SYSTEM DESIGN 1

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

In the world of software engineering, designing a system is a two-step process that starts with a broad overview and progressively drills down into the finer details. These two steps are known as High-Level Design (HLD) and Low-Level Design (LLD).

High-Level Design (HLD)

High-Level Design (HLD) is the architectural blueprint of the system. It provides an abstract, bird's-eye view of the system's overall structure, its core components, the modules involved, and how they interact with each other. The primary focus of HLD is on **what** needs to be built, the selection of appropriate technologies and platforms, and how the system will integrate with any external services or components.

Low-Level Design (LLD)

Low-Level Design (LLD) is the subsequent phase that takes the architectural blueprint from the HLD and breaks it down into specific, implementable parts. It delves into the internal mechanics of individual modules and components, defining everything from the algorithms and data structures to the database schema and the intricate logic that will govern their behavior. The focus of LLD is on **how** the system will be built.

HLD vs. LLD: A Comparative Overview

Aspect	**High-Level Design (HLD)**	**Low-Level Design (LLD)**
Definition	An abstract design that defines the system's architecture.	A detailed design that defines how components are implemented.
Focus	What the system will do and how its components interact.	How the system's components will be built.
Scope	Overall architecture, modules, and their interrelationships.	Internal logic, algorithms, and data structures of modules.

Granularity	Macro-level (the big picture).	Micro-level (the intricate details).
Audience	Stakeholders, architects, and project managers.	Developers and engineers.
Deliverables	Architecture diagrams, module overviews, and interface definitions.	Class diagrams, detailed algorithms, and method-level logic.
Purpose	To ensure alignment on the system's structure and goals.	To provide a detailed blueprint for developers to implement.

Components of a High-Level Design (HLD) Document

A comprehensive HLD document typically includes the following key components:

1. **System Architecture:**
    - **Architectural Style:** Defines the foundational pattern for the system (e.g., Monolithic, Microservices, Event-driven).
    - **Inter-service Communication:** Specifies how different parts of the system will communicate (e.g., REST APIs, gRPC, message queues).
2. **Functional Modules:**
    - A breakdown of the system into its major functional blocks or subsystems.
    - A clear description of the purpose and functionality of each module.
3. **Data Flow and Integrations:**
    - Diagrams like Data Flow Diagrams (DFDs) to visualize how data moves through the system.
    - A description of interactions with external entities such as third-party services or APIs.
    - Details on integration points, including the protocols and communication mechanisms to be used.

4. **Technology Stack:**
    - **Programming Languages:** (e.g., Java, Python, Go)
    - **Frameworks:** (e.g., Spring Boot, Django, Node.js)
    - **Databases:** (e.g., SQL like PostgreSQL, or NoSQL like MongoDB)
    - **Infrastructure:** (e.g., Cloud providers like AWS, GCP, Azure)
    - **Middleware/Message Queues:** (e.g., Kafka, RabbitMQ)
5. **Database Design:**
    - Identification of key entities or tables.
    - An outline of the relationships between these entities (often using ER diagrams).
    - The chosen approach for data storage and retrieval (e.g., relational, NoSQL).
6. **Non-Functional Requirements (NFRs):**
    - **Scalability:** How the system will handle growth in user load and data.
    - **Performance:** Expected response times, latency, and throughput targets.
    - **Reliability:** Uptime requirements and failover mechanisms.
    - **Security:** The high-level approach to authentication, authorization, and data protection.
    - **Maintainability:** Strategies for modularity, code reusability, and system monitoring.
7. **Risk and Mitigation:**
    - Identification of potential risks related to the chosen design.
    - High-level strategies to mitigate these identified risks.

Core System Design Concepts

Scalability

Scalability is a system's capacity to handle an increasing workload without a drop in performance or availability. It is crucial for ensuring a system remains responsive and functional as it grows.

- **Vertical Scaling (Scaling Up):**
    - **What it is:** Adding more resources (e.g., CPU, RAM, storage) to a single machine.
    - **Pros:** Simple to implement.
    - **Cons:** Limited by hardware capacity.
    - **Example:** Upgrading a server from 16GB RAM to 64GB RAM.
- **Horizontal Scaling (Scaling Out):**

    - **What it is:** Adding more machines or instances to distribute the workload.
    - **Pros:** Offers much higher scalability potential.
    - **Cons:** More complex to implement and manage.
    - **Example:** Adding more web servers behind a load balancer.
- **Key Enablers of Scalability:**
    - **Load Balancing:** Distributes incoming traffic across multiple servers (e.g., NGINX, AWS Elastic Load Balancer).
    - **Stateless Architecture:** Designing components that do not store session-specific data, making it easy to scale horizontally. For instance, storing user sessions in a distributed cache like Redis instead of on the server itself.
    - **Database Scalability:**
        - **Archival:** Moving old or infrequently accessed data to cheaper storage.
        - **Sharding:** Partitioning data across multiple databases.
        - **Replication:** Creating read replicas of a database to handle high read traffic.
    - **Caching:** Storing frequently accessed data in-memory to reduce database load and improve retrieval speed (e.g., InMemory, Client-Side, or Distributed Cache).
    - **Auto-Scaling:** Dynamically adjusting the number of resources based on real-time demand (e.g., AWS Auto Scaling).
- **Challenges:**
    - **CAP Theorem:** The trade-off between Consistency, Availability, and Partition Tolerance in distributed systems.
    - **Network Latency:** Increased communication delays in horizontally scaled systems.
    - **Complexity:** Managing and monitoring a large, distributed system is inherently more complex.

