# SYSTEM DESIGN TOPICS PART 3


| | | |
|---|---|---|
| 1 | Introduction to DevOps | 1 |
| 2 | Single point of failure | 5 |
| 3 | Virtualization and Containerization | 9 |
| 4 | Service Discovery and Health Checks | 18 |
| 5 | Cascading Failure | 21 |
| 6 | Rate Limiting | 24 |
| 7 | Auto,Pre - Scaling and Batch Processing | 30 |
| 8 | Caching , Gradual Deployment and Smart Coupling | 37 |
| 9 | Anomaly Detection in Distributed Systems | 53 |
| 10 | Distributed Caching , Cache Policies and Global Cache | 61 |
| | | |

# DevOps

**Dev** = Development (writing code)
**Ops** = Operations (running code on servers)

So, **DevOps is a way of working where developers and operations teams collaborate closely to build, test, and release software faster and more reliably.**

In the past, developers wrote code and "threw it over the wall" to operations teams, who then had to figure out how to run it. This caused delays, bugs, and a lot of frustration.

**DevOps solves this by improving:**

- **Speed** – Faster releases of software
- **Stability** – Fewer bugs in production
- **Collaboration** – Teams working together, not in silos

## Key Principles of DevOps:

1. **Collaboration**
   Developers and operations teams work closely together, breaking down silos to share responsibilities and goals throughout the software lifecycle.

2. **Automation**
   Repetitive tasks like building, testing, and deploying code are automated to save time, reduce errors, and ensure consistency.

3. **Continuous Integration (CI)**
   Code changes are merged frequently and automatically tested to detect and fix bugs early in the development process.

4. **Continuous Delivery/Deployment (CD)**
   Software is automatically prepared for release (delivery) or directly deployed to users (deployment) in small, frequent updates.

5. **Monitoring and Feedback**
   Applications and infrastructure are continuously monitored, and real-time feedback is used to improve performance and user experience.

6. **Infrastructure as Code (IaC)**
   Servers and systems are managed using code, making infrastructure setup reproducible, version-controlled, and easy to scale.

7. **Security (DevSecOps)**
   Security is integrated early and throughout the DevOps process, not just at the end, ensuring safer and more compliant software.

## Example in Real Life

Imagine you're building a website like an online store:

- The **developer** writes code for the shopping cart.
- Using DevOps:
    - That code is automatically tested (CI).
    - It's automatically pushed to the live website if all tests pass (CD).
    - The system is monitored, and alerts are set in case something breaks.

This could all happen in **minutes**, instead of **days or weeks** like before.

## Common DevOps Tools

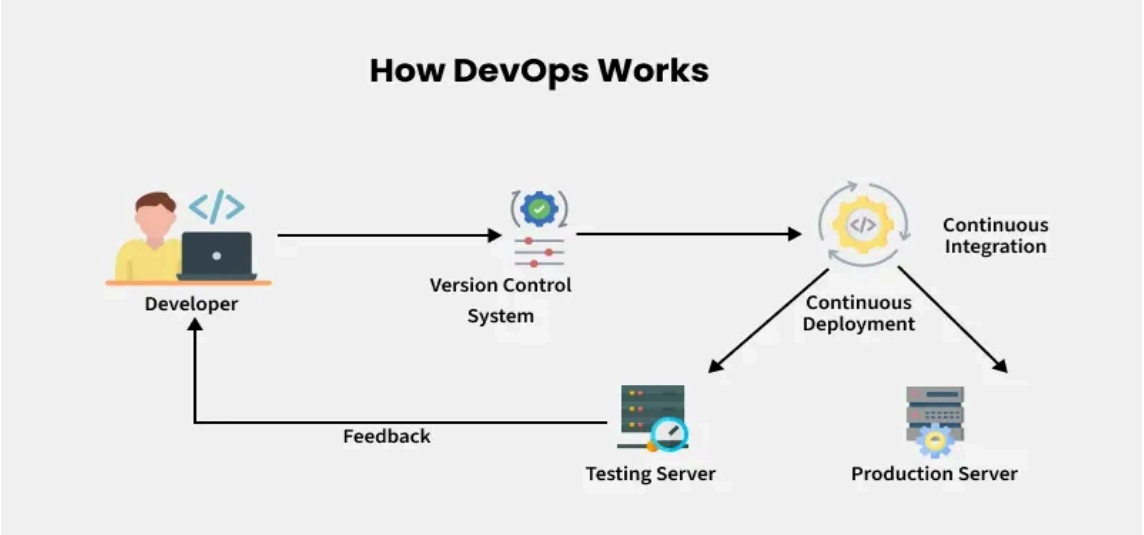
- **Version Control**: Git, GitHub
- **CI/CD**: Jenkins, GitLab CI, CircleCI
- **Containerization**: Docker
- **Infrastructure as Code**: Terraform, Ansible
- **Monitoring**: Prometheus, Grafana, Datadog

(Version control is a system that helps you track and manage changes to your code or files over time.)

(Containerization is a method of packaging software and everything it needs to run — like code, libraries, and settings — into a single, lightweight unit called a *container*.
Normally, software works on your computer but **fails on another** because of different environments (like missing libraries, different settings, etc.).

**Containerization solves this** by:

- Bundling the software **with all its dependencies**
- Making sure it runs **exactly the same** on any machine )



Socho ek software company hai jahan Rahul ek developer hai. Uska kaam hai naye features banana aur existing bugs fix karna. Aaj Rahul ne ek naya feature banaya — maan lo ek “Dark Mode” option app ke liye. Jab uska coding complete ho gaya, to usne apna code **Git** jaise **Version Control System (VCS)** me push kiya. Is system ka kaam hai saare developers ke code ko safely store karna aur track rakhna ki kisne kya change kiya.

Ab jaise hi Rahul ne code commit kiya, **Continuous Integration (CI)** process automatically shuru ho gayi. CI ka matlab hai ki system turant Rahul ke code ko **build** karta hai (yaani executable form banata hai) aur **automated tests** run karta hai. Tests ka kaam hai check karna ki naye code se purana system to nahi toota, koi bug to nahi aayi. Agar sab tests pass ho jaate hain, to CI ka step successful mana jaata hai.

Uske baad aata hai **Continuous Deployment (CD)**. CD ka kaam hai code ko **Testing Environment** me bhejna — yaani ek dummy server jahan real users nahi hote. Yahan pe QA (Quality Assurance) team ya automation tools fir se system ko test karte hain: kya feature sahi kaam kar raha hai, koi performance issue to nahi, aur security theek hai ya nahi. Agar sab kuch OK hota hai, to same code **automatically Production Server pe deploy** ho jaata hai — jahan se real users us feature ko use karte hain.

Jaise hi feature live hota hai, users usko use karte hain aur feedback dete hain: “Dark mode awesome hai” ya “thoda bug hai night theme me”. Ye feedback **Developer (Rahul)** tak wapas aata hai. Wo us feedback ke basis pe code ko improve karta hai. Fir se woh code VCS me push karta hai, aur **poora DevOps cycle dobara shuru ho jaata hai** — automate testing, deployment, aur delivery tak.

