SCALING: VERTICAL & HORIZONTAL

- Let your server is hosted on a cloud, then the business requirement could be:
  - There are a lot of customers are using your server. But Can the server handle this much load?
  - So, the following solutions can be possible
    - Buy Bigger Machine (Vertical Scaling)
    - Buy More Machine (Horizontal Scaling)
  - These things are used to increase the Scalability of your system.

	Horizontal Scaling	Vertical Scaling
1	◦ Load Balancing required	◦ N/A
2	◦ ***Resilient*** (Able to recover readily)	◦ ***Single Point of Failure*** (if the machine goes down, then its gone. As only 1 machine is there)
3	◦ Network Calls (*RPC: Remote Procedure Calls*) ◦ The systems will be communicating with each other through network calls. So, comparatively ***Slower***.	◦ Inter-process communication. So, ***Faster***.
4	◦ Data is ***Inconsistent***. ◦ Data is complicated to maintain. ◦ If a transaction has to be made, the operation has to be atomic. ◦ So, we have to lock all the server which is impractical. So, ***loose transactional guarentee***.	◦ One system is there where the system resides. ◦ So, here, Data is ***Consistent***.
5	◦ It is ***Scalable***. ◦ We can add more system to scale up the system according to the number of customers.	◦ ***Hardware Limitation***. ◦ We can't make the computer bigger and bigger and bigger.

- In real-world, both Horizontal & Vertical Scaling are used.
  - We can take points 2, 5 from Horizontal Scaling & 3, 4 from Vertical scaling (Positive points).

- Initially we'll *Vertical Scale*, and at the point where we can't scale it vertically more, we'll use *Horizontal Scale.*

LOAD BALANCING & CONSISTENT HASHING

- Let you have ***N*** no. Of servers and A user sends one request, "which server that request will go to?"
- All the servers are carrying ***Load*** on it to process the incoming requests.
- Concept of taking ***N*** no. Of servers and trying to balance the load among them ***evenly*** is called **Load Balancing** .
- You'll get a request ID whenever the user sends the request to the server.
  - That request ID is **uniformly random** i.e. ***0*** *..to..* ***M-1*** (uniformly random: probability of getting any thing out of a list is same I.e. suppose an array of numbers is there and you are told to pick one among those numbers)
  - **H(r)** ⟶ **M** (H: Hash, r: request ID)
  - The server to which the request will go is: **M % N** (N: No. Of Servers)
- Let you have 4 servers, S0, S1, S2, S3.
  - H(10) = 3 (let) ; 3 % 4 = 3 ; It'll go to S3
  - H(20) = 15 (let) ; 15 % 4 = 3 ; It'll go to S3
  - H(35) = 12 (let) ; 12 % 4 = 0 ; It'll go to S0
- So, as the request IDs are uniformly random, also the hash function is uniformly random;
  - So, each of the servers will have ***X/N*** **loads if there are** ***X*** **requests in total.**
  - So, the *Load Factor* is ***1/N.***
- **But, what happens if we need to add more servers?**
  - Let 5 servers are there now. S0, S1, S2, S3, S4.
    - H(10) = 3 (let) ; 3 % 5 = 3 ; It'll go to S3
    - H(20) = 15 (let) ; 15 % 5 = 0 ; It'll go to S0
    - H(35) = 12 (let) ; 12 % 5 = 2 ; It'll go to S2
  - So, all the things now changed.
  - We can find the changes via a ***pi*** chart.
    - Initially we had 4 servers. So, the ***pi*** was having 25% keys per server. (Key: Request ID)
    - Servers were having keys S0(0-25), S1(25-50), S2(50-75), S3(75-100).
    - One more server added in the system. So the following would happen now.
      - **S0** : It will be having 0-20 now. Loosed 5%. Change = 5; total=5
      - **S1** :It will get 5 of S0 now. Means 20-50. Gained 5%. Change = 5; total = 10

- Again, S1 will be having 20-40. So loosed 10% i.e. loosed 40-50. Change = 10; total = 20
- **S2**: It'll get 10 of S1 now. Gained 10%. Change = 10; total = 30
  - Again, S2 will be having 40-75. So loosed 15% i.e. loosed 60-75. Change = 15; total = 45
- **S3**: It'll get 15 of S2 now. Gained 15%. Change = 15; total = 60
  - Again, S3 will be having 60-100. So loosed 15% i.e. loosed 80-100. Change = 20; total = 80
- **S4**: (new server): It'll gain 20%. Change = 20; total = 100

- From the above experiment we can see that the change of **Key-to-Server** mapping went to **100%** which is not good.
- So, total is **100%** changes.

