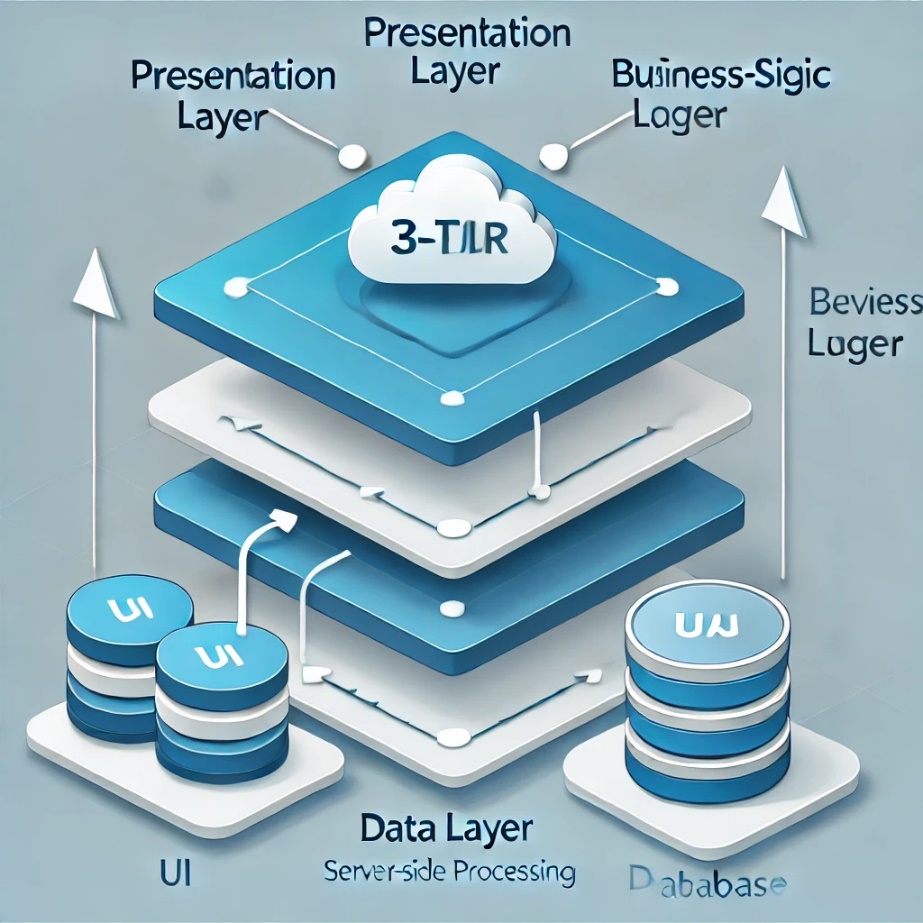
**System Design Overview:**

System design is the process of defining the architecture, components, modules, interfaces, and data for a system to meet specified requirements. It breaks down a complex problem into smaller, manageable parts and ensures these components work together to achieve key objectives like scalability, reliability, performance, and maintainability.

**Key Aspects:**

* **Architecture:** Defines the high-level structure of the system, including how various components interact.
* **Components:** Individual parts of the system, each with distinct functionality (e.g., services in microservices).
* **Interfaces:** Defines how different components communicate with each other (e.g., APIs).
* **Data Design:** Organizes how data is stored, retrieved, and managed within the system, often involving database design and access optimization.