## Common DevOps Tools & Their Purpose

### Version Control

Tools that track code changes and manage collaboration.

- **Git** – Tracks code history and changes
- **GitHub / GitLab / Bitbucket** – Platforms to host Git repositories and collaborate with teams

### Continuous Integration / Continuous Deployment (CI/CD)

Tools that automatically build, test, and deploy your code.

- **Jenkins** – Automates build, test, and deployment steps (very flexible)
- **GitHub Actions** – CI/CD built into GitHub
- **GitLab CI/CD** – Integrated CI/CD with GitLab
- **CircleCI / Travis CI** – Cloud-based CI/CD tools for automated workflows
- **ArgoCD / Spinnaker** – Tools for advanced deployment in Kubernetes environments

### Containerization

Tools that package your app and its dependencies into lightweight, portable units.

- **Docker** – The most popular tool for creating and running containers
- **Podman** – Similar to Docker but daemonless (no background service)

## Container Orchestration

Tools to manage and scale multiple containers in production.

- **Kubernetes** – Automatically deploys, manages, and scales containers
- **Docker Swarm** – Lightweight alternative to Kubernetes (less used now)

## Infrastructure as Code (IaC)

Tools to define and manage infrastructure using code.

- **Terraform** – Declare infrastructure (servers, networks) using simple code
- **Ansible** – Automates configuration and app deployment
- **Pulumi** – IaC using real programming languages (JavaScript, Python, etc.)
- **Chef / Puppet** – Older but still used for configuration management

## Monitoring & Logging

Tools to track performance, health, and errors in your app and servers.

- **Prometheus** – Monitors systems and applications (great with Kubernetes)
- **Grafana** – Visualizes data from Prometheus and others
- **Datadog** – Full-stack observability tool (monitoring, logging, APM)
- **ELK Stack (Elasticsearch, Logstash, Kibana)** – Collects and analyzes logs
- **New Relic / Splunk** – Enterprise tools for app performance monitoring

## Security (DevSecOps)

Tools to integrate security into DevOps workflows.

- **Snyk** – Scans for security vulnerabilities in dependencies
- **Aqua / Prisma Cloud** – Container security tools
- **OWASP ZAP** – Open-source tool to scan for web app vulnerabilities

## Collaboration & Communication

Helps teams communicate and manage workflows.

- **Slack / Microsoft Teams** – Team communication
- **Jira / Trello** – Project and issue tracking
- **Confluence** – Documentation and team knowledge sharing

EOT

## Single Point of Failure

A **single point of failure (SPOF)** in a distributed system is any component whose failure can lead to the collapse or significant degradation of the entire system. In simpler terms, it's a critical part of your system that, if it fails, takes down a major or even the whole service.

### Common Examples of Single Points of Failure

1. **Centralized Databases:**
   If your entire system depends on one database, and that database goes down, the application may become unavailable.

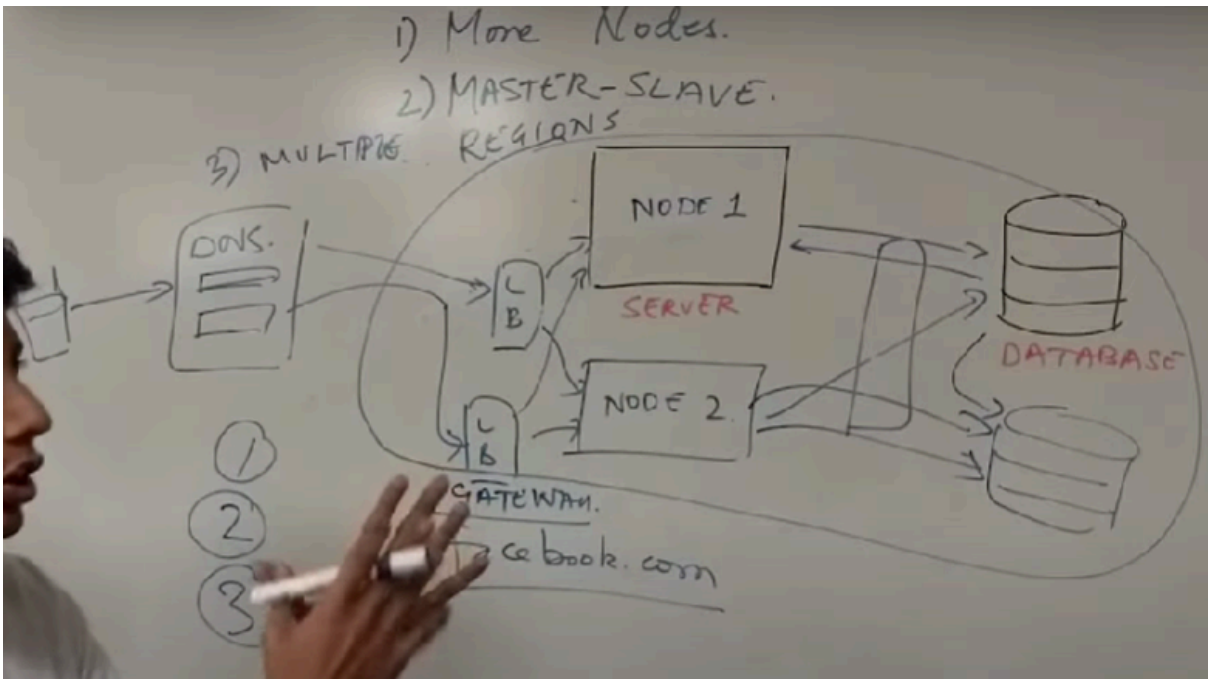
2. **Load Balancers:**
   A single load balancer without redundancy can be a SPOF since it directs traffic across your servers.

3. **DNS Servers:**
   Relying on one DNS provider or instance can be problematic if that server fails, causing users to be unable to resolve your domain name.

4. **Application Servers or Microservices:**
   If a key microservice is centralized and lacks replication, its downtime can affect dependent services.

5. **Network Components:**
   Critical routers, switches, or firewalls that are not duplicated can become SPOFs.

6. **Configuration Services:**
   Centralized configuration management services that multiple nodes depend on, if not replicated, may create vulnerabilities.




## Step 1: A user wants to open Facebook.

They type **facebook.com** into their browser.

But how does your computer know where Facebook is?

That's where the **DNS (Domain Name System)** comes in.
It acts like a **phonebook**, converting "facebook.com" into a real IP address — like `123.45.67.89` — where the server lives.

**If DNS fails and there's no backup**, users can't even *reach* your app.
**DNS can be a Single Point of Failure (SPOF)**.

## Step 2: The user request reaches the Load Balancer.

The DNS points to a **Load Balancer (LB)** — think of it like a **traffic controller**.

- Its job is to decide which **server (Node 1 or Node 2)** should handle this user.
- Maybe Node 1 is free, Node 2 is busy — LB handles it smartly.