- It is a ***Simple Hashing***. **Not *Consistent Hashing***.

➢ **Simple Hashing**

- *Hash(Key) = Val*; *Val % N = Particular server* (Key: Request Id)
- Here, the assignment of the hash to the server is not consistent. Means if the number of servers increases, the key assignments also changes.

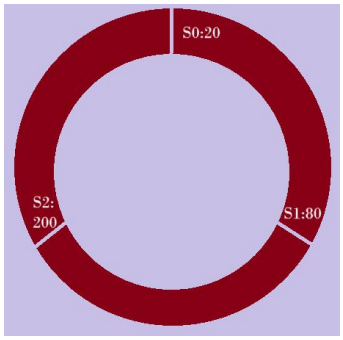
  So, this is not **Consistent Hashing**. Here the change factor can be go up-to **100%**.

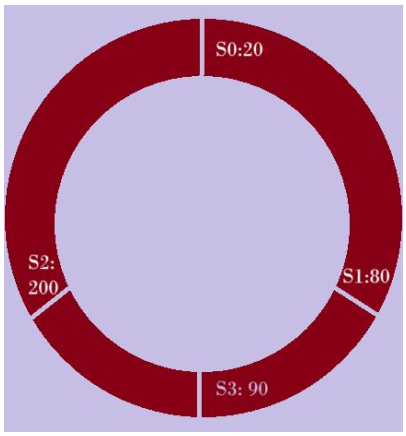
➢ **NOTE**

- In practice, the Request ID (Key) is constant.
- So, if a particular key is mapped to a particular server, then it would be better if we store some frequently accessed data of that particular user as **Cache** rather than accessing the database repeated time.
- But, what in case of Simple Hashing, the Key-to-Server mapping depends upon the number of Servers present in the System.
- So, if we store the frequently accessed data of a particular user in a server, and the number of servers changes, then that cache will be dumped. And will be of no use.

➢ **We can take 5% of each server out of those 4, and assign those to the new 5th server S4. Then the change will be only 20%.**

- **Only 20% of the keys will get affected.**

- **Consistent Hashing:**
  - Technique:
    - Hash the Servers and keep those in a circular ring.
    - Hash each Keys and keep those in that circular ring.
    - Assign each Key to its next Server in *clockwise* direction.
    - **Server's hash >= Key's hash** in **clockwise** direction, will be assigned to the key.
    - Let we have 2 servers; S1:30, S2:40; and the req:30
      - So, it'll go to *S1*, as 30(S1)>=30(req) True.
  - Example:
    - Let we have 3 servers having hash as follows
      - S0 = 20
      - S1 = 80
      - S2 = 200
    - 

      - 21 to 80 will go to S1:80
      - 81 to 200 will go to S2: 200
      - Remaining will go to S0: 20
    - Now, one server was added
      - S3 = 90
    - 

      - 21 to 80 will go to S1:80

- 81 to 90 will go to S3: 90
- 91 to 200 will go to S2: 200
- Remaining will go to S0: 20

- Here, only assignment of server for only 10 keys (81 to 90) got changed.

NOTE

- Let the ring is having capacity 0 to M.
- The user counts are N. It means the hash counts will be N (unique hash)
- If ***N < M***; there is no issue as the user will be anyhow mapped to the server that comes just next to it in clockwise direction.
- What if ***N > M***;
  - Here, ***hash(user)%M*** will be calculated.
  - Consider the following scenario:
    - Let M=20, N=30
    - Let S0:1, S1:10 is there inside the ring.
    - So, users having hash 2 to 10 will be using S1.
    - If a user comes whose hash will be more than 20; let 25
    - Now the value = 25 % 20 = 5
    - So, it'll use S1 now.
    - Its not a fault because, the users (having hash from 2 to 10) were already sharing S1 for their requests.
    - ***Even the modulo of server's hash will be stored.***

In case of *consistent hashing*, the drawback is **server load distribution**.

