**System –** Architecture or collection of technologies that work together to serve users to fulfill requirements of set of users.

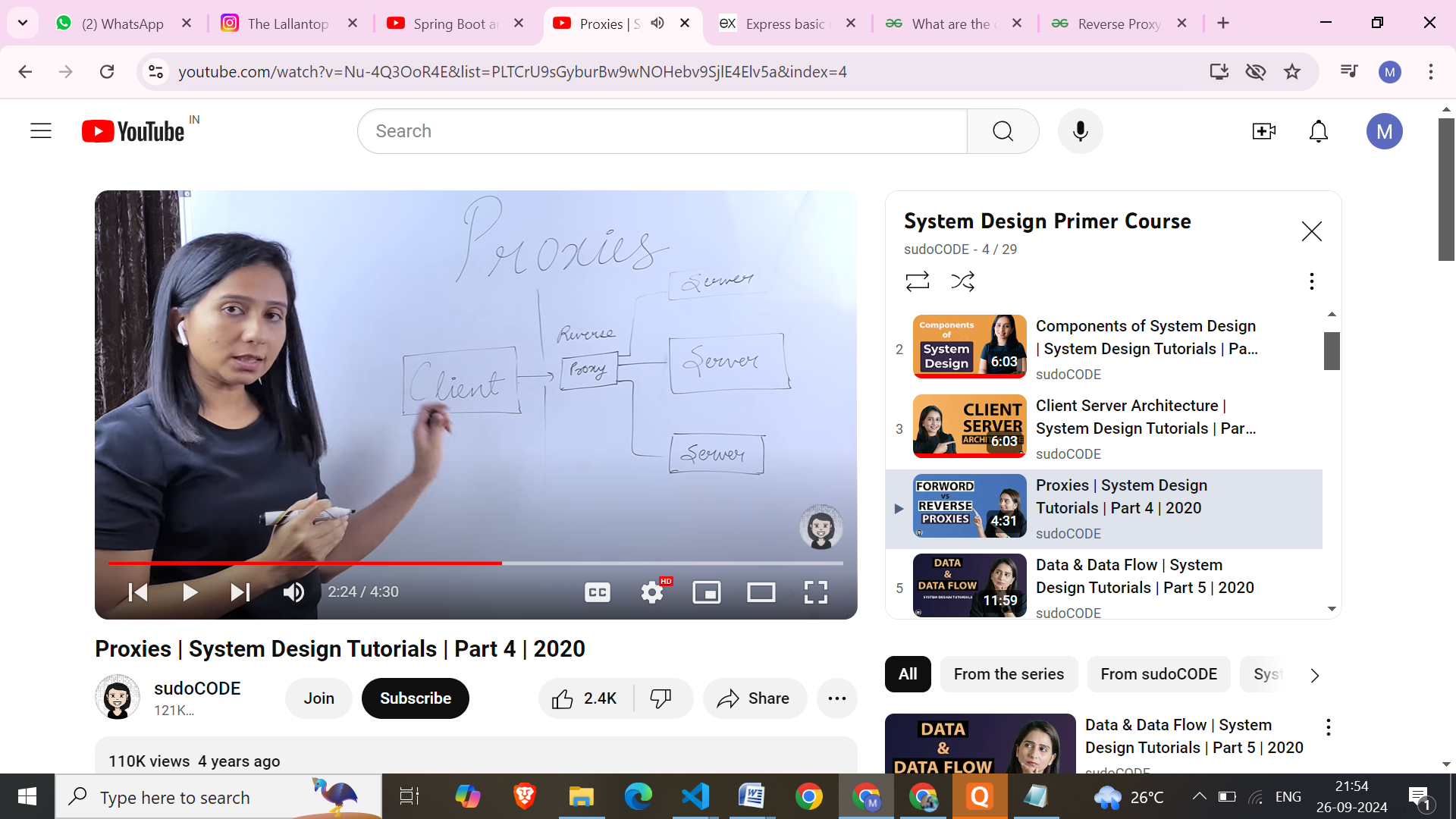
**Thick Client –** where the processing occurs at client side( video games)

**Thin Client –** where the processing occurs at server side( e commerce)

**n-tier architecture** – more that 3 layers i.e., introduction of load balancer or caches

**Proxy Server –** Two types **forward proxy** and **reverse proxy**

Forward proxy is better when there is no need to know the ip address of the client.   
Reverse Proxy is good when it is not known to the client to which server we need to send the request.



| **Feature** | **Reverse Proxy** | **Load Balancer** |
| --- | --- | --- |
| Functionality | Acts as an intermediary between clients and servers, handling requests and responses. It can also provide **caching**, SSL termination, and security features. | Distributes incoming traffic across multiple backend servers to improve performance, scalability, and reliability. It can also perform health checks on servers. **It is a mechanism that is included in gateways.** |

**\*\*\*\* Reverse** proxys/ load balancers are used within API gateways. Some examples of API gateways are: -- (to learn : Kong, traefik) (We can use built-in API gateways when we just need basic gateway features or dedicated gateways when we require separate configuration and monitoring.)

