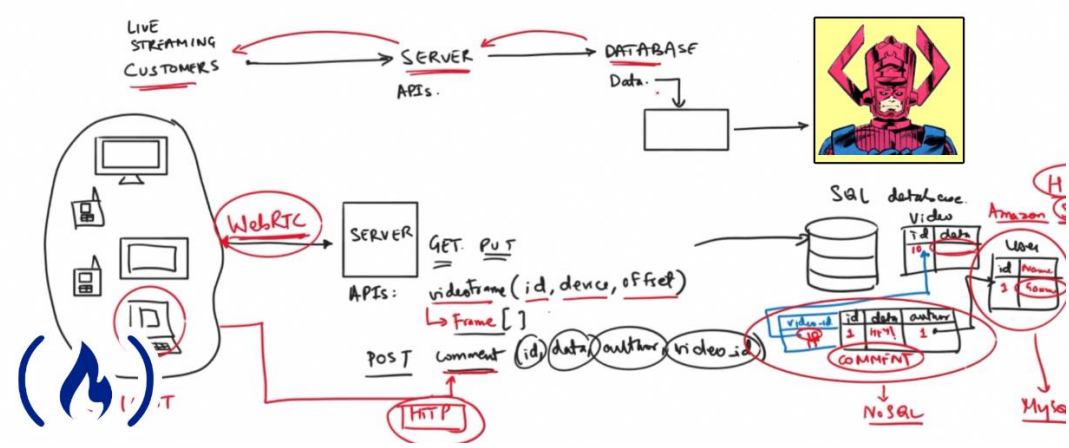
System Design

System design is the process of defining the architecture, components, interfaces, and data of a system to satisfy specified requirements. It involves breaking down a complex problem into smaller, more manageable parts and determining how those parts will work together to achieve the desired outcome.

- **Architecture:** The overall structure or blueprint of the system, including its components and how they interact.
- **Components:** The individual parts or modules that make up the system.
- **Interfaces:** How the components communicate and interact with each other.
- **Data:** How data is stored, retrieved, and managed within the system



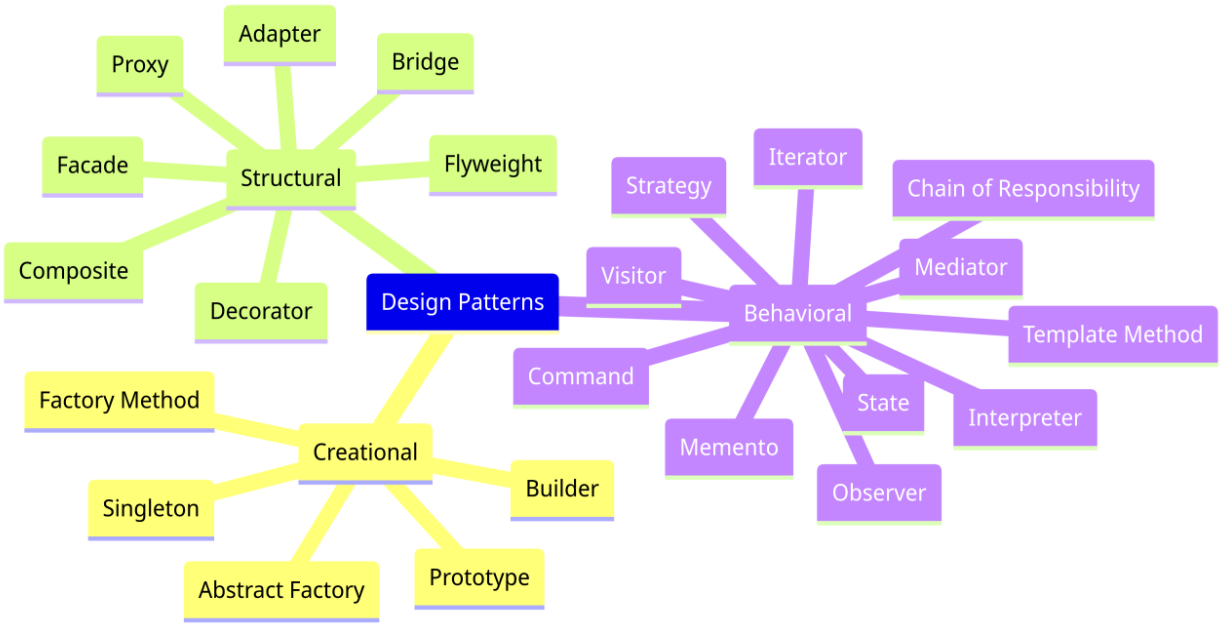
System design is a critical aspect of many fields, including software development, hardware engineering, network engineering, and business process management. It is a valuable tool that can help organizations achieve their goals by creating efficient, reliable, and effective systems.

Design Patterns: Design patterns are standardized solutions to common problems that arise in software design. They provide a template or blueprint that developers can use to address specific challenges in a flexible and efficient manner. Design patterns help promote best practices, enhance code reusability, and improve communication among developers.

There are three main categories of design patterns:

- **Creational Patterns**: These patterns deal with object creation mechanisms, aiming to create objects in a manner suitable to the situation. Examples include Singleton, Factory Method, and Abstract Factory.
- **Structural Patterns**: These patterns focus on how classes and objects are composed to form larger structures. Examples include Adapter, Composite, and Decorator.
- **Behavioral Patterns**: These patterns define how objects interact and communicate with one another. Examples include Observer, Strategy, and Command.

By using design patterns, developers can improve the scalability, maintainability, and clarity of their code, ultimately leading to more robust software solutions.



Choosing the right design pattern:

- **Analyze the problem:** Identify the specific design challenges or requirements.
- **Consider the benefits and drawbacks:** Evaluate the trade-offs of different patterns.
- **Balance simplicity and flexibility:** Aim for a solution that is easy to understand and maintain while being adaptable to future changes.