**If you only have one LB and it goes down**, nobody can reach your app.
So you create **two LBs** — one takes over if the other fails.

This is **removing the SPOF**.

## Step 3: The request goes to a server (Node 1 or Node 2)

Now the user is being served by **Node 1** (a running app instance).
These are the **application servers** — they process user requests like login, post, comments, etc.

You create **multiple nodes (Node 1, Node 2...)** so that:

- If one fails, the others still serve users.
  **No SPOF here**.

## Step 4: The server talks to a Database

The server needs to get the user's info, so it calls the **Database**.

You don't keep just one database — you use **Master-Slave** or **Primary-Replica** setup.

- Master: accepts writes
- Replica: copies data and can serve reads

If the master fails and you don't have a backup, the app can't save new data.

So we use **failover** — replicas can promote themselves as new master.
**Backups remove SPOFs in data layer**.

## Step 5: You spread all of this across multiple regions

What if one city's data center gets hit by a power outage?

You don't want Facebook to go down worldwide.

So you put **Node 1, Node 2, databases, load balancers — all duplicated** across **multiple regions**.

## General Ways to Avoid SPOFs

- **Redundancy**: Duplicate every critical component (hardware, services, etc.).
- **Failover Mechanisms**: Automatic switch to backup systems.
- **Health Checks & Monitoring**: Detect and respond to failures quickly.
- **Decentralization**: Avoid single control points (e.g., leader election algorithms like Raft).
- **Chaos Testing**: Intentionally break parts of the system to ensure fault tolerance (e.g., Netflix's Chaos Monkey).

Now generalizing ,

**Common SPOFs in Distributed Systems & Their Solutions**

| SPOF | Description | Solution |
|---|---|---|
| **Centralized Database** | One database for all services — if it crashes, everything fails. | • Use **replication** (master-slave or master-master)<br>• Use **distributed databases** (e.g., Cassandra, CockroachDB) |
| **Single Load Balancer** | If the load balancer fails, requests can't reach servers. | • Use **redundant load balancers**<br>• Deploy in **active-passive** or **active-active** mode |
| **Single Application Server** | All traffic goes to one server — a crash halts the app. | • Use **horizontal scaling**<br>• Deploy multiple instances behind load balancers |
| **DNS Provider** | If the DNS goes down, users can't resolve your domain. | • Use **multi-DNS providers**<br>• Implement **DNS failover** |
| **Configuration Server** | System components rely on one config server. | • Use **replicated config servers**<br>• Use **caching** to retain last known config |
| **Authentication Service** | Central auth system goes down → no one can log in. | • **Replicate auth services**<br>• **Graceful fallback** or **token caching** |
| **Cloud Region or Data Center** | Outage in one region affects the whole app. | • Use **multi-region** or **multi-zone** deployment<br>• Implement **geo-redundancy** |

(**Distributed Database** is a database that runs on multiple machines (nodes), but appears to users and applications as a single database.)
(**Active-Passive**: One node handles traffic while the other stays on standby and takes over if the active one fails.
**Active-Active**: Multiple nodes handle traffic simultaneously, providing higher availability and load distribution.)
(**DNS failover** is a technique that automatically redirects traffic to a backup server or location if the primary server becomes unreachable.)
(**Config servers** are special tools that store app settings (like passwords or URLs) in one place, so all services can use the same up-to-date info.)
(**Graceful Fallback**The app checks if a service (like login or search) is working.
If not, it shows a backup message or feature instead of crashing.)
**(Token Caching** When a user logs in, the app saves the token (login proof) temporarily.
If the login server goes down, the app still uses the saved token to keep the user logged in.)

EOT

## Virtualization and Containers

Virtualization is a technology that lets you run multiple operating systems on a single physical machine.

Imagine one big computer pretending to be several smaller computers, each running its own operating system.

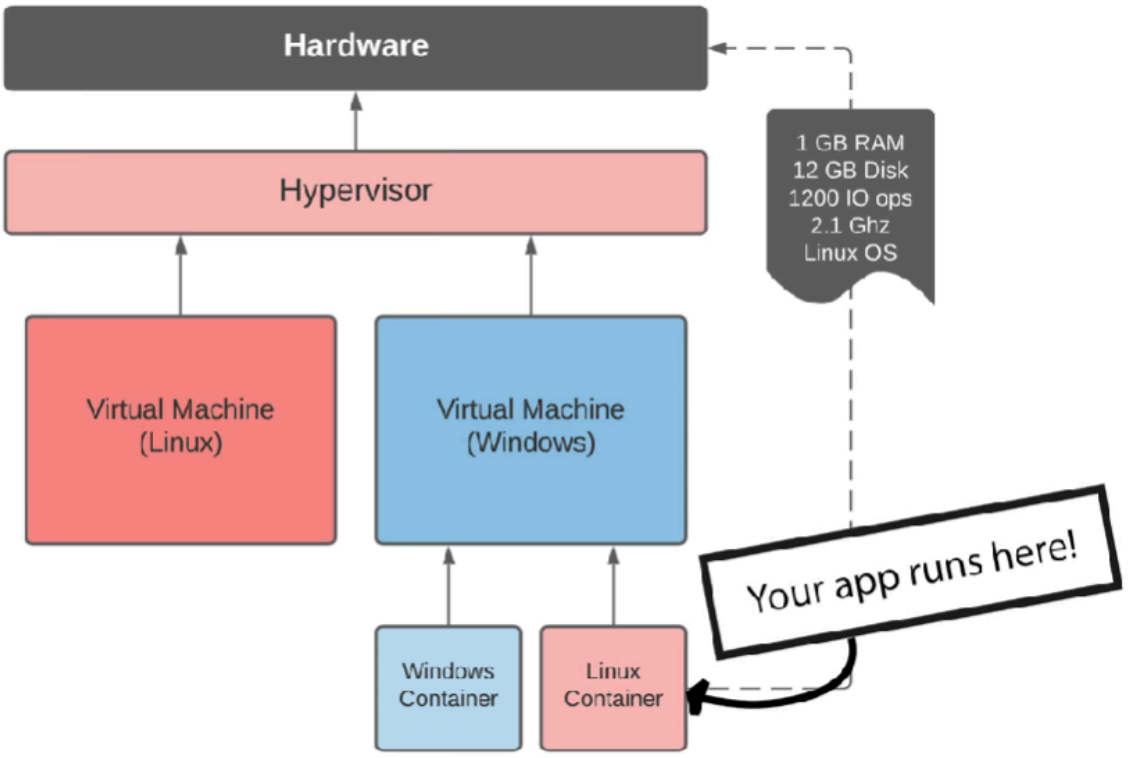
Virtualization is a technology that creates virtual versions of computer resources such as hardware platforms, operating systems, storage devices, and network resources. It's like creating a software-based replica of a physical machine, allowing you to run multiple isolated environments on the same hardware or across a distributed system.