**Design Patterns:**

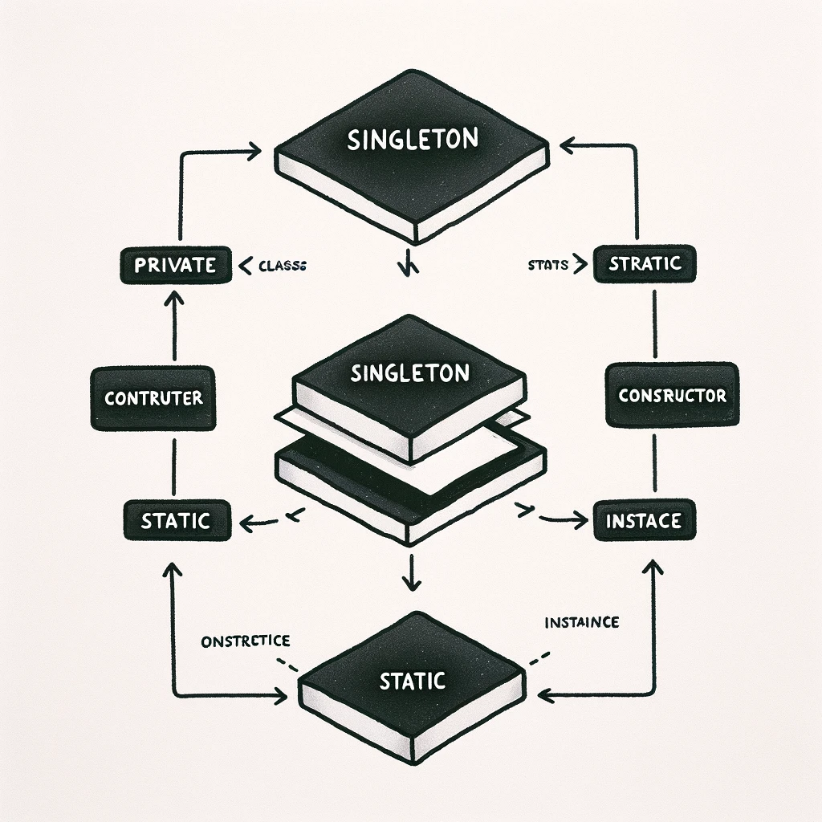
Design patterns offer reusable solutions to common problems in software design. They enhance scalability, maintainability, and efficiency in code and are typically categorized into three main types:

1. **Creational Patterns:** Manage the creation of objects, ensuring proper instantiation and efficient memory management.
   * **Singleton:** Restricts object creation to one instance.
   * **Factory Method:** Defers instantiation of objects to subclasses, promoting flexibility.
2. **Structural Patterns:** Focus on how classes and objects are composed to form larger, cohesive systems.
   * **Adapter:** Allows incompatible interfaces to work together.
   * **Composite:** Combines objects into tree structures to represent hierarchies.
3. **Behavioral Patterns:** Concerned with communication and interaction between objects.
   * **Observer:** Ensures one object automatically updates when another changes.
   * **Strategy:** Allows an algorithm's behavior to be selected at runtime.

Using design patterns helps standardize best practices, making code easier to understand and maintain.

**Example for Singleton Pattern:** In logging systems, you often only need one logger object that is shared across all classes in the application. A Singleton ensures that the same logging instance is used by different parts of the system, preventing the creation of multiple logger instances.

**Example for Factory Pattern:** In a content management system (CMS), the Factory Pattern is used to create objects like articles, blogs, and pages. Depending on the user’s selection, the factory instantiates the correct type of content.



This UML diagram shows how only one instance of a class is created and used.

**Scalability in System Design:**

Scalability is critical for systems expected to handle increasing amounts of data or users. It ensures that a system can expand without sacrificing performance.

**Types of Scalabilities:**

* **Vertical Scaling (Scaling Up):** Adds resources (CPU, RAM) to an existing server, improving performance for single-server systems. However, it has hardware limits.
* **Horizontal Scaling (Scaling Out):** Adds more servers to distribute the load. This method provides virtually unlimited scalability but requires coordination between servers using load balancers.

**Example:** In a video streaming service like Netflix, horizontal scaling is employed by adding more servers to handle increased user demand during peak hours. Each server processes a portion of the load, ensuring that all users can stream content without disruption.

**Load Balancing:**

A load balancer distributes incoming network traffic across multiple servers, ensuring no single server gets overwhelmed. It helps maintain performance, reliability, and availability.