Key Steps:

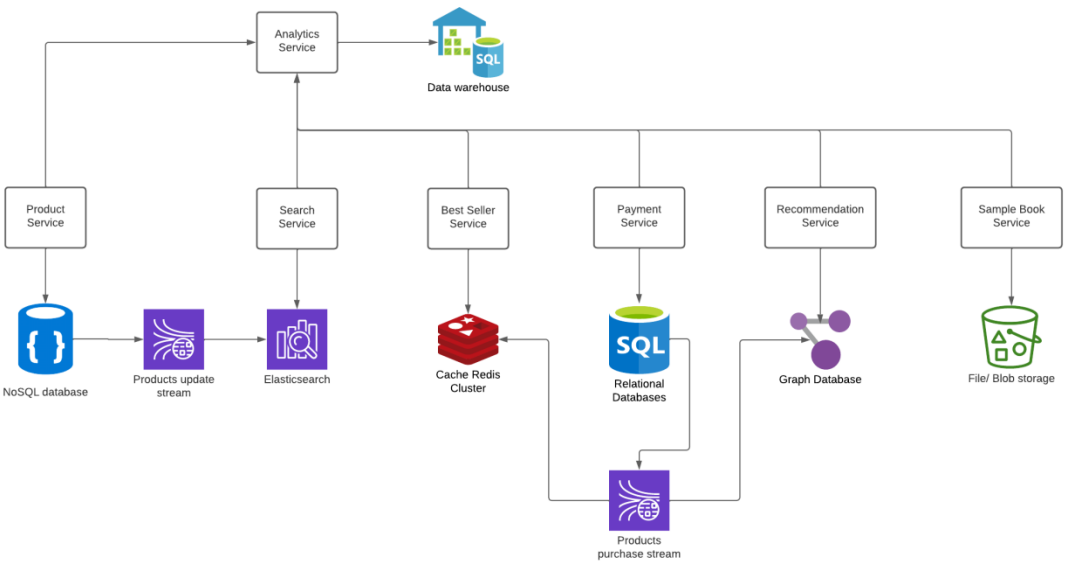
- **Problem Definition and Requirements Gathering:** Clearly define the problem and identify the key requirements for the system.
- **Feature Definition and Data Modeling:** Convert requirements into specific features, define data structures, and map them into objects.
- **Object-Oriented Design and Pattern Selection:** Design the system using an object-oriented approach and select appropriate design patterns.
- **Scalability and Redundancy:** Ensure the system can handle increasing load and protect against data loss by designing for scalability and redundancy.
- **Iterative Development and Testing:** Develop and test the system incrementally, refining the design and implementation based on feedback and results.

Extensibility: It is a crucial aspect of software design that allows a system to be easily modified or extended to accommodate new features or requirements without requiring extensive changes to existing code. Design patterns play a significant role in achieving extensibility by providing well-defined structures and mechanisms for adding new functionality.

Database storing the video and network protocols to query that and stream to the users.

Database: The choice of database plays a crucial role in system design and can significantly impact performance, scalability, and overall architecture. Here are some common types of databases to consider:

- **Relational Database:** MySQL, PostgreSQL, Oracle, SQL Server.
- **NoSQL Database:** MongoDB, Cassandra, Redis, Neo4j.
- **Hybrid Database:** PostgreSQL with JSONB or HStore extensions, MongoDB with Atlas Search.



Considering database:

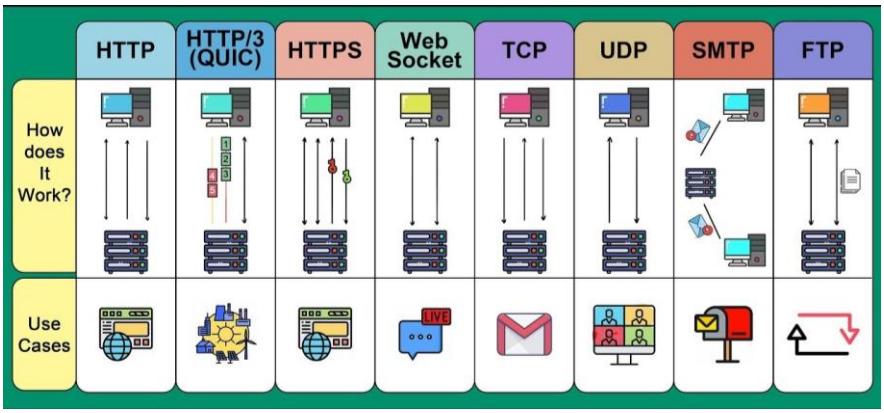
- Assess the type, structure, and volume of data to inform design decisions.
- Analyze the types of queries expected to optimize performance and efficiency.
- Determine scalability needs for handling large data and traffic and decide on the level of consistency required.
- Consider licensing, hardware, and operational costs, and choose design patterns that align with the system's requirements.

For example we will consider the streaming platform such as Youtube / vimeo :

- Accept client requests from various devices, including mobile phones, tablets, desktops, laptops, and TVs.
- Once a request is received, send it to the server for processing.
- The server executes the appropriate actions based on the request type and gathers data as needed.
- For GET requests, provide the necessary data; for POST requests, save the data to the database accordingly.
- Organize and map the data into the appropriate database tables to ensure efficient data management.

Network Protocols:

Network protocols are standardized rules and conventions that govern the communication between devices on a network. They define how data is transmitted, received, and processed, ensuring that devices can understand each other and interact effectively.



Types of network protocols:

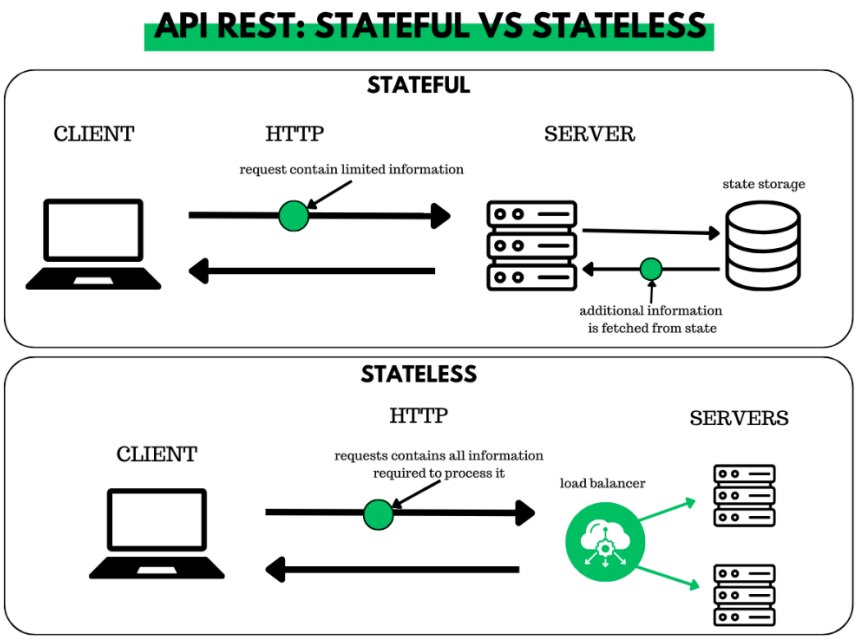
- **Transport Layer Protocols**: TCP, UDP.
- **Application Layer Protocols:** HTTP, FTP, SMTP, POP, IMAP, SIP, DNS, LDAP.
- **Network Layer Protocols**: IP, ICMP.
- **Real-Time Communication Protocols:** WebSocket, RTP.
- **Security Protocols**: SSL, TLS, IPsec.
- **Wireless Protocols**: Wi-Fi, Bluetooth
- **File Sharing Protocols:** NFS, SMB.