- S1:10, S2:11, S3:12, S4:90
- Here we can clearly see, S4 is overloaded; handling almost 78% of the key space.
- This is the main drawback of **Basic Consistent Hashing**.
- If we will be able to place the same server at multiple place of the rings, then it'll solve this problem.
- Like S1, S2, S3, S1, S4, S1, S3, S2, S4, S2, S1.. like this (of course in a ring only)
- If we take multiple hash functions.
  - Let we have **k** hash functions
  - So for each server, we'll be having **k** hashes.
  - So, one server can be placed at **k** points in the rings.

- Same for all the servers.
- So, if we have **n** servers, then we'll be having **k*n** server points inside the ring.
- So, here the load distribution will be possible evenly.
- **Now, the *pi* of 4 servers, from which we were taking *5% keys* of each of those 4 servers and assigning those *keys* to the 5^{th} server, is successful**.

➢ Load Balancing is extensively used in Distributed System.

MESSAGE QUEUE

- Pizza shop analogy:
  - Client-1 orders a pizza. All he wants is a confirmation that the order is placed.
  - Now, he got the confirmation that the order is placed.
  - Now, client-2 comes to order a pizza. He doesn't have to wait till pizza of client-1 is ready and his process is completed, rather client-2 also get a confirmation that his order is placed.
  - So, a list has to be maintained to keep track of the orders.
  - Whenever a client comes and place an order, the list order has to be written in the list according to the priority (the user might have an urgency) or complexity (the order might be just a coke).
  - It'll give flexibility to do something or use some other resources of the pizza shop rather than focusing on to the pizza only.
  - Also, the shop manager can also receive the orders of other customers.
  - When the pizza is ready, the pizza shop owner will ask the client to pay. In response of which the client pays the bill and take the pizza out.
  - After this, the process is completed, and this user's order will be removed from that list.
  - It all happened asynchronously.
- Lets take a complex scenario, you have 4 pizza shops. PS0, PS1, PS2, PS3
  - Let the shops got some orders. All are maintaining their lists of orders.
  - Now, if a shop, let PS3, goes down, the list will be gone and you'll not be able to process those orders.
  - Solution is using a database and keeping all those orders data in that database.
  - So, let some orders like
    - O1 > PS1
    - O2 > PS3
    - O3 > PS0
    - O4 > PS1
    - O5 > PS2
    - O6 > PS3
  - Now, PS3 goes down. But the orders data are stored in Database.
  - There will be a **Notifier** available in between the shops and the database.

- The Notifier will be checking in some particular interval of time (let each 15 seconds). As the PS3 is down, Notifier will not get any response from PS3 and will assume it as down.
- Now, the Notifier will query the database and check which orders are in N (Not done) state.
- Then it re-distribute all those orders to the Pizza Shops.
- But now, there might be chance of duplication here. For example
  - Let PS0 is preparing the order O3 already.
  - After the re-distribution, let PS1 got O3.
  - So, now the customer will receive 2 pizzas from PS0 and PS1.
  - So, it'll lead to too many duplication and complication.
  - So, here we can use Load Balancer with the Notifier.
  - One important feature of Load Balancer is to remove the duplication along with balancing the load (Consistence Hashing method).
  - So now, from that ring, all the instances of PS3 will be gone now. So, those orders which had to be processed by PS3 will be assigned to some other shop.
  - And, remaining shop's orders will not be removed. Just they'll get some extra orders which were assigned to the shop PS3.

- **Assignment/Notification + Load Balancing + Heart Beat + Persistence = Message/Task Queue**
- Working of Message Queue:
  - Producer sends a message to the message queue. (producer is not the user, it is the backend/server that puts the request to the message queue)
  - The *message queue* (RabbitMQ, Kafka, Amazon SQS ..etc) holds that message in memory/disc.
  - One or more consumers are waiting (or polling) for new messages.(consumer: background service or worker process. e.g., Node.js script who listen to the queue)
  - A consumer picks up the message, process it, and optionally sends an acknowledgement.
  - The queue removes the message after successful processing (or retires if failed).