**Common Load Balancing Algorithms:**

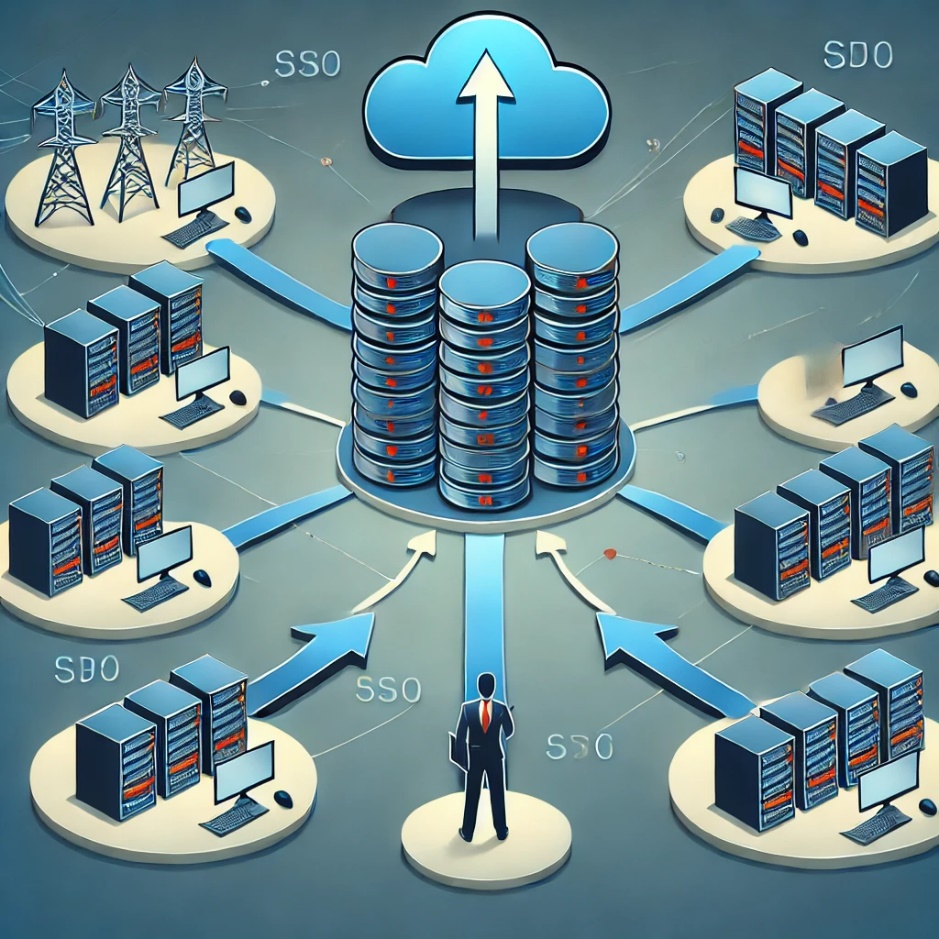
* **Round-robin:** Sequentially distributes requests to each server in a pool.
* **Least Connections:** Routes traffic to the server with the fewest active connections.
* **IP Hashing:** Consistently routes requests from the same client to the same server by hashing the client’s IP address.

**Benefits:**

* **Improved Availability:** If one server fails, the load balancer redirects traffic to operational servers.
* **Scalability:** As traffic increases, more servers can be added without downtime.
* **Enhanced Performance:** By distributing the load, the system remains responsive during heavy traffic.

Load balancing is essential for building resilient systems that can handle large volumes of traffic and ensure high availability.

**Example:** For a global e-commerce platform, a load balancer distributes user requests across different data centers located in multiple regions to ensure users experience fast response times, regardless of location. The load balancer uses a least connections algorithm to send traffic to servers with the least active connections.



This is how a load balancer distributes traffic across multiple servers.

**Fault Tolerance and Reliability:**

Reliability ensures a system functions correctly even when part of it fails. **Fault tolerance** is a key component of reliability, allowing a system to recover from errors or failures without significant impact on users.

**Fault Tolerance Techniques:**

* **Redundancy:** Duplicate critical components (e.g., multiple servers or databases) to take over if one fails.
* **Failover:** Automatically switches to a backup system when the primary system fails.
* **Graceful Degradation:** If one component fails, the system continues to operate in a reduced capacity rather than shutting down entirely.

By incorporating fault tolerance into the system design, downtime is minimized, and reliability is enhanced.

**Example:** In cloud infrastructure, platforms like Amazon Web Services (AWS) use redundant **servers** spread across multiple availability zones. If one server or zone goes down, a failover mechanism ensures users are redirected to a backup server without any noticeable downtime.

**Network Protocols:**

Network protocols govern how data is transmitted across networks, ensuring different devices can communicate effectively.

**Types of Network Protocols:**

* **Transport Layer Protocols:** Ensure reliable data transfer between systems (e.g., TCP, UDP).
* **Application Layer Protocols:** Facilitate communication between applications (e.g., HTTP, FTP, DNS).
* **Real-Time Communication Protocols:** Enable low-latency communication for real-time applications (e.g., WebSocket, RTP).
* **Security Protocols:** Secure data transmission (e.g., SSL, TLS, IPsec).