- Imagine you have a powerful computer but you only use a small portion of its resources.
- Virtualization allows you to split that computer into several virtual machines (VMs), each acting like a separate computer with its operating system and applications.
- Each virtual machine is isolated from the others, meaning issues in one virtual machine won't affect others.
- This allows you to optimize resource utilization, run multiple applications on a single machine, and improve scalability by easily adding or removing virtual machines as needed.

**How it works:**

- A special software called a Hypervisor sits between the hardware and the virtual machines (VMs).

- The Hypervisor divides the hardware (CPU, RAM, Disk) among multiple Virtual Machines (VMs).




## What is a Hypervisor?

A **Hypervisor** is software that lets you run **virtual machines (VMs)** on a physical system. It acts like a **manager** between your real hardware (CPU, RAM, disk) and the virtual machines.

### How Hypervisor Works:

- It **splits** your system's physical resources (RAM, CPU, etc.) and gives each VM a **portion**.
- It also **isolates** each VM, so they don't interfere with each other.
- You can run **multiple OSes** (Windows, Linux, etc.) on the same machine using VMs.

### Two Types of Hypervisors:

| Type | Name | Where it runs |
|---|---|---|
| Type 1 (Bare Metal) | VMware ESXi, Microsoft Hyper-V | Runs **directly on hardware** (no OS below) |
| Type 2 (Hosted) | VirtualBox, VMware Workstation | Runs **on top of your OS** (like an app) |

## What is a Virtual Machine (VM)?

A **VM** is a **virtual computer** created by the hypervisor. It behaves like a real computer, but runs **inside your actual computer**.

Each VM has:

- Its **own OS** (e.g., Linux, Windows)
- **Virtual CPU, RAM, storage**
- Its own apps and files

You can install and use software in a VM just like on a real machine.

## Memory Limits

Each VM uses memory (RAM) that's **reserved from your host system**.

- If your system has **8 GB RAM**, and you give:
    - VM1 = 2 GB
    - VM2 = 2 GB
      You only have **4 GB left** for your host system.

That's why running many VMs can slow things down — they **share** the same cake .

### Inside the VM:

- You save files like you normally would — e.g., `/home/user/file.txt` in a Linux VM.

### But physically?

- Those files are saved **inside a big file** on your host system's disk — usually with `.vdi`, `.vmdk`, or `.qcow2` extension.
- This file is like a **virtual hard drive**.

### Can you access those VM files from your host system?

- Normally, **no** — the VM's files are **separate and isolated**.
- But you can set up **"shared folders"** so your host and VM can exchange files.
- Some tools (like VirtualBox) let you open the VM's disk image if needed.

## What Problems Does Virtualization Solve?

### 1. Too Many Physical Machines Needed

Before: 1 app = 1 computer = lots of wasted space & cost
With virtualization: Run many apps/OSes on 1 machine as **Virtual Machines (VMs)**

### 2. Hard to Test or Try Things

Before: Need separate PC to test Linux or an old version
Now: Just **create a VM**, test, delete or roll back safely

### 3. Hardware Failures = Big Trouble

If your PC dies, your app or data is gone
Now: VMs are just files → move or restart them easily on another PC

### 4. Can't Run Different OSes Together

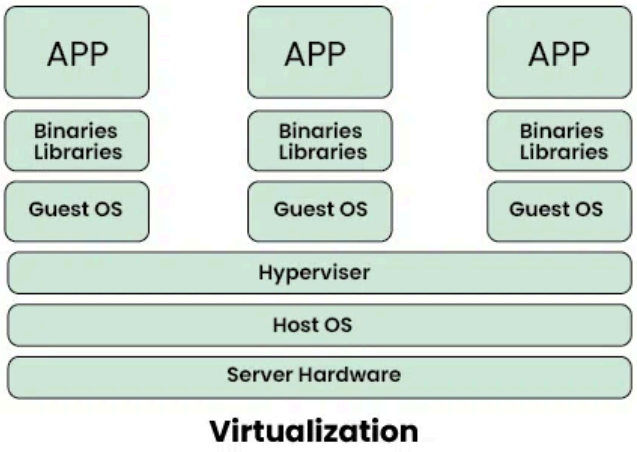
Windows & Linux can't run on one machine
Virtualization lets you **run both at the same time** in VMs

### 5. Apps Can Crash Each Other

One bad app can bring down the whole system
VMs are **isolated** — if one breaks, others are safe

### 6. Upgrading is a Pain

Upgrading RAM/CPU on a real PC = hard
VMs let you **change settings easily** (e.g., more RAM with a click)



**Virtualization**

Now let'sss Move on with Containerization

**Containerization** is a technology that allows you to **package an application with everything it needs** (code, libraries, tools) into a **single lightweight unit** called a **container**.

You can then run that container **anywhere** — on any system — and it will behave the same.

Containerization is a lightweight form of virtualization that allows you to run applications and their dependencies in isolated containers. Each container shares the same operating system kernel but is isolated from other containers, providing a portable and consistent runtime environment for applications.

- Containers provide process isolation, ensuring that applications running in one container do not affect applications running in other containers.
- Containers encapsulate all dependencies and configuration required to run an application, making them portable across different environments.
- Containers are lightweight compared to traditional virtual machines (VMs) because they share the host operating system kernel.
- Containers are designed to be scalable, allowing you to quickly scale up or down based on demand.
- Containers enable developers to build, test, and deploy applications more efficiently, leading to faster release cycles and improved collaboration between development and operations teams.

mtlb jese vm alg alg os k lekr ek system m chl skta hai

container sme os hi lekr alg alg system jese ek pr chlta hai

Mtlb container ko alg alg environment chahiye vm ko alg alg os + env smjhe Evironment mtlb jo path setup krte hai hum humesha ape sytem p aur chaotic ho jaata hai dusri cheez k liye dusra smjhrhe ho

Kuchh Examples :

### 1. Companies like Netflix, Amazon, Flipkart

- **Use case**: Har feature (login, payments, search, etc.) ek **alag container** mein hota hai.
- Isse har feature ko **alag update, test, scale** kar sakte hain — bina pura system rukaye.

### 2. Developers' Laptops

- **Use case**: Developer ek hi laptop par **multiple apps** run karta hai (ek Python, ek Node.js).
- Har app ko **container** mein rakhne se version conflicts nahi hote. Easy testing!

### 3. Software Testing

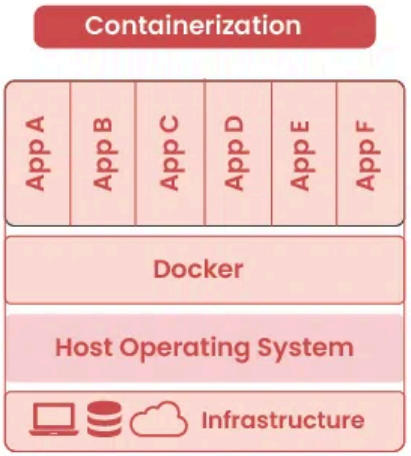
- **Use case**: Testers ko alag OS ya setup chahiye hota hai.
- Containers se woh turant **test environment** bana kar use kar lete hain, bina system kharab kiye.