MICROSERVICES ARCHITECTURE

- There is a myth that says:
  - Monolithic : All the things run in a single machine.
  - Microservices : All the things run in different machines.
- But, these concepts are wrong.
  - Monolithic : Functionality is in a single code-base and deploys together?
  - Microservices : Functionality is split into small, independent services with separate deployments.
- ***NOTE :*** Even if the different independent services of a microservice code-base are hosted in a same machine, then also this will be called a **Microservices Architecture**.
- Example of a Monolithic Architecture Code-base:
  -
```
ecommerce-monolith/
├── server.js
├── routes/
│   ├── auth.js
│   ├── products.js
│   └── orders.js
├── controllers/
│   ├── authController.js
│   ├── productController.js
│   └── orderController.js
├── models/
│   ├── User.js
│   ├── Product.js
│   └── Order.js
└── utils/
    └── sendEmail.js
```
  - Here, everything are included in a single express app which can be deployed as a whole.
- Example of Microservices Architecture Code-base:
  -
```
ecommerce-microservices/
├── auth-service/
│   ├── index.js
│   └── routes/auth.js
├── product-service/
│   ├── index.js
│   └── routes/products.js
├── order-service/
│   ├── index.js
│   └── routes/orders.js
└── shared/
    └── utils/sendEmail.js
```
  - Here, every services which are independent can be deployed separately. Hence, can be scaled easily.

- Monolithic Architecture:
  - Monolithic Architecture means building the entire application in a single code-base, single deployment.
  - Everything will be bundled together i.e.,
    - Login Logic
    - Business Logic
    - API routes
    - Database access
    - UI (if included)
  - Advantages:
    - Simple to build
    - Easy to debug
    - Quick to deploy
    - Low cost at start
    - Faster prototyping : Ideal for MVPs (early-staged products)
  - Disadvantages:
    - Tight coupling : Changes in one part may effect others
    - Scaling is difficult : You can't scale individual features
    - Long build/deploy time : Full app build even for small changes
    - Harder to test in isolation : Everything is coupled tightly
    - Risky deployments : A small bug can crash the whole app
    - Single point failure
  - For a smaller target, Monolithic Architecture is preferable.
- Microservices Architecture:
  - Microservices Architecture means the whole application is divided into smaller, independent services, each responsible for a specific feature or functionality.
  - Each microservices
    - Runs independently
    - Has its own code-base
    - Can be deployed, scaled, or updated separately
    - ***Communicate*** with other microservices via ***APIs*** or ***message queues***.
    - Connect to its own database *(ideally)*
  - How the microservices architecture works?
    - Lets we have multiple services in our e-commerse application like
      - User service

- Product service
- Order service
- Payment service
- Email service

- Each services will be running on different ports i.e. 4000, 4001, 4002 …etc
- All services connect to their own databases **(ideally)**
- Talks to other services using HTTP APIs or message queues

DATABASE SHARDING

- **DATABASE** vs **DATABASE SERVER**
  - A **database server** can host multiple **databases**.
  - Whenever a client (like an Express app) queries the **database**, it sends the request to the **database server** first. The server then queries the appropriate **database** and returns the result.
  - The client cannot query the **database** directly — it must go through the **database server**.
    - For example, when you create a MongoDB cluster, you are essentially running one or more **mongod processes** (i.e., MongoDB servers) somewhere in the cloud or infrastructure.
  - ***NOTE***
    - You can have multiple **database servers**, each running its own instance of **mongod** (the MongoDB server process).
    - A **database server** is simply a **running instance of mongod** that manages one or more databases.
    - For example:
      - If you run mongod on port 27017 on your local machine, that's your local MongoDB server.
      - If you run mongod inside a cloud cluster (e.g., MongoDB Atlas), that's a remote MongoDB server.
    - In all cases, it's always the same **mongod** binary, but you can run multiple instances of it in different environments (local, cloud, containers, etc.).
    - Each instance is a separate server, and can manage different databases independently.
    -
- Database sharding is nothing but partitioning the database into smaller parts.
- There are 2 types of partitioning of Database:
  - Horizontal Partitioning:
    - splitting the database taking a **key** (column in SQL)
    - Ex:
      - I have a table **User**(id, name, email, age) having **10 million** data
      - I'll take **id** as the key and partition the table like **10 partitions** having **1 million** data each.
      - **Horizontal Partitioning** is called **Database Sharding**.