Stateful Protocol: A stateful protocol maintains connection state across multiple interactions, allowing the server to remember past requests and responses. This enables seamless user experiences but requires more server resources for session management.

Examples: TCP, FTP, SIP, HTTP/1.1 with Cookies, RPC.

Stateless protocols: A stateless protocol does not retain information about previous interactions between the client and server. Each request is treated independently, containing all the necessary information for processing. This simplifies server design and reduces resource usage but may require additional data to be sent with each request to maintain context.

Examples: HTTP, UDP, SMTP, DNS, REST.



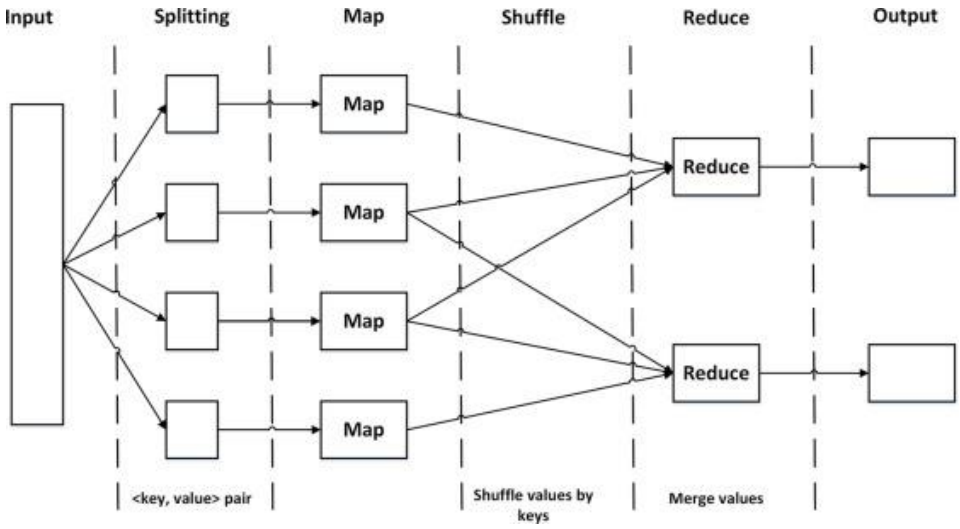
Stateful vs Stateless protocols:

Aspect	Stateful Protocols	Stateless Protocols
Connection State	Maintains connection information	Does not retain information
Resource Usage	Requires more server resources	Uses fewer resources
Complexity	More complex due to session management	Simpler design
Performance	Better for multiple interactions	May have overhead due to redundant data
Applications	TCP (web browsing), FTP (file transfers), SIP (VoIP)	HTTP (web pages), UDP (video streaming), DNS (domain resolution)
Real-Time Applications	Online banking (session management), multiplayer gaming	REST APIs (web services), live chat

We also consider the database based on the files we are storing, and cost efficient, reliable and cheap and fast access.

Map-Reduce: MapReduce is a programming model and framework used for processing large datasets across clusters of computers. It's a popular choice in big data processing and distributed systems.

- **Map:** The first stage of the process. It takes a large dataset and breaks it down into smaller chunks, applying a user-defined function to each chunk to produce key-value pairs.
- **Reduce:** The second stage. It combines the key-value pairs produced by the map phase for each key, applying another user-defined function to produce a result.



High-Level Design: It is a blueprint of a system.

HLD is the initial phase of system design, where you outline the overall architecture, components, and interactions of a software system. It provides a foundational blueprint for the subsequent detailed design and implementation phases.

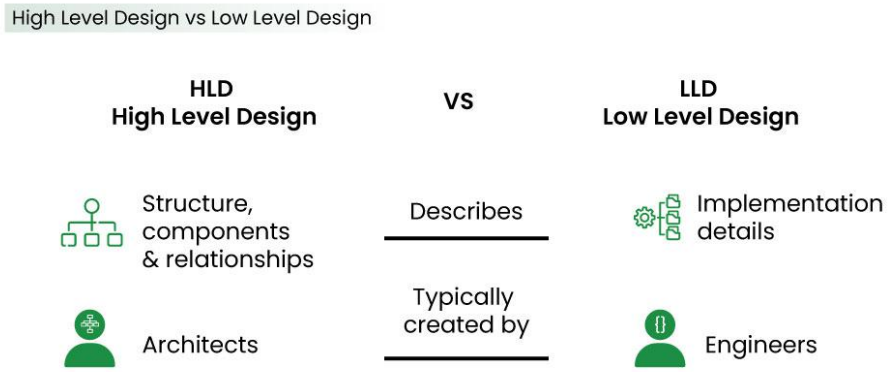
- System architecture outlines the overall structure, components, and interactions, while component diagrams show dependencies.
- Data flow diagrams illustrate the movement of data through the system, highlighting input, processing, and output stages.
- Use case diagrams depict interactions between the system and its users or external systems, along with class diagrams detailing classes and relationships.
- Deployment diagrams represent the physical distribution of the system's components across hardware.