 **AWS API Gateway**: Manages API requests and integrates with backend services using reverse proxy capabilities.

 **Traefik(**Built-in **reverse proxy** and **load balancing)**: A modern reverse proxy that doubles as an API gateway for microservices and containers.

 **Kong(built-in proxy and load balancing)**: A popular open-source API gateway that uses reverse proxy for routing and adds advanced API management features.

 **NGINX Plus**: Acts as both a reverse proxy/ load balancer and an API gateway with extensions for authentication, rate limiting, and caching.

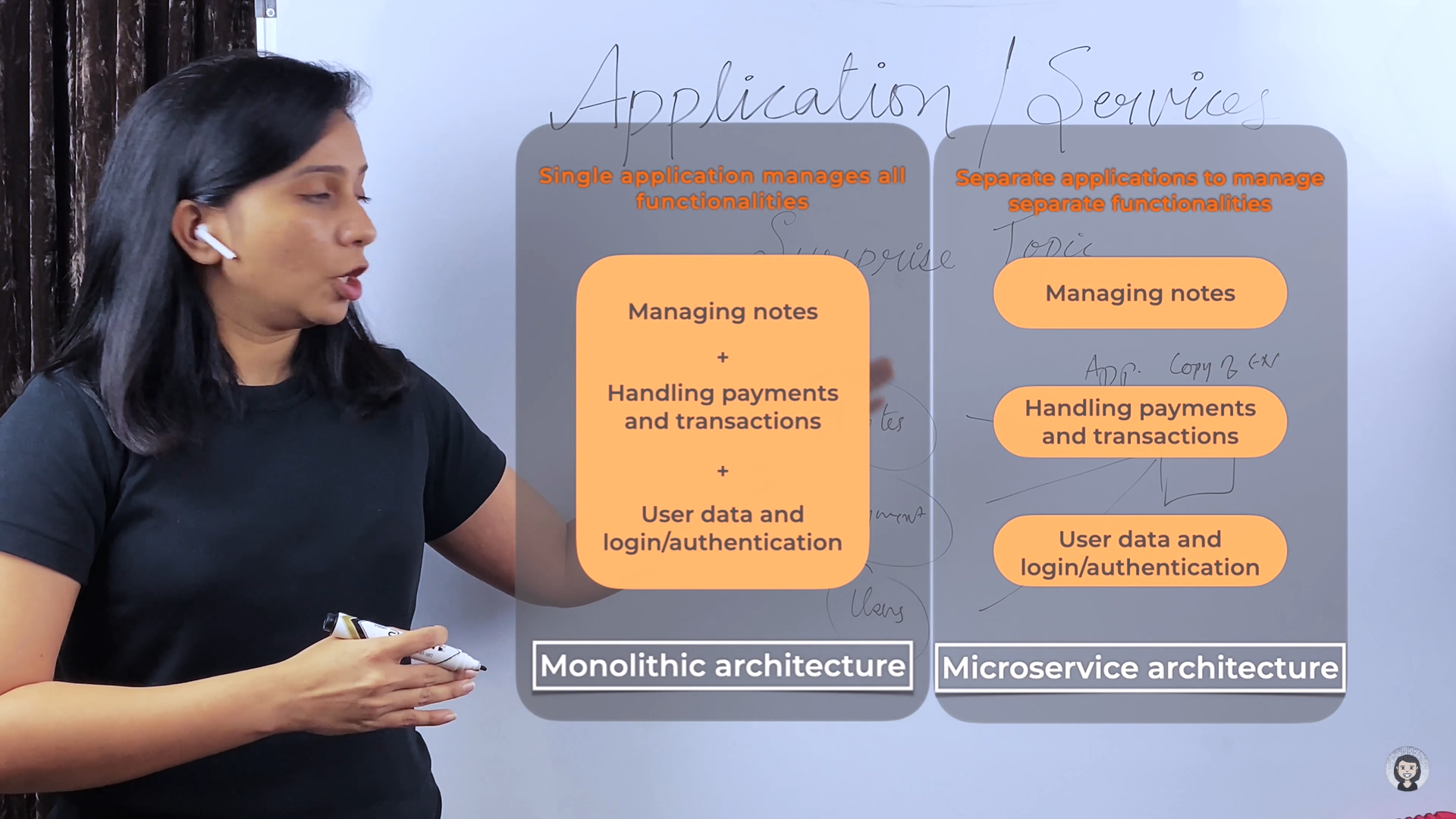
**SQL –** When the size of the DB becomes large and then we want to change the schema, it becomes complicated. Even in large and complex DB the joins become costly. Better option is to use document-based no-SQL (ex – MongoDB, Apache Cassandra)