- Vertical Partitioning:
  - Divide the table into smaller parts making the total number of column less.
  - Ex: User(id, name, email, age) => User_db1(id, name), User_db2(id, email, age)

➢ **Database Sharding:**

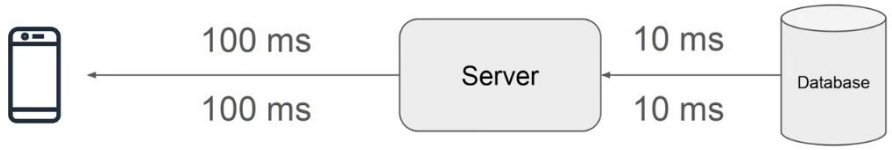
- Let's say you have a **database server** running a **database** called **userDB**.
- Inside this **database**, you have a **collection/table** called **User** with **10 million** entries.
- ***Database sharding*** means:
  - The User **table/collection** will be horizontally partitioned based on a **shard key** (e.g., user ID, location, etc.).
  - For example, if this were a travel or hotel app, hotels might be partitioned by location.The collection is logically split into 10 chunks (e.g., 1 million records each).
- Then, 10 new **database servers** (called **shards**) are introduced.
  - Each shard runs its own instance of the database server (e.g., MySQL, MongoDB, etc.).
  - Each shard stores a separate copy of the userDB **database**, but only contains 1 chunk (i.e., 1 million rows) in the User collection/table.
- The application does not talk directly to the shards.
  - Instead, it connects to a Query Router / Shard Router / Gateway.
  - This router determines which shard holds the data for a given request and forwards the query accordingly.
  - **Simply,** *In sharding, the backend connects to a router which routes queries to the correct shard server, each storing only a partition of the original data***.**
- **Sharding enables horizontal scaling by distributing data across multiple DB servers, while the app remains connected to a single router that hides the complexity.**
- Problems in Database Sharding:
  - In case of Joins, it'll cause problem. For example:
    - Let we have 2 tables User, Order.
    - Both have been sharded using user_id.
    - If we want to Join the tables (User & Order) and if both are on different shard then it'll slow down the query.
- Best Practices:
  - Its better to create index on the shards.

- Lets you have a query to fetch all the users staying in London having age greater than or equal to 50.
  - Your database is sharded taking the location as key.
  - Then, you just need to go to that shard, and just make the query for >= 50.
  - It'll be faster.

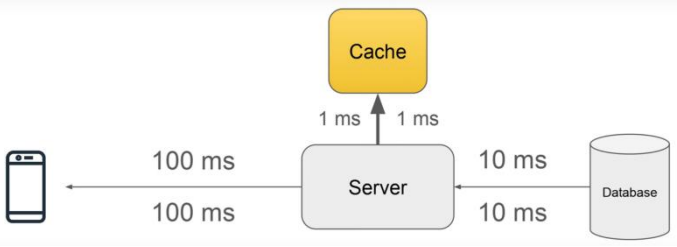
- What if a shard goes down?
  - To ensure availability and fault tolerance, each shard is typically deployed using a Master-Slave (or Primary-Secondary) replication model.
  - The Primary (Master) node is responsible for all write operations and contains the most updated data.
  - Secondary (Slave) nodes replicate the data from the Primary — either in real-time or near real-time.
  - What if the Primary (Master) Goes Down?
    - If the Primary node in a shard becomes unavailable:
    - The remaining replica nodes (Secondaries) hold a leader election.
    - One of them is promoted to become the new Primary.
    - Clients will start routing write operations to the newly elected Primary.
    - This ensures the shard stays available even when one node fails.