**Stateful protocols** (e.g., TCP) maintain a connection across multiple requests, allowing for smoother communication, while **stateless protocols** (e.g., HTTP) treat each request independently, requiring fewer resources but needing more context in each request.

**Caching:**

Caching stores copies of frequently accessed data in a temporary storage area, allowing faster access. It reduces latency and offloads traffic from the primary data source, improving system performance.

**Types of Caches:**

* **Memory Cache:** Stores data in RAM for quick access.
* **Disk Cache:** Stores data on disk, providing persistence but slower access than memory.
* **Web Cache:** Caches web content for fast access and reduced server load.
* **Distributed Cache:** Used across multiple servers, enabling high-speed access in distributed systems.

**Cache Invalidation:**

Ensures outdated data is replaced or removed from the cache. Methods include:

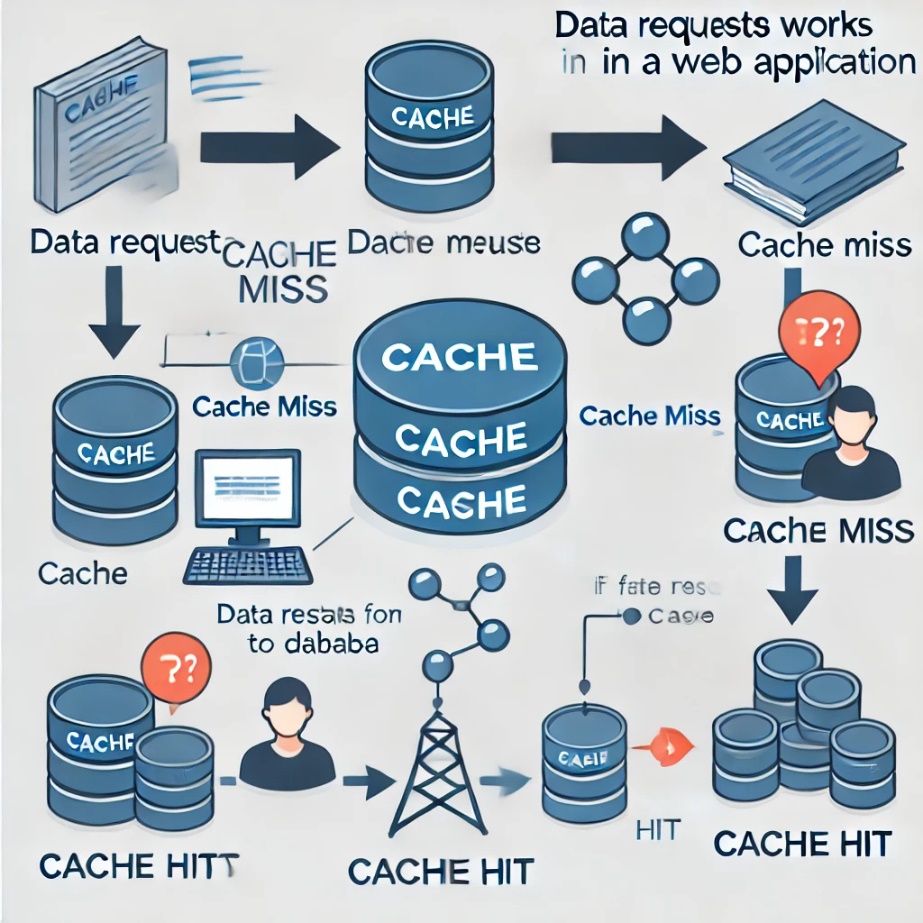
* **Time-based Invalidation:** Data expires after a set period (TTL).
* **Event-based Invalidation:** Cache is updated when underlying data changes.

**Cache Eviction Policies:**

When the cache is full, older data must be removed to make room for new data:

* **Least Recently Used (LRU):** Evicts the data that hasn’t been accessed for the longest time.
* **First In, First Out (FIFO):** Evicts the oldest data first.

**Example:** In a news website, caching frequently accessed articles reduces load on the main database. When users request a popular article, the cache returns the data instead of querying the database. This speeds up response time and reduces server load, especially during high traffic events like breaking news.



Flow of requests in a web application, showing cache hits and cache misses.

**Sharding:**

**Sharding** is a database architecture technique that divides large datasets into smaller pieces, or **shards**, and stores them across multiple servers. It enhances performance and scalability by distributing the load.