If a DB requires heavy read/ write operation its better to use **column DBs. It** store data in columns rather than rows. Ex- Snowflake

**Search DBs** can be used on websites where a large number of search is required. Ex – for an Ecommerce website, the data can be in noSQL but the attributes that are needed to be search very frequently can be set up in search database.

**Application Types**

**Example :**

****

**Monolithic App:** It is said the monolithic approach is usually better for simple, lightweight applications. This approach is generally faster.

In a monolithic architecture, any changes made to one part of the application can have unintended consequences on other parts and it can be very difficult to scale or work with the individual components. These applications become more difficult to maintain and update as they grow in complexity.

 For frequent code changes or evolving scalability requirements, this approach is not at all suitable.

Stack Overflow uses this architecture.

**Microservices architecture** is a software development approach that organizes an application as a collection of independent services that communicate with each other. Google, facebook uses this architecture.

**Common Tech Stacks for Microservices**

1. **Java + Spring Boot + Docker + Kubernetes + Kafka**
2. **Node.js + Express + MongoDB + RabbitMQ + Docker**

* **Advantage-** Easier to scale. Parallel development of separate services is easy.
* **Disadvantage-** requires very smart architect

Microservices need to communicate with each other, and this can be done in various ways: **REST APIs**, **Message Brokers** (For asynchronous communication and event-driven architectures, message brokers such as: **RabbitMQ**: A popular message broker using queues for communication.)

**Api –** Services written on different languages can easily interact with the help of API.

**Cache** can be done with the help of **reverse** **proxy servers**(server side) and browser cache(clientside). When we use cache we should keep a expiration time for it. If we keep expiry time less then it becomes simiar to no caching, but if we keep expiration time more then we get the previous data and not the updated one. Keep this time accordingly, if the data changes more frequently its better to keep the caching expiration time small.



**Message Queue(***combination**of**load balancing, \*\*\*heartbeat check[if a server is working properly], notifier[notify all the failed transactions]* **) -** *is a form of service-to-service communication that facilitates asynchronous communication.*  It functions as a temporary storage and routing system for messages exchanged between different components, applications, or systems within a larger software architecture. There is a producer consumer concept. Multiple consumers can read messages concurrently from the queue. Message queue allow applications to send and receive messages without having to wait for a response. This **decouples** applications from each other, allowing them to be developed independently. Message queues can be scaled to handle large volumes of messages by adding more servers. This makes them ideal for high-traffic applications.

* **Anti-pattern –** if we use database as message queues then it is termed as this. We should avoid this. All the servers poll themselves to the database at specific intervals to check if they have some message for them.  
  Frequent polling – load on DB

**L**ong intervals – inefficient, users will be unsatisfied,

* We need to understand where we need just a DB and where we need specialized message queues. If the system is small or the servers donot need to interact with each other a lot, then anti-pattern can be used

**Pub/ Sub messaging(classic example of event-driven architecture)**

* Disadvantages : Does not ensure atomicity, not good for banking/ critical systems

**Event-driven architecture** (**EDA**) - is a [software architecture](https://en.wikipedia.org/wiki/Software_architecture) paradigm concerning the production and detection of [events](https://en.wikipedia.org/wiki/Event_(computing)). Here services donot interact with each other directly but introduces an event that something has occurred.

In a Pub-Sub model, events are published by producers (publishers) and received by consumers (subscribers) without direct connections between them. Instead, an **event broker** (like Kafka, RabbitMQ, or AWS SNS) sits in the middle, allowing multiple consumers to subscribe to specific event topics. When an event is published to a topic, all services subscribed to that topic receive and react to it.