Low-level Design: Diving into the Details

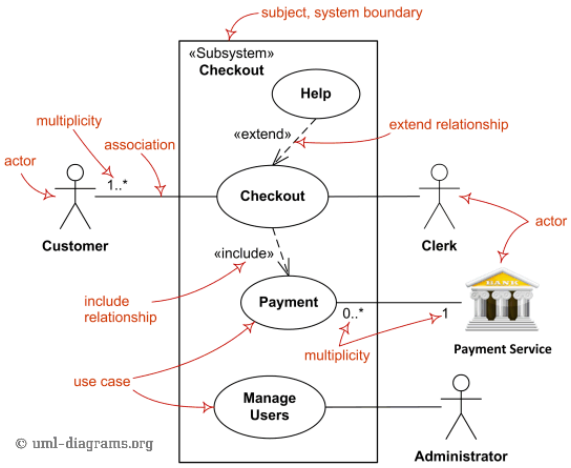
LLD is the next step after high-level design. It focuses on defining the internal structure and implementation details of individual components within the system.

- Detailed class diagrams define attributes, methods, and relationships between classes, while sequence diagrams illustrate interactions between objects over time.

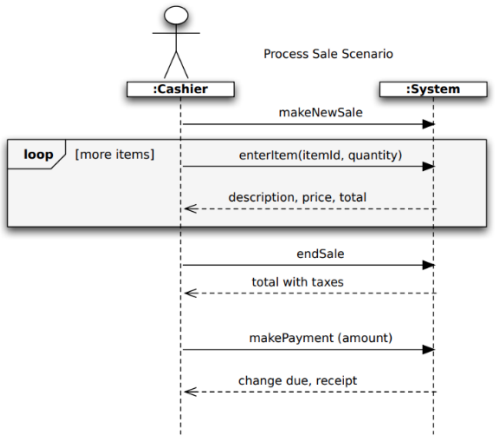
- Activity and data flow diagrams showcase the flow of control and detailed data movement within the system.
- Specify algorithms and data structures used within components.
- Outline error handling strategies and consider performance optimization techniques such as caching, indexing, and algorithm improvements.



Use Case diagram:



Sequence diagram:



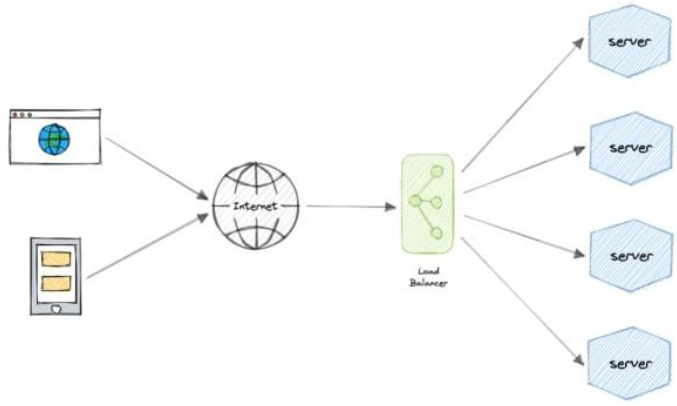
Load balancers: Load balancers are systems or devices that distribute incoming network traffic across multiple servers to optimize resource use, ensure high availability, and improve application performance by preventing any single server from becoming a bottleneck.

There are various types as well few of them are as follows:

- **Hardware Load Balancers**: Physical devices designed to distribute traffic and optimize performance, often used in enterprise environments.
- **Software Load Balancers**: Applications that perform load balancing tasks, deployed on standard servers. Examples include HAProxy and Nginx.
- **Cloud Load Balancers**: Managed services provided by cloud platforms, such as Google Cloud Load Balancing, AWS Elastic Load Balancing, and Azure Load Balancer.

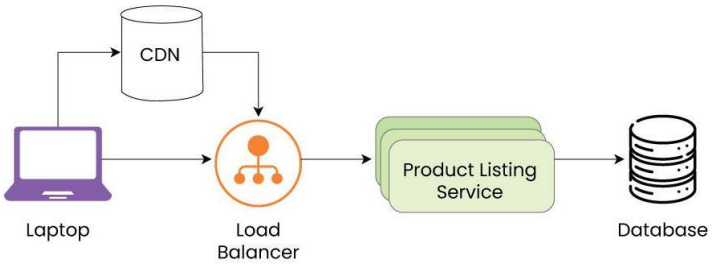
Use cases of these load balancers:

- **Web Applications:** Distributing user traffic across multiple servers to enhance responsiveness and ensure high availability.
- **Microservices Architectures:** Managing communication and routing requests between various microservices for improved scalability.
- **High Availability Systems:** Ensuring continuous operation by rerouting traffic to healthy servers during failures.



Cache: Cache is a storage area that keeps copies of frequently accessed data, allowing faster retrieval. Instead of getting the same information from a slower source each time, the system can quickly pull it from the cache, making applications run faster and improving performance.

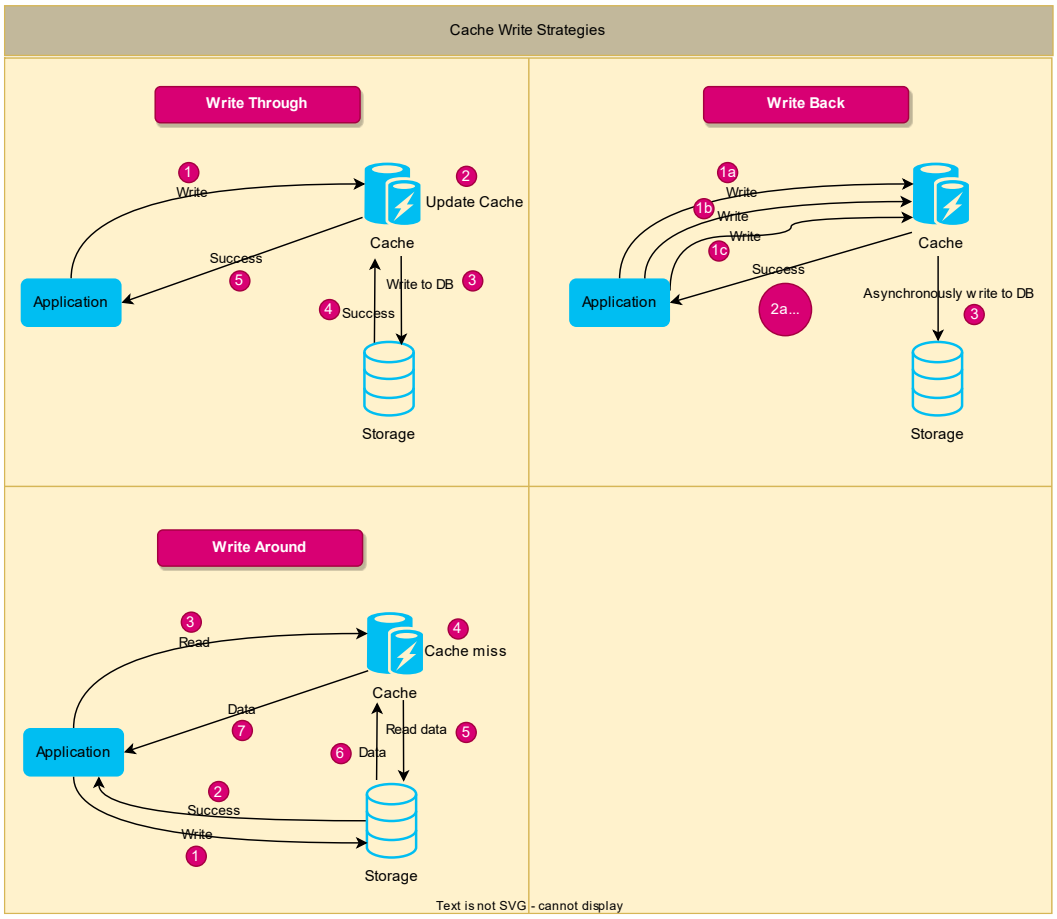
There are various types of caches such as: Memory Cache, Disk Cache, Web Cache, CPU Cache, Distributed Cache.