### 4. Schools/Colleges for Coding Labs

- **Use case**: 100 students ko alag setup chahiye (Python, Java, DB).
- Container images banake sabko **same environment instantly** mil jata hai — koi setup ka jhanjhat nahi.

### 5. CI/CD Pipelines (Automatic Build/Deploy)

- **Use case**: Code push hote hi **container create hota hai**, test hota hai, aur deploy bhi.
- Fast delivery, less errors — har company jo DevOps use karti hai, woh **containers zaroor use karti hai**.




**Virtualization vs Containerization**

| Feature | Virtualization (VMs) | Containerization (Containers) |
|---|---|---|
| **Definition** | Running multiple **OS instances** on one machine | Running multiple **apps** in isolated environments on the same OS |
| **Technology Used** | **Hypervisor** (e.g., VirtualBox, VMware) | **Container engine** (e.g., Docker, containerd) |
| **Components** | Each VM = Full OS + App + Dependencies | Each container = App + Dependencies (shares host OS) |
| **Size** | Large (often **GBs**) | Small (usually **MBs**) |
| **Startup Time** | Slow (30 sec – minutes) | Fast (seconds) |
| **Efficiency** | Heavy – uses more **RAM, CPU, Disk** | Lightweight – uses **less resources** |
| **Isolation** | Stronger (VMs are fully isolated) | Good enough for most use cases, but shares OS kernel |
| **Portability** | Less portable – VM images are heavy | Highly portable – run same container anywhere |
| **OS Support** | Can run **different OSes** on the same system | All containers share the **same OS kernel** as the host |
| **Use Case Examples** | Full system testing, running multiple OSes (Windows, Linux) | Microservices, CI/CD, fast deployment, cloud-native apps |
| **Snapshots & Rollback** | Supports full VM snapshots | Supports image versions and rollbacks |
| **Cloud Usage** | Still used in private cloud setups | **Standard** in modern cloud (Kubernetes, Docker) |

## What is Mounting?

**Mounting** means **attaching a storage device (like a USB drive or disk partition) to your system**, so that the system can read/write files from it.

Imagine plugging in a **USB drive**:

- Just plugging it in is **not enough**.
- Your system needs to "mount" it — i.e., connect it to a **folder** where you can **access its files**.

In Linux, everything is a file. So a disk becomes accessible as part of the **file system tree** — usually under `/mnt` or `/media`.

### Linux Example:

mount /dev/sdb1 /mnt/mydrive

This command tells the system:

> "Take disk **sdb1** and attach it at folder **/mnt/mydrive**."

Now, if you go to that folder, you'll see all the files from that disk.

## What is Unmounting?

**Unmounting** means **safely disconnecting** a mounted device so it can be removed without data loss.

Unmounting is like **safely ejecting a USB drive**:

- If you remove it directly, you may lose unsaved data.
- So, we "unmount" it first, to make sure nothing is in use.

### Linux Command:

umount /mnt/mydrive

> This safely disconnects the mounted disk from that folder.

## Mounting in Containers

In Docker or container systems, mounting is also used to **give containers access to host files**.

### Example (Docker):

docker run -v /home/user/data:/app/data mycontainer

This command **mounts your host folder** `/home/user/data` **into the container** at `/app/data`.

So the container can read/write to your actual system — very useful!

(windows m to ye automatic hota hai wese agr usb ki baat kre to baaki stuffs k liye apna apna hai )

| Problem | Mounting/Unmounting Solution |
|---|---|
| Can't access USB/disk | Mount it to a folder |
| Risk of data loss while removing | Unmount it first |
| Container needs local data | Mount host folder into container |
| Access files on network | Mount shared network storage |
| Keep system clean | Mount data on separate disk |

EOT

### Service Discovery and Health Checks

Servers crash due to various reasons like hardware faults and software bugs. Service Discovery and Health Checks are essential for maintaining a service ecosystem's availability and reliability. We talk about how a heartbeat service can be used to maintain system state and help the load balancer decide where to direct requests. Now when a server crashes, the heartbeat service identify and restart the service immediately on the server.

Service Discovery is another important part of deploying and maintaining systems. The load balancer is able to adapt request routing. Both features allow the system to report and heal issues efficiently.

## What are Microservices?

**Microservices** are small independent services that work together to form a complete application.

Example: In an e-commerce app:

- One service handles user accounts
- One handles payments
- One handles orders
- One handles shipping

These services need to **communicate** with each other.

## Problem: How do services find each other?

Services may:

- Move between machines
- Crash and restart
- Scale up/down dynamically

So their **IP addresses keep changing**. If Service A wants to talk to Service B, how does it know where to find it?

## Solution: Service Discovery

This is like a **phone book** for services.

### How it works:

1. Every service **registers itself** to a central service registry (like saying "Hey, I'm here!").
2. When a service wants to talk to another one, it **asks the registry** for the latest location.

### Tools for this:

- **Consul**
- **Eureka (Netflix)**
- **Kubernetes DNS**

### Example:

Order Service asks the registry:
"Where is the Payment Service?"

Registry replies:
"Payment Service is at IP 10.0.0.25"

## Problem: How do we know a service is still running?

Even if a service is registered, it might **crash or hang**. We don't want other services to send requests to a dead one.

## Solution: Heartbeats

A **heartbeat** is a simple signal that a service sends to say "I'm alive".

### How it works:

- Every few seconds (e.g., 10s), a service sends a **heartbeat** to the registry.
- If the registry **doesn't receive heartbeats** for a certain time (e.g., 30s), it marks the service as **unhealthy**.

### Example:

User Service sends heartbeat every 10s.

If no heartbeat is received in 30s:
→ Service is marked as down
→ Other services stop sending requests to it

## Where are Heartbeats used?

### In Microservices:

- To know which services are up and healthy
- To do **automatic failover**

**In Data Pipelines:** (A data pipeline is a system that moves data from one place to another and processes it along the way.)

Data pipelines have **workers** that process data (e.g., stream of user activity, logs).
If a worker crashes or stops working, we want to know!
Workers send **heartbeats** to a central controller.
If no heartbeat → that worker is restarted or replaced.