CACHING IN DISTRIBUTED SYSTEMS

- Distributed System:
  - It simply means the system is deployed across multiple servers or machines.
  - It doesn't depend on the architecture (whether it's monolithic or microservices).
  - Microservice as Distributed:
    - If all the components/services in a microservices architecture are deployed on **multiple servers**, then it becomes a distributed system.
    - But if all microservices run on the same machine, it is still microservices architecture, but **not a distributed system**.
  - Monolithic as Distributed:
    - To handle high traffic or load, the monolithic system (entire app) is replicated and deployed on multiple servers.
    - This setup also qualifies as a distributed system, even though the architecture is monolithic.



- Here total time starting from user sending request to getting the response: 100 + 10 + 10 + 100 = 220
- Here 2 types of connections are there: **Client-Server**, **Server-Database**
- Let, for an example, we'll take **instagram feed**:
  - Let a user having some particular criteria like he plays football, wants to get the instagram feed.
  - So, we can store this user's feed as **cache** inside the server, and whenever another user having same area of interest comes, we can serve that directly.
  - Caching reduce the latency.
- Further, *we can store some cache inside the client device, if the user is accessing the same thing again and again, which will lead to overall reduction of the latency (preventing that 200ms latency to get response from the server).*



- If the caching is not proper, i.e. if the things that users query for are not present in the cache, then there'll be a waste of some time (that takes to query the cache) every time.
- **Addition of data to cache**, and **removing the data from cache** (if it is full and a new thing to be added) is the most important thing.

- **Drawbacks:**
  - Cache replacement (adding and removing data from cache)
    - Let you are using *Least Recently Used* policy and the user queries for 1,2,3,4,1,2,.. and let the cache size is 3
    - 1, 2, 3 will be added
    - When user query for 4, 1 will be deleted
    - Now when user query for 1 but it is not there in cache.
    - … and so on..
  - Cache & database are unsync:
    - For example like counts on a video
    - We can se the cache to fetch the fresh data (like count) in every 30 seconds interval, user need not to see the exact count as it doesn't matter.
    - It'll improve the performance.
    - But in case of a financial transaction, we can't do this as the user'll be seeing a false data.
    - The cache and db will be in sync, but not in every moment.
  - So, if the cache is not properly maintained, it'll effect negatively on the response time.
- Placement of Cache:
  - The following placements are possible:
    - Cache in database itself.
    - Cache inside the server itself.
    - A particular cache server that can be scaled independently, which will be used to store the cache.
  - In practice, all of the above are used.
- Combining all the points:
  - Cache reduces network calls
  - Avoid repeated computations
  - Reduce database load

SINGLE POINT OF FAILURE IN DISTRIBUTED SYSTEMS

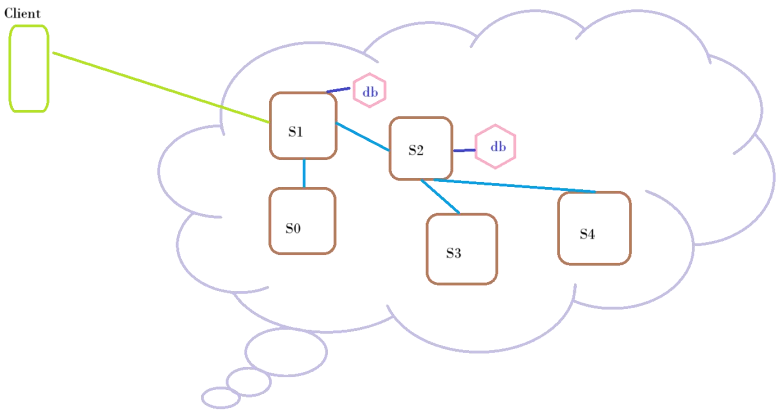
- ***Resiliency***: capability of a system to handle failures gracefully without affecting the overall system availability or performance.
- Let we have a system
  - **Node ----- Database**
    - Here, if the database goes down, all the things will fail.
    - We can use another backup database for this, which will sync with the master database every time.
    - It'll make the system (specially database) more resilient.
    - **Node ---------- (multiple Databases)**
  - But Now the Node can also go down.
    - So, we can use multiple Nodes.
    - But, how the user will go to a particular node?
    - We'll have to use **load balancer** for this.
    - **LB ---------- (multiple Nodes) ---------- (multiple Databases)**
  - **Load Balancer** itself is a single point of failure.
    - To prevent this, we'll have to use multiple **Load Balancers**.
    - But, again same question.. How the user will access the particular **LB**?
    - So, we have to maintain a **DNS** now, which will have multiple gateway IPs (IP will lead to the Load Balancer).
    - **DNS**(multiple ips) **----- (multiple LBs) ------ (multiple Nodes) ------ (multiple Databases)**
  - But now, what if due to any disaster the whole system goes down?
    - So, for this we need to deploy our system on different **geographical regions**.
  - You can also use a **coordinator** in between the **nodes** and **databases** to connect to the database maintaining the load.
- So, the key points are:
  - Multiple Nodes
  - Master-Slave
  - Regions