Cache Invalidation:

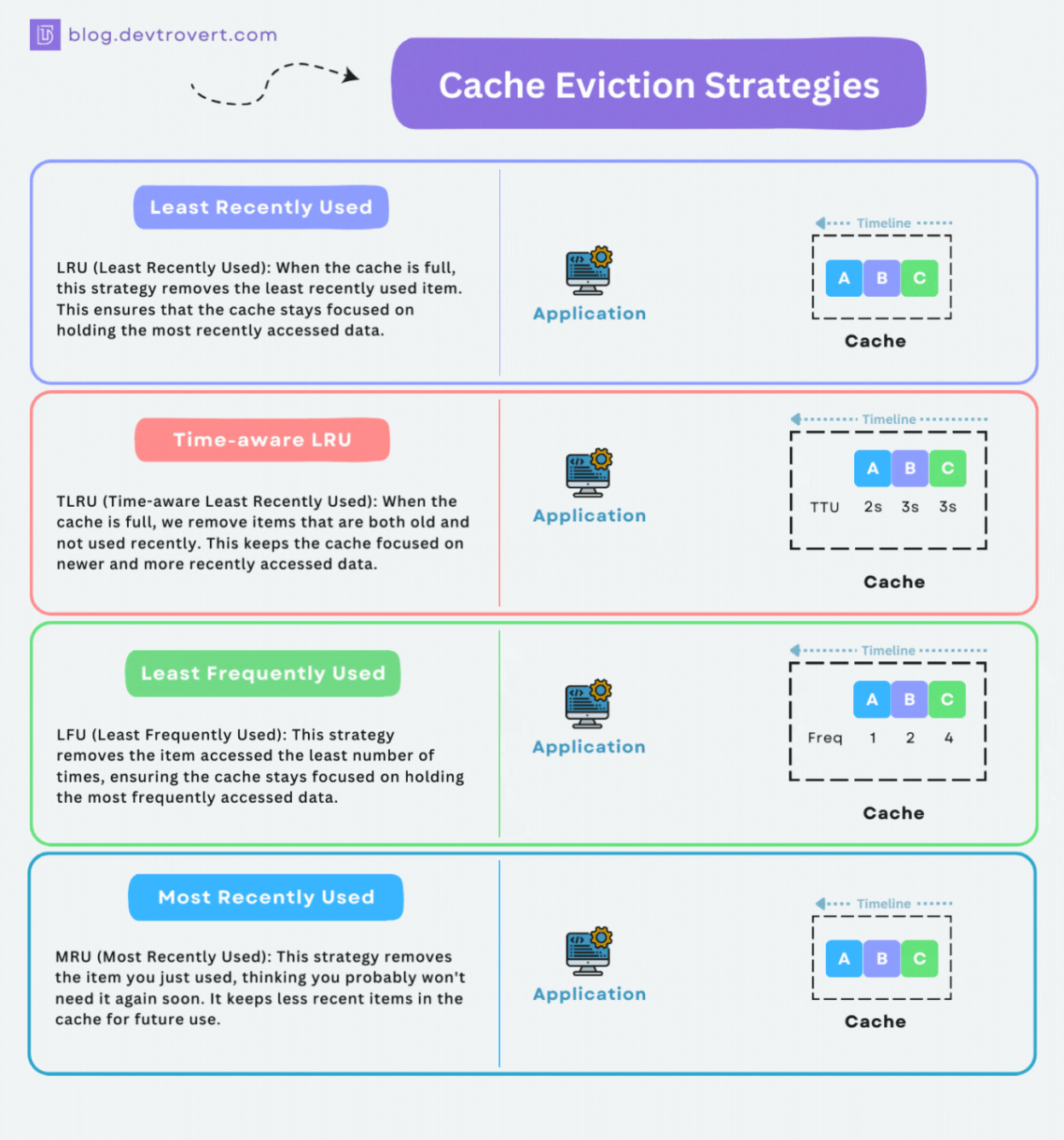
Cache invalidation is the process of removing or updating cached data to ensure that stale or outdated information is not served to users. It's essential for maintaining data consistency between the cache and the primary data source. Common strategies include time-based expiration, event-based invalidation, or manual invalidation when data changes.

- **Write Through:** Data is written to both the cache and the database simultaneously, ensuring consistency but potentially increasing latency.
- **Write Around:** Data is written directly to the database, bypassing the cache. This reduces cache pollution but may lead to slower reads for recently written data.
- **Write Back:** Data is initially written only to the cache, with updates sent to the database later. This improves write performance but risks data loss if the cache fails before the write occurs.