## How to Make a Resilient Architecture

"Resilient" means your system:

- Keeps working even if some services fail
- Can recover automatically
- Doesn't crash under pressure

### Key Techniques:

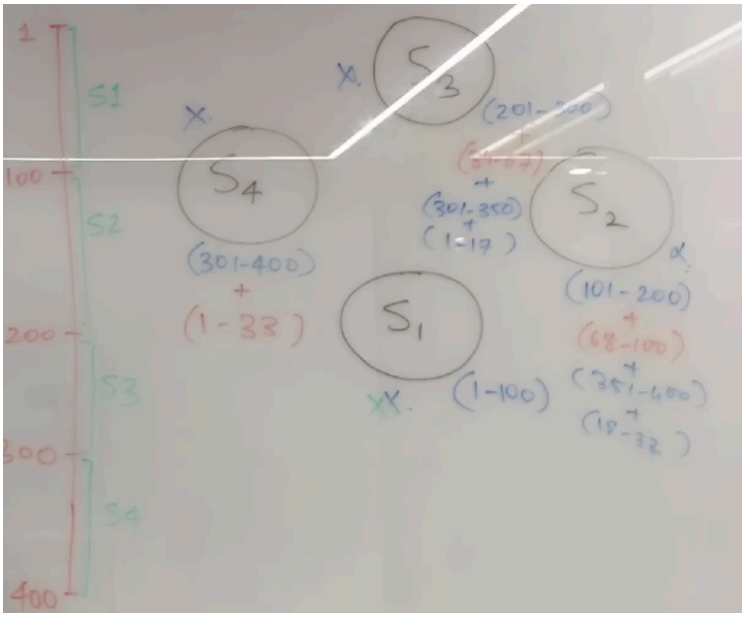
| Technique | Purpose |
|---|---|
| **Service Discovery** | So services can find each other |
| **Heartbeats** | Know which services/workers are healthy |
| **Retries with backoff** | Try again if something fails, but wait a bit longer each time |
| **Circuit Breakers** | Stop calling a service if it keeps failing, to prevent overload |
| **Bulkheads** | Isolate services so if one fails, it doesn't crash everything |
| **Monitoring and Alerts** | Know what's going wrong with dashboards and logs |
| **Auto-scaling** | Automatically add more instances when traffic is high |

EOT

## Cascading Failure

**What is a Cascading Failure?**
When designing systems on the server side, we often need to predict the capacity of a server and apply limits on the number of requests it can receive per second. This is mentioned in the service QPS document. QPS stands for queries per second.

The system design discussion helps us understand how to deal with requests based on priorities, how to deal with cascading failures, how to handle a large number of requests, viral posts or videos and their requirements.




Cascading failure happens when the failure of one service in a system causes a chain reaction, leading to the failure of other connected services. This usually occurs in microservices when one service becomes slow or unresponsive, and dependent services keep calling it without proper timeouts or limits, eventually overloading themselves. As each service fails, it pushes the load onto others, bringing down the entire system. It's like a domino effect. To prevent this, systems use timeouts, circuit breakers (which stop calls to a failing service), bulkheads (to isolate parts of the system), and fallback responses to keep other parts running even if one fails.

## Problems in Large-Scale Systems

Modern systems, especially those built on microservices and cloud infrastructure, face many challenges:

| Problem | What Happens |
|---|---|
| **1. Cascading Failure** | One service fails, others depend on it, and everything crashes like dominoes. |
| **2. Going Viral** | A sudden traffic spike (e.g., trending video or post) overloads servers. |

| | |
|---|---|
| **3. Predictable Load Increase** | Load rises at expected times (e.g., mornings, holidays) but still needs handling. |
| **4. Bulk Job Scheduling** | All cron jobs or tasks start at the same time and overwhelm the system. |
| **5. Popular Posts / Repeated Queries** | Same data is requested over and over again (e.g., viral tweets, profile views). |

## Solution Set

| Solution | What It Solves | Simple Explanation |
|---|---|---|
| **1. Rate Limiting** | Prevents overload, cascading failures | Restrict the number of requests per second to keep services healthy. |
| **2. Pre-Scaling** | Viral or scheduled traffic | Manually add more resources in advance of expected spikes. |
| **3. Auto-Scaling** | Predictable load increase | Automatically increase or decrease servers based on current traffic. |
| **4. Batch Processing** | Bulk jobs | Spread out workload across time or process in small chunks. |
| **5. Approximate Statistics** | High-load analytics | Use fast, low-cost estimates instead of real-time exact counts. |
| **6. Caching** | Repeated requests | Store and reuse frequent responses to reduce backend load. |
| **7. Gradual Deployment** | Risk of new features | Slowly release features (5% → 50%) to reduce the impact of bugs. |
| **8. Smart Coupling** | Cascading failure | Use timeouts, retries, and circuit breakers to stop failure from spreading. |
| **9. Request Throttling / Load Shedding** | Sudden bursts | Drop extra requests temporarily to let the system recover. |

| | | |
|---|---|---|
| **10. Message Queues (Kafka, RabbitMQ)** | Load smoothing | Queue up excess traffic and process it gradually. |
| **11. Monitoring + Alerts** | Health tracking | Watch service performance, detect failure early. |
| **12. Fallbacks / Graceful Degradation** | Service failure | Show cached data or basic responses when backend is down. |

## Key Principles Behind All These

- **Protect your system**: Rate limiting, throttling, circuit breakers
- **Use resources wisely**: Caching, approximate stats, batch processing
- **Stay ready for scale**: Auto/pre scaling, gradual rollouts
- **Avoid chain reactions**: Decouple services, use queues and timeouts

### A flow Explanation :

Jab aapke app par ek viral post aata hai, to traffic achanak se badh jaata hai. Is spike ko handle karne ke liye system already **pre-scaled** hota hai aur **rate limiting** laga hota hai, jisse extra requests ko control kiya ja sake. Viral post ka data pehle se **cache** mein hota hai, to sabhi users ko fast response milta hai bina backend ko overload kiye. Views aur analytics ko **approximate counting** ke through manage kiya jaata hai, jo fast aur lightweight hoti hai. Agar koi backend service slow ho jaye to **timeouts** aur **circuit breakers** us service se connection tod dete hain taaki baaki system pe asar na ho. **Background jobs** jaise notifications ya updates ko **batch processing** ke zariye dheere dheere process kiya jaata hai. Agar traffic aur badhne lage to system **auto-scale** karke naye servers add kar deta hai. Koi naya feature launch ho raha ho to usse **gradual deployment** ke through sirf kuch users tak pahunchaya jaata hai. Bahut zyada requests ek saath aaye to kuch ko **message queue** mein daal diya jaata hai ya **throttle** kar diya jaata hai. Poore system ki **monitoring** chalti rehti hai, jisse agar koi service issue kare to turant alert mil jaye. Is flow ke through system viral traffic ke bawajood stable aur responsive bana rehta hai.

EOT

Chalo ab thoda deep dive krna padenga in sab mein :

## 1. Rate Limiting

**Rate limiting** is a technique used to control the amount of incoming or outgoing traffic to/from a network resource — like an API or a web server — over a specific time window.