Messages

A message is communication data sent from sender to receiver. Message data types can be anything from strings to complex objects representing text, video, sensor data, audio, or other digital content.

Topics

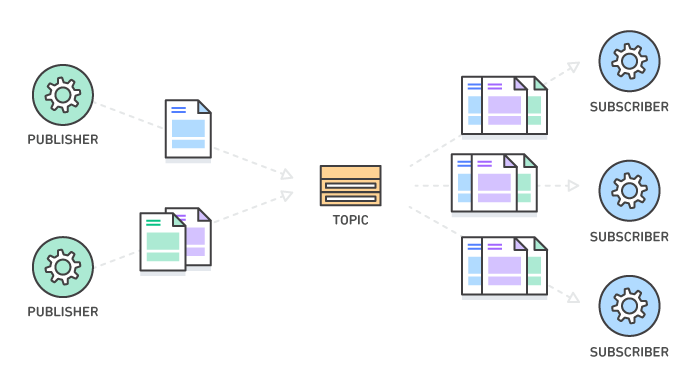
Every message has a topic associated with it. The topic acts like an intermediary channel between senders and receivers. It maintains a list of receivers who are interested in messages about that topic.

Subscribers

A subscriber is the message recipient. Subscribers have to register (or subscribe) to topics of interest. They can perform different functions or do something different with the message in parallel.

Publishers

The publisher is the component that sends messages. It creates messages about a topic and sends them once only to all subscribers of that topic. This interaction between the publisher and subscribers is a one-to-many relationship. The publisher doesn’t need to know who is using the information it is broadcasting, and the subscribers don’t need to know where the message comes from.



## What is the difference between message queues and pub/sub messaging?

A [message queue](https://aws.amazon.com/message-queue/) is another form of asynchronous communication used in serverless and [microservices](https://aws.amazon.com/microservices/) architectures. Messages are stored in the queue until they are processed and deleted. Message queues require the sender to know who they are exchanging messages with. Message ordering may also cause bottlenecks in the system.

In contrast, the publish-subscribe (pub/sub) pattern allows for more flexibility. Several interested subscribers can receive messages simultaneously and asynchronously. Publishers don't need to know who the subscribers are. Message handling is more scalable and reliable, and it gives better performance.

**Performance Metrics**

1. Throughput(should be high) – Work Done/ Total Time

* Ex- Number of API calls served per unit time

1. Bandwidth(should be high) – Amt. of data transfer per unit time

* We need to adjust b/w bandwidth and resources to adjust the throughput. If both are high then throughput is also high.

1. Response Time(less) –
2. Latency - Latency is that annoying delay you experience when trying to access a web page or video stream before it fully loads on your device

We can use **Performance Management** tools to take care of everything

**Fault** is the cause, **Failure** is the effect. A Fault is a state that causes the software to fail and therefore it does not achieve its necessary function. Failure is the accumulation of several defects that ultimately lead to Software failure and results in the loss of information in critical modules thereby making the system unresponsive/ crash.

**Scaling**

Increasing the capacity of an existing resource is **vertical scaling.** This is consistent. Resilient; if one server goes down others can respond.

Increasing number of resources is **horizontal scaling.** Can lead to inconsistency. This has single point of failure.

**Database Replication**

* Saves us from data loss if a DB crash
* Improves latency. Suppose Data is needed in multiple geographical centers then to retrieve data from nearest center\*\*\*(1) will be faster
* Also increase application performance, since several DB can be just kept for reading, etc
* This can lead to **problems** such as data inconsistencies, for the time being when the data is not copied to the replicas. The issue could be resolved using **synchronous replication.** When data consistency is of less importance we can use **asynchronous replication** to increase write speed and thus increasing application speed.

**synchronous replication-** We wait for all the replicas to acknowledge, after successfully completing their write and then the main DB writes/ commits.

**asynchronous replication-** Host is acknowledged after primary DB is updated.

**Hybrid replication-** We can wait just for 1 or n number of acknowledgement before updating the primary table.

**CAP Theorem**

It may seem like if there is **partition**, then the system could be either serve **availability** or **consistency**, but not both.