Fault Tolerance

Fault tolerance is the ability of a system to continue operating correctly even when some of its components fail. This is essential for building reliable and highly available systems.

- **Key Aspects of Fault Tolerance:**
    - **Redundancy:** Deploying duplicate components so that if one fails, another can take over (e.g., running multiple replicas of a microservice).
    - **Failover Mechanisms:** Automatically switching to a backup component upon failure (e.g., a primary-replica database setup with automatic failover).
    - **Replication:** Creating multiple copies of data or services to ensure availability during failures (e.g., Kafka's partition replicas).
    - **Health Monitoring and Self-Healing:** Continuously monitoring system health

    and automatically restarting or replacing unhealthy components (e.g., Kubernetes restarting failed pods).
  - **Load Balancers:** Detecting failed servers through health checks and redirecting traffic to healthy ones.
  - **Data Backup and Recovery:** Regularly backing up data and having a well-defined recovery process (e.g., periodic database snapshots stored in AWS S3).
  - **Distributed Setup:** Spreading components across multiple physical locations or data centers to avoid a single point of failure (e.g., using a geographically distributed CDN).
- **Challenges:**
  - **Overhead:** Redundancy and replication can increase resource usage and introduce delays.
  - **Complexity:** Implementing fault-tolerant systems adds to the architectural complexity.
  - **Data Consistency:** Maintaining data consistency across replicas in a distributed system can be challenging.

Reliability

Reliability refers to a system's ability to perform its intended functions consistently and without failure over a specified period. It's a cornerstone of user trust.

- **Key Pillars of Reliability:**
  - **Uptime and Availability:**
    - Defining a target uptime (e.g., 99.9% availability, known as a "three-nines" SLA).
    - Implementing strategies like redundancy, failover, and load balancing to achieve high availability.
  - **Fault Tolerance:**
    - Designing the system to handle hardware, software, and network failures gracefully.
    - Employing strategies like data replication, circuit breakers, rate limiting, and retries with exponential backoff.
  - **Data Integrity:**
    - Ensuring the accuracy and consistency of data, especially during failures.
    - This can involve ACID compliance for transactional systems or eventual consistency for distributed systems.
  - **Scalability for Reliability:**
    - Ensuring the system can scale effectively under load to maintain its reliability.
  - **Monitoring and Alerts:**

        - Implementing real-time monitoring of key metrics (latency, error rates, resource utilization) using tools like Prometheus, Grafana, or Datadog.
        - Setting up alerts to be notified of any anomalies.
    - **Backup and Disaster Recovery:**
        - Having robust backup strategies in place.
        - A clear Disaster Recovery Plan (DRP) with defined Recovery Time Objective (RTO) and Recovery Point Objective (RPO).

Testing

In HLD, the focus of testing is on defining a comprehensive strategy that aligns with the system's goals and architecture.

- **Testing Strategy:**

    - A high-level plan that encompasses different testing phases:
        - **Unit Testing:** Testing individual components in isolation.
        - **Integration Testing:** Testing the interaction between different components.
        - **Acceptance Testing:** Validating that the system meets business requirements.
- **Non-Functional Testing:**

    - **Performance Testing:**
        - **Stress Testing:** Evaluating system limits under extreme loads.
        - **Load Testing:** Verifying system behavior under expected user traffic.
    - **Reliability Testing:** Simulating failures to validate fault tolerance (e.g., using Chaos Engineering).
    - **Scalability Testing:** Incrementally adding load to verify that the system scales as expected.
    - **Security Testing:**
        - **Penetration Testing:** Identifying security vulnerabilities.
        - Validating authentication and authorization mechanisms.
- **Other Key Testing Aspects:**

    - **Automated Testing:** Creating a plan for automated testing to improve efficiency and coverage.
    - **End-to-End Testing:** Testing complete user workflows from start to finish.
    - **Test Environment and Tools:** Designing a test environment that mirrors production and utilizing mock services where necessary.
    - **Testing in Production:** Using strategies like A/B testing or canary deployments to test new features with minimal risk.

Key Performance Metrics

Tracking performance metrics is vital for understanding system behavior, identifying bottlenecks, and ensuring business requirements are met.