### Why Use Rate Limiting?

1. **Prevent abuse** (e.g., DoS attacks, spam, scraping)
2. **Ensure fair usage** (avoid one user consuming all the resources)
3. **Protect backend systems** from overload
4. **Ensure consistent performance** for all users

### Key Concepts

| Term | Description |
|---|---|
| **Limit** | The maximum number of allowed requests |
| **Window** | The time frame in which the limit applies (e.g., 1000 reqs per minute) |
| **Burst** | A sudden spike in traffic that may be allowed temporarily |
| **Quota** | A broader measure that may include multiple endpoints or services |

Quota mtlb kitni requests handle kri jaa skti hai
Aur window to ki quota k paas kitna time hai mtlb quota kitne time m reset hoga

### Algorithms rate limit krne k liye :

## 1. Fixed Window Counter

### How it Works:

- Divide time into **fixed intervals** (e.g., 1-minute windows).
- Maintain a counter per user/IP.
- If the number of requests exceeds the limit in the current window, block them until the window resets.

### Example:

```
Window = 1 minute

Limit = 100 requests

If a user sends 100 requests at 12:00:10, they are blocked
until 12:01:00.
```

### Pros:

- Simple to implement
- Low memory usage

### Cons:

- **Burst issue**: User could send 100 requests at `12:00:59` and again 100 at `12:01:00`.

### Good for:

- Simple applications with tolerant load spikes

## 2. Sliding Window Log

### How it Works:

- Store a **timestamp for every request** in a log (usually per user/IP).
- For each request, check how many timestamps are in the last N seconds.

### Example:

```
Limit = 100 requests per minute
```

```
Check if user made < 100 requests in the past 60 seconds.

Agar 100 ho chuki to reject karega, chahe window change hua ya nahi.
```

**Pros:**

- Precise control over rate
- Smooth rate limiting

**Cons:**

- Memory-intensive (especially at scale)
- Storage and performance concerns in high traffic

**Good for:**

- High-security APIs (e.g., banking)
- Use cases needing exact limits

## 3. Sliding Window Counter (Approximation of Log)

**How it Works:**

- Divide time into **smaller intervals** (e.g., 1 second), and count requests per interval.
- When a new request arrives, sum recent intervals and decide.
- This **smooths out spikes** while using less memory.

**Example:**

```
Store: [58s: 10 req, 59s: 20 req, 60s: 30 req]
Total last 60s = 60 reqs → Accept
```

Mtlb this stores according to time but checks according to block ki pichhle n blocks m kitne req. Aayi hai
Sliding Counter **assume karta hai** ki har window (bucket) ki request **uniformly spread** hui hai. Jabki aisa zaroori nahi hota.

**Pros:**

- More memory efficient than full logs

- Better precision than fixed windows

**Cons:**

- Slightly more complex
- Still needs interval data storage

**Good for:**

- APIs with moderate traffic and need for accuracy

(mtlb isme frk sirf boundaries ka hai aur koi frk ni hai jyada mtlb 1 - 60 ye 60 sec, hue log ke
Lein counter k 1 - 10 , 11 - 21 , 22 - 32 , 33 - 43 , 44 - 54 , 54 - 64 ye hue 6 blocks 10 -10 seconds k aur vo isko 60 min. Ki request boldeta hai )
(Sliding Counter ne accuracy thodi compromise karke speed, simplicity, aur scalability ka problem solve kiya.

## 4. Token Bucket

**How it Works:**

- Bucket holds **tokens** (representing the right to make a request).
- Tokens are added at a fixed rate.
- Each request removes 1 token.
- If no tokens, request is rejected (or queued).

**Example:**

```
Bucket size = 10, refill rate = 1 token/sec
→ User can make 10 requests instantly (burst)
→ After that: 1 per second
mtlb token bucket mei esa hota hai
ki token mtlb requests se jo cancel out honge
bucket mtlb ek fized set of tokens for continuous flow of time
to ek baar kisi time pr bucket ne 5 token decide kre hai aur
1token per second to mtlb 5 sec. m 5 request handle kri jaa
skti hai
```

ab suppose 1st second m ek request aayi aur baaki 4 m ek bhi nahi to aage use aage 4  token count hoga aur aage k tokens bhi add ho skte hai
jb tk token hai tb tk request accept ki jaayegi aur nahi hai tokens khmt to reject

**Pros:**

- Allows **short bursts**
- Smooth traffic after bursts
- Easy to adjust rate and burst size

**Cons:**

- Slightly more complex
- Needs background token refill logic

**Good for:**

- APIs needing burst handling (e.g., user login attempts)
- Applications where consistent rate is okay after a burst

## 5. Leaky Bucket

**How it Works:**

- Requests are **queued** in a "bucket".
- Requests leave the bucket at a **constant rate** (like a leaky bucket).
- If the bucket is full, new requests are dropped.

**Example:**

```
Bucket capacity = 100
Outflow rate = 1 req/sec
→ Can smooth out bursty traffic into steady flow
```

mtlb ek bucket hai aur usme ek chhed hai niche ,jese jese uss bucket m request aayegi vo chhed se nikl kr queue ki capacity k hisaab se queue m jaayegi aur phir process hogi , jb tk vo process horhi hai agr tb tk bucket full hogyi aur full hone k baad bhi request aayi to vo automatic drop hojayegi ?

**Pros:**

- Ensures **constant processing rate**
- Smooth and predictable traffic

**Cons:**

- Bursts may be **delayed or dropped**
- Less flexible than token bucket for spikes

**Good for:**

- Systems that process messages/events
- Queued jobs (e.g., email senders)

**Tools & Libraries for Rate limiting:**

- **NGINX** (rate limiting module)
- **Envoy**, **Kong**, **Traefik**
- **Redis** (for distributed counters)
- **Libraries**:
  - Python: `ratelimit`, `limits`
  - Node.js: `express-rate-limit`
  - Golang: `golang.org/x/time/rate`

EOT

## Pre - Scaling ,Auto - Scaling and Batch - Processing

### 1. Auto-scaling hoti kaise hai?

Auto-scaling ka matlab hai: **System khud decide kare ki kitne servers (instances, pods, containers) chahiye**, based on traffic ya load.

### Kaise hoti hai:

1. **Monitoring Tools** (jese metrics gather karte hain):
    - CPU kitna use ho raha hai?
    - Memory ka load?
    - Requests per second?
    - Response time?

2. **Threshold Set hota hai** (rule bna hota hai):
    - "Agar CPU 70% se upar jaaye 1 min ke liye → naye pod chalu karo"
    - "Agar traffic kam ho gaya → kuch pod band karo"

3. **Scaling Controller** use hota hai (alag platform pe alag):
    - Kubernetes: `Horizontal Pod Autoscaler (HPA)`
    - AWS: `Auto Scaling Groups (ASG)`
    - Azure: `VM Scale Sets`
    - GCP: `Instance Groups`

4. **Action**:
    - Auto-scaling engine naya pod/server/container chalu kar deta hai.
    - Jab kaam kam ho jaata hai, toh extra wale band ho jaate hain (scale-in).

## 2. Pre-scaling hoti kaise hai?

Pre-scaling ka matlab: **Traffic aane se pehle hi system ko tayyar kar lena**.