**Types of Sharding:**

* **Horizontal Sharding:** Divides rows of a table across shards based on a key (e.g., user ID ranges).
* **Vertical Sharding:** Divides tables by functionality (e.g., separating user data from transactions).
* **Directory-Based Sharding:** Uses a lookup table to map data to specific shards.

Sharding is crucial for large-scale systems like social media platforms, which handle vast amounts of data and need to optimize query performance.

**Indexing:**

An **index** is a data structure that improves the speed of data retrieval by reducing the need to scan entire datasets.

**Types of Indexes:**

* **Single-Column Index:** Indexes a single column of a table.
* **Composite Index:** Indexes multiple columns for complex queries.
* **Full-Text Index:** Optimizes searches on large text datasets (e.g., for search engines).
* **Spatial Index:** Used for geographical data, improving location-based queries.

Indexes can drastically improve query performance, but they also require storage and may slow down write operations as the index must be updated with every change.

**Example:** In a search engine, full-text indexing allows users to quickly search through millions of documents. By indexing specific keywords and phrases, the search engine can retrieve relevant results faster than scanning the entire dataset each time.

**MapReduce:**

**MapReduce** is a distributed computing framework used for processing large datasets. It divides the task into two phases:

* **Map:** Processes input data by breaking it into smaller chunks and applying a function to each chunk, producing key-value pairs.
* **Reduce:** Aggregates the results from the map phase and combines them to produce the final result.

MapReduce is commonly used for big data tasks like large-scale log analysis, data processing, and aggregation across clusters.

**Proxy Servers:**

A **proxy server** acts as an intermediary between clients and servers. It enhances security, improves load balancing, and provides anonymity.

**Types of Proxy Servers:**

* **Forward Proxy:** Acts on behalf of clients, sending requests to servers.
* **Reverse Proxy:** Sits in front of servers, distributing incoming client requests.
* **Transparent Proxy:** Intercepts requests without altering them, often used for caching.
* **Anonymous Proxy:** Hides the client’s IP address, enhancing privacy.

Proxies can also cache frequently requested content, improving response times and reducing the load on backend systems.

**Messaging Queues:**

A **messaging queue** facilitates asynchronous communication between different components of a system by temporarily storing messages until they are processed.

**Common Types of Messaging Queues:**

* **Point-to-Point:** Messages are sent from one sender to one receiver.
* **Publish-Subscribe:** Messages are broadcast to multiple receivers (subscribers).
* **FIFO (First In, First Out):** Ensures that messages are processed in the order they were sent.
* **Priority Queue:** Allows high-priority messages to be processed first, regardless of when they were sent.

**Apache Kafka** and **RabbitMQ** are common messaging systems used to decouple components in distributed systems, allowing for reliable and scalable communication.

**Example:** In e-commerce platforms, Apache Kafka is used for processing user-generated events like purchases, product reviews, and notifications. The publish-subscribe model ensures that messages about new purchases are processed in real time and passed to systems responsible for sending confirmation emails, updating inventory, and processing payments.

**Monolithic vs. Microservices Architecture**

**Monolithic Architecture:**

* **Monolithic:** All components of the application are bundled together into one large codebase.
  + **Pros:** Simple to develop, test, and deploy for small-scale applications.
  + **Cons:** Hard to scale and maintain as the system grows, changes require redeploying the entire application.

**Microservices Architecture:**

* **Microservices:** Breaks down the application into smaller, independent services that communicate via APIs.
  + **Pros:** Easier to scale and maintain, individual services can be deployed independently.
  + **Cons:** More complex to manage and requires robust monitoring and orchestration.

Microservices are ideal for large, complex systems that require flexibility and scalability, while monolithic architectures are better suited for simpler applications.

**Hashing and Consistent Hashing**

**Hashing** converts input data into a fixed-size value, known as a hash, using a hash function. It is commonly used for:

* **Data retrieval:** Quick lookups in large datasets.
* **Password storage:** Securely stores hashed passwords instead of plain text.
* **Digital signatures:** Ensures data integrity and authenticity.

**Consistent hashing** is a technique that minimizes data movement when scaling a system (e.g., adding or removing servers). It is commonly used in distributed systems like distributed databases and caching systems to ensure even data distribution across nodes.

**Example:** Git uses hashing to generate a unique hash for each commit, ensuring that changes to code are tracked efficiently. In distributed databases like Cassandra, consistent hashing is used to distribute data evenly across nodes, ensuring that data is easily retrievable even when nodes are added or removed.