing the network locations (IP and Port) of all active service instances (e.g., Netflix Eureka, Consul, or etcd).
- **Registration**: When a service starts, it registers its location with the registry. It typically sends periodic "heartbeats" to prove it is still healthy.
- **Discovery (The "Lookup")**: When Service A needs to talk to Service B, it queries the registry to get the current list of available IP addresses for Service B.

Discovery Patterns

- **Client-Side Discovery**: The client (Service A) is responsible for querying the registry and using a load-balancing algorithm to pick an instance of Service B.
- **Server-Side Discovery**: The client sends a request to a **Load Balancer**, which then queries the registry and routes the request to an available instance.

Why it matters

- **Eliminates Hardcoding**: You no longer need to manage configuration files with static IPs for every environment.
- **Enables Auto-scaling**: As new instances of a service are spun up to handle high traffic, they are automatically added to the registry and become available for requests.
- **Improves Fault Tolerance**: If a service instance crashes, the registry marks it as unhealthy after a missed heartbeat, ensuring no more traffic is routed to that failed instance.

---

7. Message Queues and Asynchrony

Not every task needs to happen immediately.

- **Synchronous:** User waits for the task to finish (e.g., logging in).
- **Asynchronous:** Task is offloaded to a queue (e.g., generating a PDF report, sending an email).
- **Tools:** Apache Kafka (high throughput), RabbitMQ (complex routing), Amazon SQS.

Back-Pressure

When using asynchronous message queues, a "Fast Producer" may send data at a rate higher than a "Slow Consumer" can process.

- **Definition: Back-Pressure** is a feedback mechanism where the consumer signals the producer to slow down or buffer data.
- **Why it Matters:** Without back-pressure, the system risks stability issues, such as memory overflows or message queue crashes due to exhaustion of resources.
- **Implementation:** Producers can be designed to drop messages, buffer them temporarily, or wait for an "ACK" (acknowledgment) from the consumer before sending the next batch.

Summary Checklist for HLD

When you are designing the HLD for a system (like a "Video Streaming App" or "E-commerce Site"), always ask yourself:

1. **Where is the bottleneck?** (Is it DB writes? Is it high traffic?)
2. **How do I handle failure?** (If Service A goes down, does the whole system break?)
3. **Is it stateless?** (Can I add 10 more servers easily?)
4. **Is the data consistent?** (Do all users see the same count of 'likes'?)

Low Level Design (LLD)

While HLD is about the "boxes and arrows" of a system, Low-Level Design (LLD) is about the "nuts and bolts" of the code. It focuses on the internal logic, class hierarchies, and how modules interact to ensure the code is readable, maintainable, and extensible.

As a Software Engineer, you might know that bad LLD leads to "spaghetti code" that is impossible to refactor. Here is a deep dive into the core pillars of LLD.

---

1. The SOLID Principles

These five principles are the foundation of clean Object-Oriented Design (OOD).

- **S: Single Responsibility (SRP):** A class should do one thing. If a User class is handling both database persistence and email notifications, it's violating SRP.
- **O: Open/Closed (OCP):** You should be able to add new functionality (extension) without changing existing code (modification). Think of using **Interfaces** or **Abstract Classes**.
- **L: Liskov Substitution (LSP):** Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.
- **I: Interface Segregation (ISP):** Better to have many specific interfaces than one "fat" interface. Don't force a class to implement methods it doesn't need.
- **D: Dependency Inversion (DIP):** High-level modules should not depend on low-level modules; both should depend on abstractions (e.g., depend on a MessageService interface, not a TwilioSMSService implementation).

---

2. Object-Oriented Design Patterns

1. Creational Patterns

These patterns deal with **object creation mechanisms**, trying to create objects in a manner suitable to the situation.

- **Singleton:** Ensures a class has only one instance and provides a global point of access to it.
- **Factory Method:** Defines an interface for creating an object but let's subclasses decide which class to instantiate.
- **Abstract Factory:** Lets you produce families of related objects without specifying their concrete classes.
- **Builder:** Lets you construct complex objects step-by-step. It allows you to produce different types and representations of an object using the same construction code.
- **Prototype:** Lets you copy existing objects without making your code dependent on their classes.