**Explanation –**

1. Distributed System – is a system consisting of group of machines working together so as to appear a single system to the end user. Ex- local servers present on different cities. If the nearest ones goes down, we can use farther one, atleast it will be able to serve the requests even if late.
2. Consistency – any read that is happening after the latest write should return the value of the latest write
3. Availability – every node in the system should respond in non-error format to any read request without the guarantee of returning latest write
4. Partition Tolerance – system will be responding to all the reads and writes even if the communication channel(middleware) between those nodes are broken(partitioned).
5. CAP theorem tells that atleast one of the features should be sacrificed. It is impossible to have all the three holding together. But Partition is a common thing that might occur anytime in a system, so we need to sacrifice either the consistency or availability according to the system requirements. We can serve all of them by increasing the number of connections between the systems so that if one network fails other can be used. But this will be costly and complicated.

**Data Sharding**

Logical Sharding(one machine having different partitions) and vertical sharding(data on different machines).

* Algorithmic sharding

Uses a consistent algorithm to determine the shard for each piece of data. This method is also known as hashed or key-based sharding. Algorithmic sharding is straightforward and fast, and it can distribute data more evenly across shards. However, it can be inflexible and difficult to add or remove shards.

* Dynamic sharding

Uses an external service(lookup table) to locate the machine that contains the data for each request.

1. Key-based Sharding – Choose a **static** key on which shard will happen

* Choose a hash function that will map a data to a particular shard
* inflexible and difficult to add or remove shards, since we will have to change the hash function and it will be very difficult

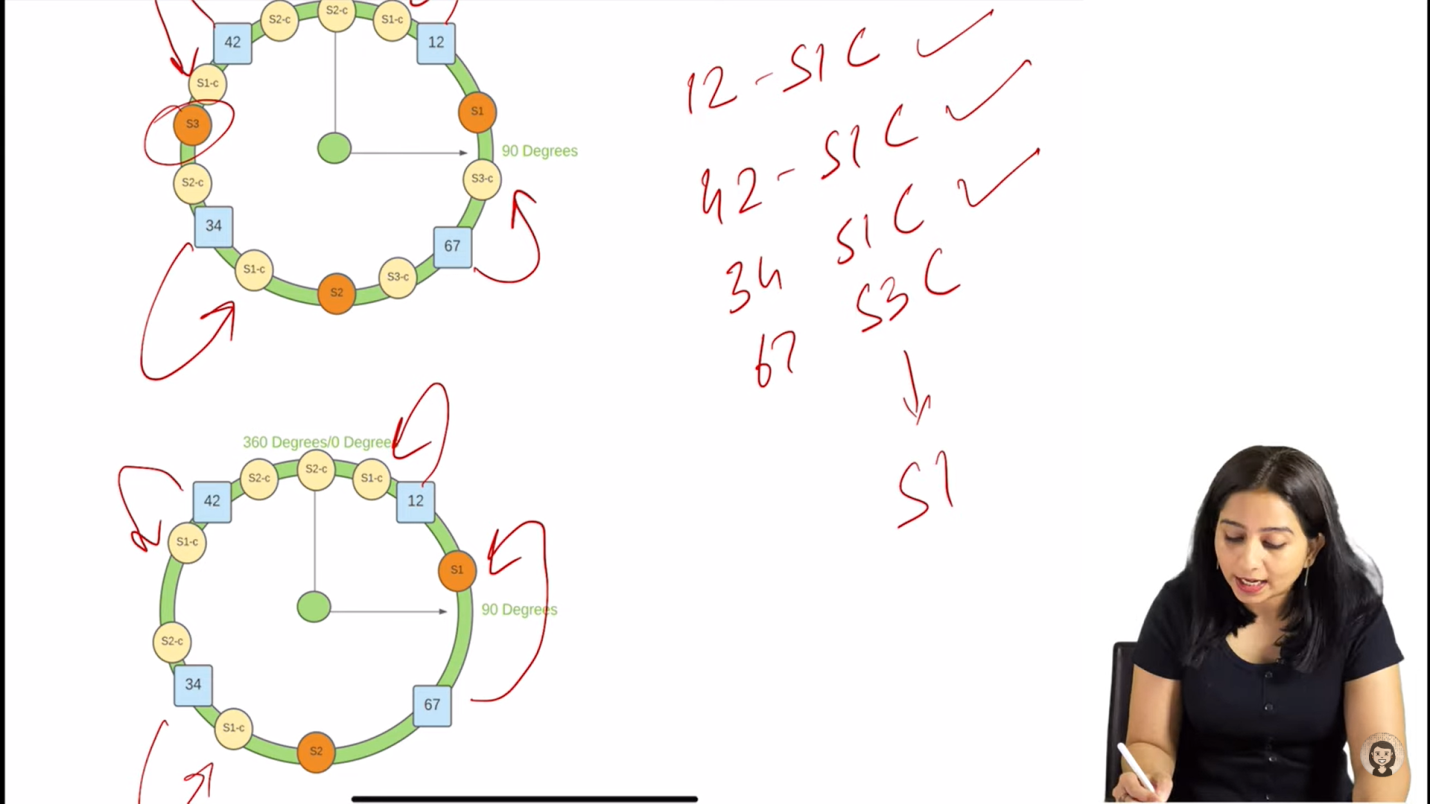
1. Range-based sharding –
2. Directory-based sharding - This flexibility facilitates efficient load balancing and adaptation to changing data patterns.It also introduces a single point of failure if the external locator service fails, if the lookup table fails, entire system will fail.