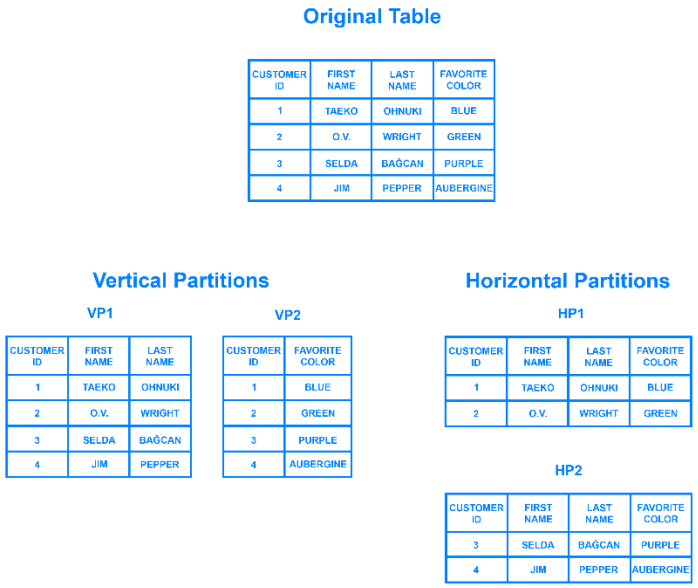
Cache eviction policies: It determine which items to remove from a full cache to optimize performance by retaining relevant data and discarding less useful items.

Types: Least Recently Used (LRU), First In, First Out (FIFO), Least Frequently Used (LFU), Random Replacement, Most Recently Used (MRU), Time-based Expiration, Segmented LRU (SLRU).



Sharding: Sharding is a database architecture pattern that divides a large dataset into smaller, manageable pieces called "shards." This improves scalability and performance by distributing data across multiple servers, allowing for parallel processing.

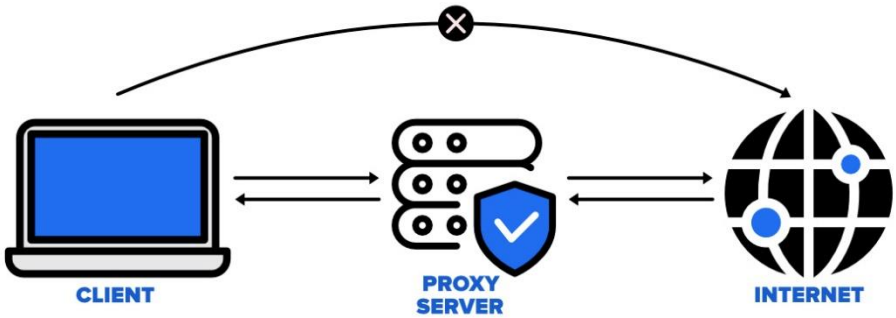
Types: Vertical, Horizontal, Directory based shardings.



Indexes: Indexes are data structures that improve the speed of data retrieval operations on a database table. They work like a book's index, allowing the database to find and access rows more quickly without scanning the entire table.

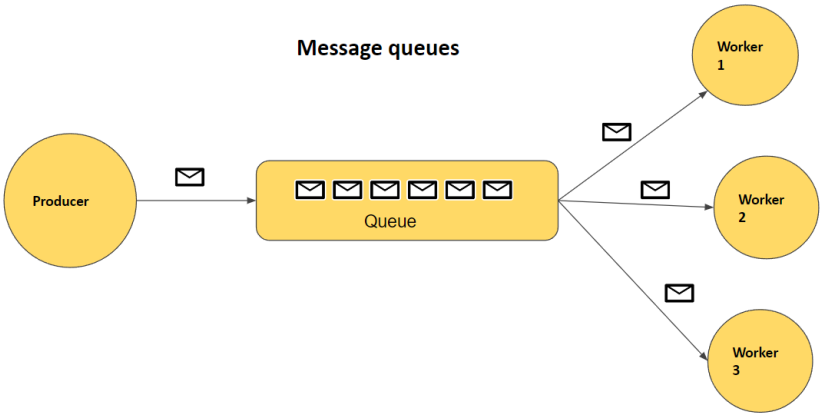
Proxy server: A proxy server acts as an intermediary between a user and the internet. It receives requests from clients, forwards them to the desired server, and then returns the response to the client. This can enhance security, improve load times, and provide anonymity by masking the user's IP address. Proxies can also be used to bypass geographic restrictions or filter content.

Types: Forward Proxy, Reverse Proxy, Transparent Proxy, Anonymous Proxy, High Anonymity Proxy, Web Proxy.



Message queue: A message queue is a communication system that allows different components to send and receive messages asynchronously. It acts as a buffer, storing messages from a producer until they can be processed by a consumer, enabling decoupling, scalability, and reliability in distributed systems.

Types: Point-to-Point, Publish-Subscribe, FIFO, Priority Queues, Distributed Queues, Transactional Queues.



Monolithic Architecture: Monolithic architecture is a software design where an entire application is built as a single, unified unit. All components are interconnected and run as one process, simplifying development but potentially complicating scalability and maintenance.

Use cases:

- **Small to Medium-Sized Applications:** Ideal for startups or projects with limited scope where simplicity and speed of development are priorities.
- **Internal Tools:** Tools developed for internal company use, where scalability and extensive integration are less critical.
- **E-Commerce Sites:** Smaller online stores that manage limited products and traffic without the need for a highly distributed system.
- **Single-User Applications:** Applications intended for individual use, such as desktop software or personal productivity tools.

Microservices Architectures: Microservices architecture is a design approach where an application is composed of loosely coupled, independently deployable services, each focusing on a specific business capability. This promotes flexibility and scalability but can introduce complexity in communication and management.

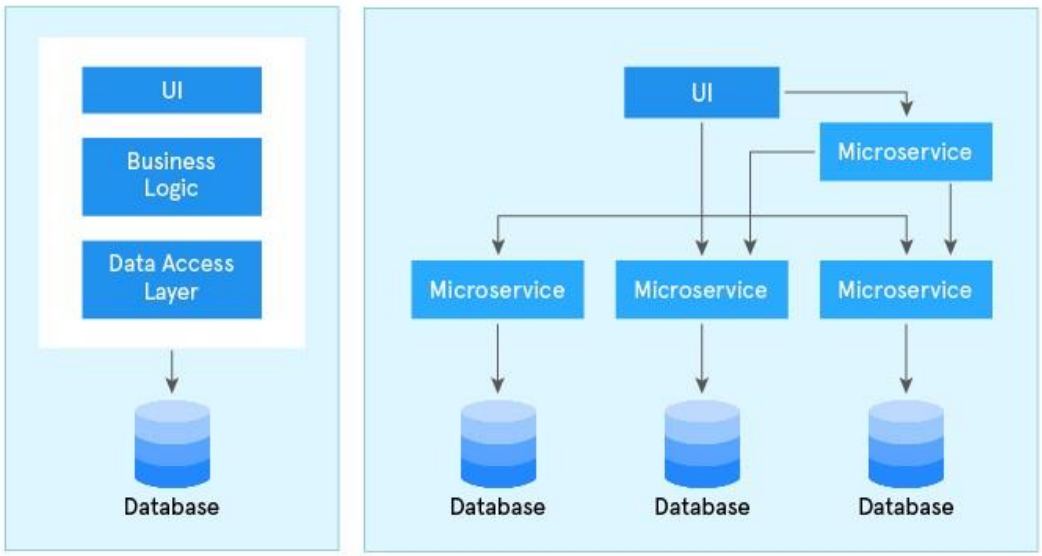
Use cases:

- **Enterprise Applications**: Large organizations that require different departments to have specialized, scalable applications integrated into a cohesive system.
- **Streaming Services**: Applications like Netflix or Spotify that need to manage multiple functionalities like user profiles, content delivery, and recommendations independently.

- **Social Media Applications**: Platforms that handle diverse functionalities like user profiles, messaging, and notifications, allowing for independent updates.
- **Banking and Financial Services**: Systems that require secure, scalable services for transactions, account management, and reporting.

Monolithic vs Microservices Architectures:

Feature	Monolithic Architecture	Microservices Architecture
Structure	Single codebase	Multiple independent services
Scalability	Limited, whole app scaling	Independent scaling of services
Deployment	Single deployment	Independent deployments
Technology Stack	Uniform stack	Diverse stacks per service
Fault Isolation	A failure impacts the entire app	Failures isolated to individual services



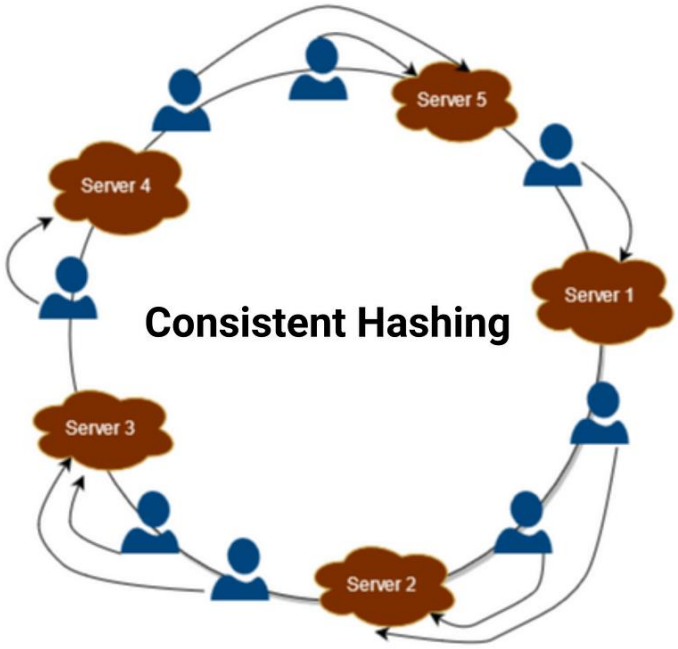
Monolithic Architecture Microservices Architecture

Hashing: Hashing is the process of converting input data into a fixed-size string, known as a hash value, using a hash function. It ensures data integrity, securely stores passwords, enables digital signatures, and facilitates fast data retrieval in hash tables. Key properties include determinism, quick computation, and collision resistance.

Types: Cryptographic Hash Functions, Non-Cryptographic Hash Functions, Checksums, Message Digests, Keyed Hashes.

Various types of collision techniques: Chaining, Open Addressing, Linear Probing, Quadratic Probing, Double Hashing, Robin Hood Hashing, Cuckoo Hashing.

Consistent Hashing: Consistent hashing is a technique used to distribute data across a dynamic set of nodes in a way that minimizes reorganization when nodes are added or removed. It maps both keys and nodes onto a circular space, allowing for even distribution and reducing the number of keys that need to be relocated during changes. This approach is particularly useful in distributed systems and load balancing.



Examples:
URL Shortener: In this system, we will provide a long URL, which will be shortened and returned to the user. When the user accesses the short URL, they will be automatically redirected to the original URL.

Key Requirements:

- **Content Pasting**: Users should be able to paste the original URL and receive a shorten URL.
- **URL Creation**: Upon pasting content, the system generates a unique short URL.
- **Redirection to original**: Accessing the short URL should redirect to the original URL.
- **Expiration Time**: Users can set an expiration time for the URL (e.g., 5 days, 5 years).
- **Custom URLs**: Users can create custom URLs if they do not already exist in the database.
- **API Rate Limiting**: Implement limits on the number of URLs a user can create within a specific timeframe.
- **Public/Private URLs**: Users can choose to create URLs that are either public or require authentication to access.

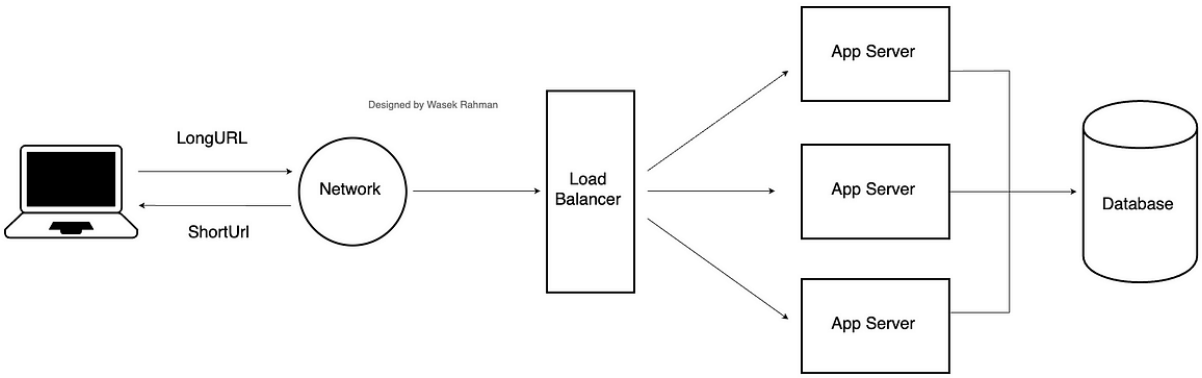
- **Analytics**: Recording the different parameters.