CDN (Content Delivery Network)

- ***Resiliency***: capability of a system to handle failures gracefully without affecting the overall system availability or performance.
- Lets take the following example:
  - The server is in India.
  - For Indians, it'll be faster than some other country like US, Austria.
  - Here, CDN comes into play.
    - CDN doesn't mean the whole copy of a server hosted in different regions.
    - It's a **Cache Server** which stores static files like images, html pages, videos, js, css.
    - It doesn't store application logic, databases, middlewares datas.
  - Some content might be relevant only for US that may not be relevant to Austria.
    - In this case, Both US and Austria can have their separate CDNs, which can be used by them only.
- CDN providers like Cloudflare or CloudFront provides the following services:
  - HTTPS support
  - DDoS protection
  - Web application firewall
  - IP filtering, Rate limiting etc.

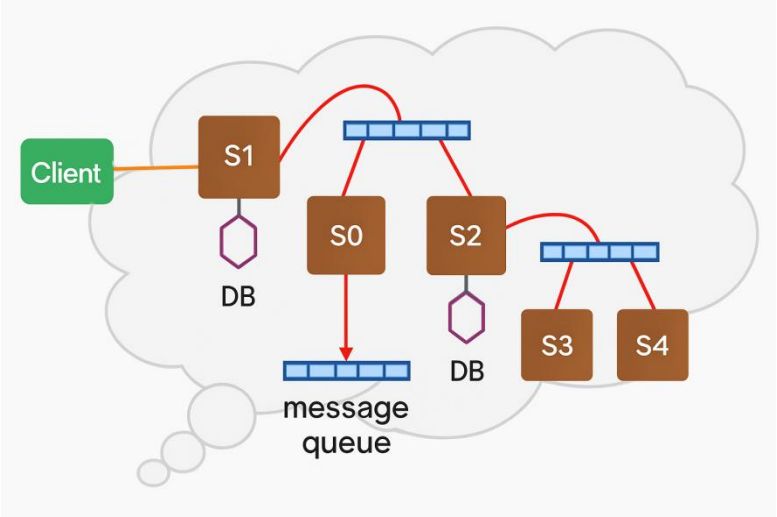
PUBLISHER-SUBSCRIBER MODEL (Pub/Sub Model)

- Pub/Sub model is a common messaging pattern where components communicate asynchronously through messages without knowing each other directly.
- For example:
  - Publisher : A weather sensor publishing temperature data.
  - Subscriber : If anyone is interested/subscribed to that, he can get the weather updates
- Components in Pub/Sub model:
  - Publisher
    - The one which sends (publishes) messages or events.
    - It doesn't care who'll receive them.
  - Subscriber
    - The one who registers interest in certain type of messages/events.
    - Receives message whenever something relevant is published.
  - Message-broker:
    - Middleware that routes messages from Publishers to the right Subscribers.
- **[Publisher] ---------- [Message Broker] ---------- [Subscriber]**
- Example with ***Kafka*** (common in backend system)
  - Order Service publishes **order_created** event.
  - Email Service subscribes to **order_created** to send confirmation emails.
  - Inventory Service subscribes to the same event to reduce item count.
- Advantages:
  - Loose coupling
  - Scalability
  - Flexibility