**Hashing** can be used to distribute servers, but this method is not good for scalable systems as when increasing/ decreasing number of servers the system will need a lot of key change(maybe approx 75%). Solution to this is **consistent hashing.**

**Consistent Hashing-** The values and the servers are mapped within the range of the values that has been already decided, say 1 – 100. Now, we choose a approach to store data in particular server(say clockwise/ anticlockwise). For each data we map it to the nearest(clockwise/ anti-clockwise) server.

**Problem** – When we remove a server all the data goes to a server making it overloaded.

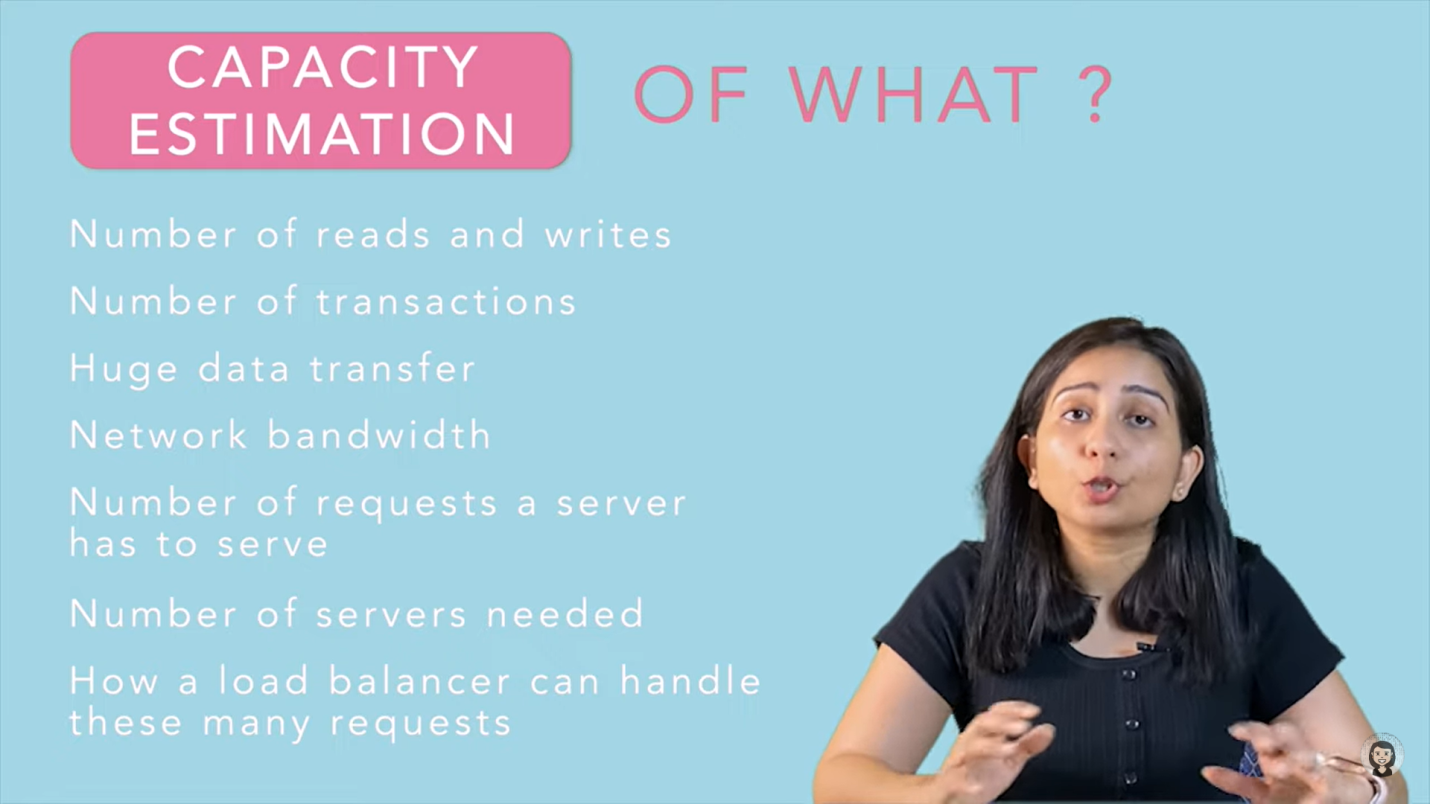
**Solution –** Use replicas of servers

****

**System Design Approach :**

* 1. **Identify functional and non-functional requirements**

****

****

K**ey points for System design**

1. Scaling
2. Preprocessing & cross job(prepare beforehand at non-peak hours)
3. Keep backups to avoid single point of failure
4. First we can keep extra servers,(use load-balancer for this, again just to keep single load-balancer is a single point of failure, so keep multiple load-balancers and to redirect client requests to one of them use DNS), *what if database goes down?*
5. Master slave database, *what if a natural disaster occurs at a place?*
6. Multiple regions
7. Micro-service architecture
8. Distributed System. (If any failure occur at a local area)
9. Load Balancer – this is a mechanism included in gateways, Consistent hashing helps to do load balancing

* It makes sure two things
* Balancing load on each server
* Not sending duplicates

1. Decoupling
2. Logging and metrics calculation(maintaining record of all the events that occurred within the organization)
3. Extensible (so that in future we can increase our business scope).

**Chat application System design**(TCP) – users can interact with each other using

1. HTTP(client-server) protocol, 2. Peer-to-peer protocol, 3. XMPP(**Extensible Messaging and Presence Protocol**) protocol(used in WhatsApp & Zoom)

Why not HTTP – server cannot send message to the client