2. Structural Patterns

These patterns explain how to **assemble objects and classes** into larger structures while keeping these structures flexible and efficient.

- **Adapter:** Allows objects with incompatible interfaces to collaborate.
- **Bridge:** Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently.
- **Composite:** Lets you compose objects into tree structures and then work with these structures as if they were individual objects.
- **Decorator:** Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
- **Facade:** Provides a simplified interface to a library, a framework, or any other complex set of classes.
- **Flyweight:** Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

- **Proxy:** Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

3. Stability Pattern.

Circuit Breaker: A pattern used to detect failures and encapsulate the logic of preventing a failure from constantly recurring during maintenance or temporary external outages. Typically implemented using a three-state machine: **Closed** (normal), **Open** (tripped/failing), and **Half-Open** (testing recovery).

4. Behavioural Patterns

These patterns are concerned with **algorithms and the assignment of responsibilities** between objects.

- **Chain of Responsibility:** Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- **Command:** Turns a request into a stand-alone object that contains all information about the request.
- **Iterator:** Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
- **Mediator:** Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
- **Memento:** Lets you save and restore the previous state of an object without revealing the details of its implementation.
- **Observer:** Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- **State:** Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
- **Strategy:** Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- **Template Method:** Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
- **Visitor:** Lets you separate algorithms from the objects on which they operate.

5. Modern/Common Patterns (Non-GoF)

While the GoF patterns are the foundation, modern software development often references these as well:

- **Repository Pattern:** Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.
- **Dependency Injection (DI):** A pattern where an object receives other objects that it depends on, typically handled by a framework (like Spring or Dagger). Frameworks like Spring/Dagger automate **Dependency Injection**, fulfilling the 'D' in SOLID by removing the need for classes to instantiate their own dependencies. Modern frameworks automate this to decouple class instantiation from business logic, making unit testing significantly easier via mocking.
- **Data Transfer Object (DTO):** Used to pass data between software application subsystems or layers.

---

3. Class Diagrams & Relationships

In LLD, you must define how classes are linked. Understanding the strength of these links is vital:

Relationship	**Description**	**Example**
Association	A general link between two classes.	A Teacher and a Student.
Aggregation	A "has-a" relationship where the child can exist without the parent.	A Department and a Professor.

Composition	A strong "has-a" relationship where the child cannot exist without the parent.	A House and its Rooms.
Inheritance	An "is-a" relationship.	A Car is a Vehicle.

4. Database Schema Design (LLD Level)

While HLD decides *which* database to use, LLD decides the *structure* within it:

- **Normalization:** Reducing data redundancy (1NF, 2NF, 3NF).
- **Indexing:** Choosing which columns to index (B-Trees vs. Hash Indexes) to speed up read queries without killing write performance.
- **Concurrency Control:** Handling race conditions using **Pessimistic Locking** (locking the row) or **Optimistic Locking** (using a version/timestamp).

5. Sequence Diagrams: Mapping Interaction

A Sequence Diagram is the most important LLD tool for documenting a specific use case. It shows the objects involved and the chronological order of messages exchanged.

6. Error Handling & Exception Strategy

Good LLD includes a robust strategy for when things go wrong:

- **Global Exception Handling:** Centralized way to catch and log errors.
- **Custom Exceptions:** Using domain-specific errors (e.g., InsufficientFundsException) instead of generic ones.
- **Retry Logic:** Implementing **Exponential Backoff** for external API calls.
- **Idempotency**: Ensuring that an operation (like a payment) can be performed multiple times without changing the result beyond the first attempt. Idempotency is the primary defence against side effects (like double-charging) during **Retry Logic** execution when network timeouts occur. It requires a **unique request ID** (idempotency key) to be stored on the server side to track processed requests.

How to approach an LLD Interview Question

When asked to design a system (e.g., "Design a Parking Lot" or "Design a Movie Ticket Booking System"):

1. **Clarify Requirements:** What are the entities? (Vehicles, Parking Spots, Tickets).
2. **Define Classes:** Start with the core entities.
3. **Establish Relationships:** Is it composition or aggregation?
4. **Apply Design Patterns:** Does this need a Factory or a Strategy?
5. **Define APIs/Methods:** What are the public methods of these classes?
6. **Concurrency:** How do we handle two people trying to book the same seat simultaneously?