System Design Considerations:

- **API Development**: Identify necessary APIs for functionality, such as creating, retrieving, and managing URLs.
- **Database Design**: Determine the structure of the database, including tables for storing original URLs, shorten URLs, and user data.
- **Load Balancing**: Consider implementing load balancers to manage traffic effectively.
- **Caching**: Use caching strategies to improve performance and reduce database load.
- **Sharding**: Explore sharding options for scaling the database.
- **Analytics**: Implement analytics to track user interactions with URLs for insights and improvements.

Workflow

- **User Pastes Content:** User inputs text into the application.
- **Generate URL:** The system creates a unique shorten URL for the pasted URL.
- **Access URL:** When the shorten URL is accessed, the system retrieves and redirects to the original URL.
- **Manage Content:** Users can set expiration, create custom URLs, and choose privacy settings.



Pastebin: A web application that allows users to paste text and receive a unique URL for later access. Users can share this URL to retrieve the pasted content.

Key Requirements:

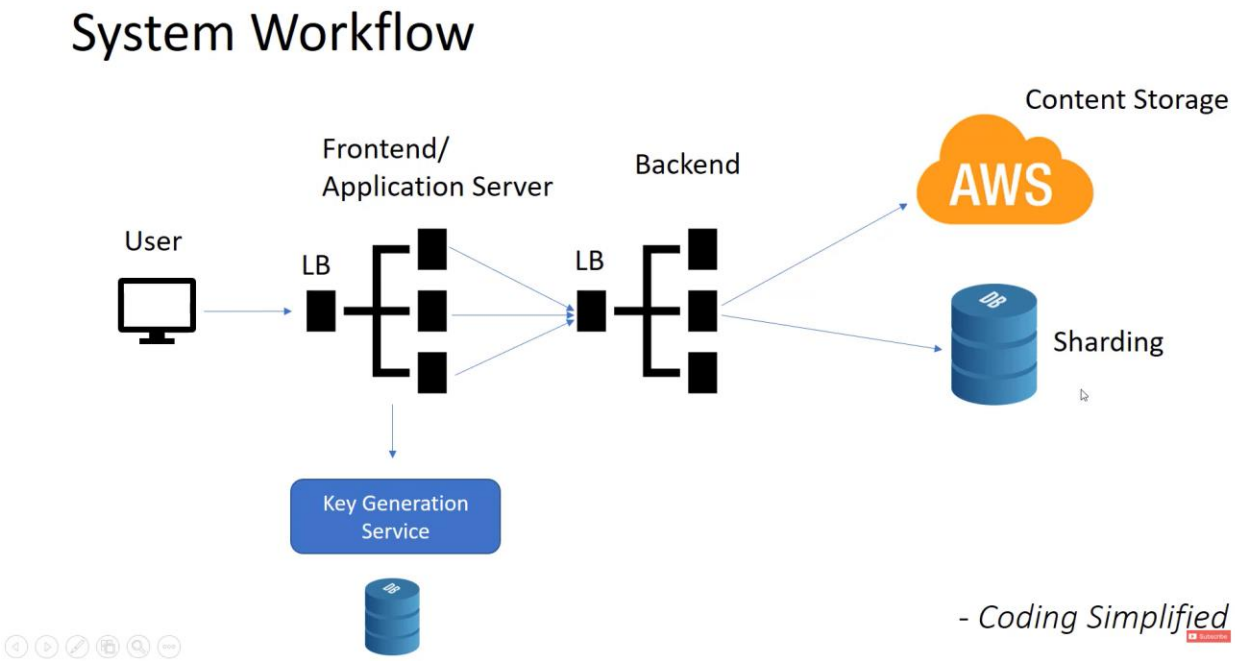
- **Content Pasting**:
    - Users should be able to paste content and receive a URL.
    - Example: User pastes "Hey everyone, creating video for pastebin" and receives a URL.
- **URL Creation**: Upon pasting content, the system generates a unique URL.
- **Content Retrieval**: Accessing the URL should display the pasted content.
- **Expiration Time**: Users can set an expiration time for the URL (e.g., 5 days, 5 years).
- **Custom URLs**: Users can create custom URLs if they do not already exist in the database.
- **API Rate Limiting**: Implement limits on the number of URLs a user can create within a specific timeframe.
- **Public/Private URLs**: Users can choose to create URLs that are either public or require authentication to access.

System Design Considerations:

- **API Development**: Identify necessary APIs for functionality, such as creating, retrieving, and managing URLs.
- **Database Design**: Determine the structure of the database, including tables for storing content, URLs, and user data.
- **Load Balancing**: Consider implementing load balancers to manage traffic effectively.
- **Caching**: Use caching strategies to improve performance and reduce database load.
- **Sharding**: Explore sharding options for scaling the database.
- **Analytics**: Implement analytics to track user interactions with URLs for insights and improvements.

Workflow

- **User Pastes Content:** User inputs text into the application.
- **Generate URL:** The system creates a unique URL for the pasted content.
- **Access URL:** When the URL is accessed, the system retrieves and displays the original content.
- **Manage Content:** Users can set expiration, create custom URLs, and choose privacy settings.



Dropbox: Services like Dropbox and OneDrive allow users to store data online instead of on local hard drives, which may have limited capacity.

Key Requirements:

- **File Management**:
  - Users should be able to **upload** and **download** files from any device (laptop, desktop, mobile).
  - Users can **share** files via generated short URLs.
  - Users should be able to **edit** files directly within the service.
- **Storage Limitations**: Non-premium users should have a storage limit (e.g., 10 GB). Additional storage requires payment.
- **API Integration**: The service should expose a **REST API** for uploading and downloading files programmatically.
- **Analytics**: Track user behavior, such as the number of files uploaded, downloaded, or edited for analytical purposes.
- **Version Control**: Maintain different versions of files to allow users to revert to previous states.

System Workflow:

- Users create a workspace on their devices where files are stored. Changes in this workspace should sync across all devices.
- Upon file upload, the backend service stores the file and its metadata (user info, file type, device used) in a database.

Database Design: The database can be either MySQL or NoSQL, with a focus on **atomicity** to ensure data integrity during uploads. MySQL is preferred due to its ACID compliance.

Load Balancing and Caching

- Implement load balancers between users and servers to manage traffic efficiently.
- Caching strategies can be applied to enhance performance.