Category	**Key Metrics**	**Use Case**
Latency	Average Latency, P95/P99 Latency, End-to-End Latency	Crucial for user-facing applications to ensure a good user experience.
Throughput	Requests per Second (RPS), Transactions per Second (TPS)	Helps in understanding system capacity and measuring scalability.
Error Rates	HTTP Error Rate (4xx/5xx), Transaction Failure Rate	High rates indicate system instability, bugs, or dependency issues.
Resource Utilization	CPU Utilization, Memory Utilization, Disk I/O, Network I/O	Helps in identifying resource bottlenecks that can degrade performance.
Scalability	Elasticity, Scaling Latency	Critical for cloud-based systems that need to handle fluctuating loads.
Availability	Uptime Percentage, MTBF, MTTD, MTTR	Ensures the system meets its Service Level Agreements (SLAs).

Queue & Wait	Queue Length, Request Wait Time, Consumption Speed, Publisher Latency	Useful for identifying bottlenecks in asynchronous systems and message queues.
Database	Query Latency, Connection Pool Usage, Transaction Throughput	Ensures optimal database performance and identifies inefficient queries.
Cache Performance	Cache Hit Ratio, Cache Eviction Rate, Cache Latency	Important for high-read systems to ensure reduced database load.
API Performance	API Latency, API Throughput, Error Rate per Endpoint	Helps in tracking the performance and reliability of critical APIs.
Security	Authentication Success/Failure Rate, Unauthorized Access Attempts	Ensures the system's security posture remains robust.
Business	Revenue per User (RPU), Customer Retention Rate	Aligns technical performance with overarching business goals.

Tools for Monitoring Performance Metrics

- **APM Tools:** Datadog, New Relic, Dynatrace
- **Monitoring & Alerting:** Prometheus, Grafana
- **Log Analysis:** ELK Stack (Elasticsearch, Logstash, Kibana), Splunk
- **Cloud Monitoring:** AWS CloudWatch, Google Cloud Monitoring, Azure Monitor

DNS & Internet Infrastructure

This layer forms the backbone of how users find and connect to your services on the internet.

DNS Resolution Flow & Caching

The Domain Name System (DNS) is the phonebook of the internet. It translates human-friendly domain names (like www.google.com) into machine-readable IP addresses (like 142.250.196.196).[1] The resolution process is a journey that involves multiple layers of caching to speed things up.[2]

Here's the step-by-step flow when you type a URL into your browser:

1. **Browser Cache:** The browser first checks its own cache.[3] If you've visited the site recently, the IP address might already be stored here. This is the fastest lookup.
2. **Operating System (OS) Cache:** If the browser cache misses, the browser makes a system call to the underlying OS. The OS maintains its own cache of DNS lookups.[4]
3. **Router Cache:** If the OS cache misses, the request goes to your local network router, which often has its own DNS cache.
4. **ISP DNS Resolver:** If all local caches miss, the request is sent to your Internet Service Provider's (ISP) recursive DNS server.[5] This is the start of the "public" DNS query. The ISP's server will now do the heavy lifting:
    - It first contacts a **Root DNS Server**.[6] The root server doesn't know the IP, but it knows where to find the server for the Top-Level Domain (TLD).[7] It directs the resolver to the .com TLD server.[8]
    - The resolver then queries the **TLD Server**. The .com server doesn't have the final IP but knows the **Authoritative Name Server** responsible for the google.com domain.[9]
    - Finally, the resolver queries the **Authoritative Name Server** (often managed by the domain registrar or hosting provider like GoDaddy or AWS Route 53).[10] This server holds the actual IP address in a DNS record and returns it to the ISP resolver.[11]
5. **Caching the Result:** The ISP resolver caches this IP address for a certain period (defined by the **Time-To-Live or TTL** value in the DNS record) and sends it back to your OS, which then passes it to the browser.[12]

This caching at every step ensures that the full, multi-step lookup process is only done once in a while, making the internet feel fast.

Domain Registration, TLDs, and Anycast Routing

- **Domain Registration (e.g., GoDaddy):** This is the process of purchasing a domain name from a **Domain Registrar** like GoDaddy, Namecheap, or Google Domains.[13]

  When you register a domain, you are essentially leasing it for a period.[14] The registrar updates the registry for the corresponding **Top-Level Domain (TLD)** with your ownership details and the addresses of your authoritative name servers.
- **Top-Level Domains (TLDs):** These are the suffixes at the end of a domain name, like `.com`, `.org`, `.gov`, or country-specific ones like `.in` and `.co.uk`. [15]They are managed by specific organizations under the authority of ICANN (Internet Corporation for Assigned Names and Numbers).[16]
- **Anycast Routing:** This is a powerful networking technique where a single IP address is assigned to multiple servers in different geographical locations.[17] When a request is sent to an Anycast IP, the network automatically routes the user to the "nearest" server based on the lowest network latency.[18] This is heavily used by major DNS providers and CDNs to:
  - **Reduce Latency:** Users get responses from a server that is geographically closer to them.
  - **Improve Availability:** If one server location goes down, traffic is automatically rerouted to the next nearest location without any service interruption.