- (Request-Response model)

- In the current design,
  - S1 waits for S0 and S2, and S2 waits for S3 and S4.
  - If S4 fails, the whole chain waits until timeout before returning failure to the client.
  - This slow failure notification is bad for user experience.
  - Databases of S1 and S2 may already be updated, so a retry could cause duplicate writes unless you have idempotency checks.
- Those are exactly the drawbacks of synchronous, tightly coupled service calls.
- This architecture is **request-driven**, not **event-driven** — meaning each service is directly invoking another instead of reacting to events asynchronously. That's why the failure cascades through the chain instead of being isolated.



➢
- Client → S1
  - The client sends a request to S1.
  - S1 processes it and publishes events into a **message broker (queue)**.
  - Once the event is in the queue, S1 immediately responds to the client (success acknowledgment), without waiting for the downstream services.
- Message Queue for S1
  - Subscribers S0 and S2 are listening to the queue.
  - When an event appears, both S0 and S2 consume it independently.
- S2 → S3 & S4
  - Just like S1, S2 also publishes events into another message queue.
  - S3 and S4 subscribe to that queue.
  - Each will process the event separately.
- Retry Mechanism

- If S4 fails, the message queue keeps retrying until the operation succeeds (or until a configured retry limit / dead-letter queue is reached).
- Once S4 processes successfully, the event is removed from the queue.

- Key Points About This Design:
  - **Asynchronous Processing** → The client doesn't wait for S0, S2, S3, or S4 to finish.
  - **Loose Coupling** → Services don't directly call each other; they communicate via queues.
  - **Reliability** → If a subscriber is down, the queue holds messages until it's back online.
  - **Retry Logic** → Failed deliveries are retried without affecting other services.
  - **Scalability** → Adding new subscribers doesn't affect existing services.
- **Drawback:**
  - Ignore S3 and S4 for now.
  - Let this is a transaction system, where to transfer the money the bank takes commission of 50 rupees.
  - Let S0 is the service to cut the commission and S2 is the service to transfer the money.
  - Now, let someone, having account balance of **1000**, sends 950 rupees to a person.
  - Now, it'll come to S1 and then it'll publish that to the message broker.
  - Now, S0 and S2 should consume this. But let S2 is down.
  - So, S1 will deduct the commission amount but S2 has not been up yet.
  - So, account balance now: 950
  - Now, the user again sent 800 rupees (As the transaction failed)
  - Now, again S0 will deduct 50. (Remaining: 900)
  - Let S2 is now up, it'll first check the first consume (950 transaction).. It'll fail because user is not having enough amount now.
  - Then it'll check for 800 rupees, then it transfers.
  - So, here even if the transaction failed, the commission was deducted.
-

EVENT-DRIVEN ARCHITECTURE (EDA)

- Pub/Sub is a technology/pattern, and Event-Driven Architecture is a bigger design that often uses it (along with other tools like queues, streams, and event stores).
- It can use:
  - Pub/Sub to fan out events
  - Message Queues to process jobs reliably
  - Event Streams for history/replay
  - Event Stores for audit
  - Event Sourcing, CQRS, etc.
- Real World Analogy:
  - Pub/Sub : Group Whatsapp Message
  - EDA : A full system where people react to messages; i.e. one replies, one book a cab, another calls a friend .. like this.
-

```
class EventBus {
  constructor() {
    this.subscribers = {};
  }

  subscribe(eventType, callback) {
    if (!this.subscribers[eventType]) {
      this.subscribers[eventType] = [];
    }
    this.subscribers[eventType].push(callback);
  }

  publish(eventType, data) {
    if (this.subscribers[eventType]) {
      this.subscribers[eventType].forEach(callback => callback(data));
    }
  }
}

module.exports = new EventBus();
```

  - It's a JS example of **Pub/Sub model**.
  - Here the subscribe function takes 2 args; eventType & callback

- It simply means, for one particular eventType, which callback function(s) is to be used.
- So, whenever a event of a particular eventType is emitted from the subscriber, all the callback will have to be executed.

➢