**\***For client-server communication **WebSocket(TCP) is preferred over HTTP(S) protocol** since HTTP(S) doesn’t keep the connection open for the servers to send frequent data to a client. With HTTP(S) protocol, a client constantly requests updates from the server, which is resource intensive and causes latency. WebSocket maintains a persistent connection between the client and a server. This protocol transfers data to the client immediately whenever it becomes available. It provides a bidirectional connection used as a common solution to send async updates from a server to a client.

[Read from here once](https://medium.com/@m.romaniiuk/system-design-chat-application-1d6fbf21b372)

* Group/one-to-one messaging
* Sent + delivered + read receits
* Online/ last seen
* File sharing
* Chat history

**Capacity planning of any video app(Youtube)-**

1. Find num of active users (1B)
2. Num of users that post videos (1B/1000 = 1M)
3. Video time duration approx(10min)
4. Size of this 10min videos (approx 3MB). Therefore, total size of all users=1M \* 3MB = 107 \* 3MB = 30TB
5. For CDN/ high availability/ lower response time/ fault tolerant there can be more than 1 copy (say 3, Therefore, 30\*3=90TB)
6. Now we have more than 1 resolutions stored, lets take 720,480, 240, 144. If 720 is making 90TB, all of these will make 180TB(somewhere around) since one size is double of other x+ x/2 + x/4 + x/8 = 15x/8 = 2x
7. Now find number of processors required to perform the task
8. 180 TB/ day = 2GB/sec
9. For processing data

Read + Video Processing Time + Write  
10ms + 20ms(assume) + 20ms  
= 50ms/ MB data

1. We need to compute (2000 \* 50 \* 10-3 sec)/ sec  
   Therefore, we need 1000sec / sec which means 1000 processors

\*\*\*(1) A content delivery network (CDN) is a group of geographically distributed servers that speed up the delivery of web content by bringing it closer to where users are.

CDNs rely on a process called “caching” that temporarily stores copies of files in data centers across the globe, allowing you to access internet content from a server near you. Content delivered from a server closest to you reduces page load times and results in a faster, high-performance web experience.

* **Reduce Latency** -  Some content delivery networks alleviate latency by reducing the physical distance that the content needs to travel to reach you.

\*\*\*(2) **Heartbeat –** are periodic signals sent between components of a distributed system to indicate that they are still alive and functioning properly.

Our data pipelines are too slow, we need to process data faster. How?

* We can check if the all the servers are up and running so that other servers are not overloaded and thus getting slow. 🡪 health checkup(heartbeat)
* We can try to restart the servers if they are dead/ not responding.

**Cache**

**Slave DB**

**Database***(Master)*

**Server**

**Server**

**Server**

**Gateway**

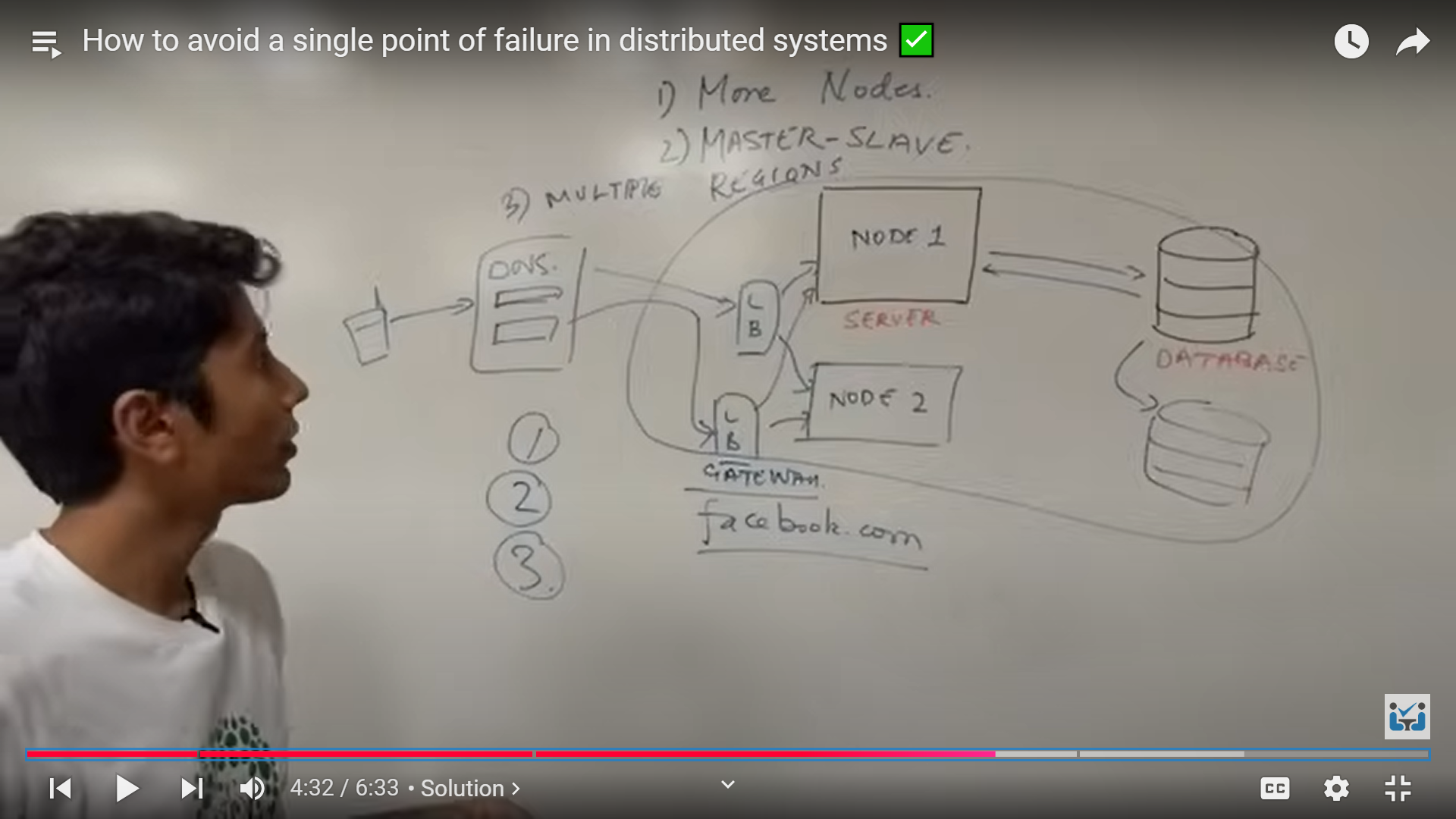
*Contains load balancing technique here*

**CDN**

**Client**

**Client**

**To Avoid Single Point Of Failures**

****

(to learn : Kong, RabbitMQ)