### Kaise hoti hai:

1. **Manual ya Scheduled hoti hai**:
    - "Roz subah 8 baje se shaam 6 baje tak 5 instances chahiye"
    - "Black Friday wale din 10 extra pods chahiye"

2. **Tools/Techniques**:
    - **Kubernetes me `CronJob`, `Keda`, ya `ScheduledScaler`**
    - **AWS me Scheduled Scaling Policies**
    - **GCP me Predictive Autoscaling** (AI-based)

3. **Use-case**:
    - Jab aapko pehle se pata hai traffic kab aayega:
        - Office hours
        - Exam time
        - Diwali sale

## 3. Kaise pata chale ki scaling ho rahi hai ya nahi?

### Monitoring Tools:

- **Grafana + Prometheus** (Kubernetes metrics)
- **AWS CloudWatch** (EC2, ECS, Lambda metrics)
- **Azure Monitor / Application Insights**
- **GCP Stackdriver**
- `kubectl get hpa` → Kubernetes me pod scaling status
- `kubectl top pods` → CPU/memory usage

### Logs aur Alerts:

- Alerting systems (e.g., Alertmanager) se message milta hai ki scale-out/in ho gaya
- Events me dikhega: "Scaled from 3 to 6 replicas"

## 4. Kab konsi scaling zyada better hoti hai?

| Situation | Preferable Scaling Type | Reason |
|---|---|---|
| Roz subah 9 baje load badhta hai | **Pre-scaling** | Pehle se tayyari ho jati hai |
| Kabhi kabhi traffic unpredictable hota hai | **Auto-scaling** | Real-time react karta hai |
| Big event ya product launch hai | **Pre-scaling + Auto-scaling dono** | Load ka idea hai, but unpredictable spike bhi ho sakta hai |
| App public website hai (viral ho sakti hai) | **Auto-scaling** | Traffic kabhi bhi badh sakta hai |

## 5. Scaling ki limit hoti hai kya?

### Haan, hoti hai:

1. **Infrastructure Limit**:
    - Nodes ki max limit (e.g., AWS EC2 quota)
    - Container image size, startup time

2. **Quota & Budget**:
    - Cloud provider har service ka limit rakhta hai (e.g., 100 pods max, 20 CPUs max)

3. **Manual Limit Set kar sakte ho**:

Kubernetes me:
minReplicas: 2
maxReplicas: 10

4. **License-based Limits** (some apps allow only X instances)

### Limit pta kaise chale:

- Cloud console me quota dekh sakte ho (AWS, GCP, Azure)
- Kubernetes me `kubectl describe hpa` me maxReplicas dikhega
- Alert system ya logs batayenge agar limit hit ho gaya

( sabhi tools k baare m ‘ll study later because need more experience for platforms and work with cloud )

Abhi **Batch Processing** padhte hai

**Batch processing** means doing a large number of similar tasks **together**, automatically, usually **on a schedule**, **without human help**.

Imagine you have to print 100 certificates. You don't want to print each one by hand — instead, you give the computer all names at once and click "Print All." That's **batch processing**.

## How Does Batch Processing Work? (Step-by-Step)

Let's say a company wants to send salary slips to 1,000 employees every month.

### Step-by-step:

1. **Scheduler starts the job** (e.g., at 2 AM on the 1st of every month)
2. **Job reads data** from the employee database
3. **Calculates salary** using formulas (basic + bonus – tax)
4. **Creates salary slip PDFs**
5. **Saves files** or emails them
6. **Logs** success or failure

No one needs to sit and do this manually — everything happens in a single **batch**.

**Batch processing means doing lots of similar tasks together —**
But to **prevent failures**, we **process them in smaller chunks** behind the scenes.

**Say you order 500 cupcakes for a party:**

- The bakery **delivers them in 5 boxes of 100** (to avoid crushing them).
- But when you open them — **you see all 500 cupcakes!**

## Types of Batch Workload Spreading

### A. Time-Based Spreading

Run jobs during off-peak hours:

- Example: Run billing or backups at 2 AM
- Helps avoid conflict with real-time users

### B. Chunk-Based Processing (Micro-batching)

Break a huge job into small parts:

- Example: Process 1000 records in 10 groups of 100
- Each chunk is easier to monitor, retry, and won't overload systems

## Key Techniques Used in Safe Batch Processing (In this context)

| Technique | Purpose | Analogy |
|---|---|---|
| **Chunking** | Break big job into small safe parts | Don’t eat whole pizza at once – slice it |
| **Throttling** | Add delay between chunks | Give system time to breathe |
| **Rate Limiting** | Control how fast requests go | Don't flood the server |
| **Backoff & Retry** | Wait longer after each failure | Don’t knock 100 times if no one’s home |
| **Resource Isolation** | Run batch jobs on separate machines or nodes | Don’t cook and iron clothes on the same power socket |
| **Scheduling during off-peak** | Run jobs when system is idle | Like washing machine at midnight |
| **Timeouts & Circuit Breakers** | Don’t let stuck tasks run forever | Auto-shutoff feature in machines |

## Real-life Systems That Use Batch to Prevent Failures

| Company | Use-case |
|---|---|
| **Netflix** | Runs batch jobs on separate clusters using spot instances so live streaming is never affected |
| **Airbnb** | Runs report generation in off-peak hours using Airflow |
| **Amazon** | Splits order processing in chunks using SQS + Lambda + Batch jobs |
| **Banking apps** | Use night-time batch windows for statement generation to avoid interfering with real-time transactions |

## Tools Commonly Used in This Context

| Tool | Why it's helpful |
|---|---|
| **Apache Airflow** | For orchestrating smart batch workflows with retries, scheduling, and chunking |
| **Apache Kafka** | Helps buffer tasks so one spike doesn’t overload downstream |
| **Kubernetes CronJobs** | Runs jobs in containers with resource limits and isolation |
| **AWS Batch** | Offloads work to managed compute so your main system stays safe |
| **Resilience4j** | Circuit breaker and retry pattern libraries |

| | |
|---|---|
| **Prometheus/Grafana** | Real-time monitoring of system health and batch job impact |

**Batch processing is like fire — powerful but dangerous.**
If you handle it smartly, it helps prevent system collapse.
If you ignore its design, it can bring everything down.

EOT

## Caching , gradual deployment and smart coupling

**Caching** means **saving frequently used data temporarily** so that it can be fetched faster next time — instead of recalculating or reloading it from the original (slow) source.

| Without Cache | With Cache |
|---|---|
| Fetches data from slow backend (e.g., database, server, disk) | Fetches from fast memory (e.g., RAM) |
| Slow and costly | Fast and cheap |
| Increases load on systems | Reduces load and response time |

Caching is used **everywhere** in computer systems:

| Where | What Is Cached |
|---|---|
| Web browsers | Images, pages, JS/CSS files |
| Web servers | API responses |
| Databases | Recently queried rows |
| Operating systems | Frequently used disk files |
| Mobile apps | User settings, images, tokens |
| CDNs (Cloudflare, Akamai) | Static content across the globe |

## Cache Expiry Policies (How Long to Keep?)

| Policy | Meaning | Example |
|---|---|---|

| | | |
|---|---|---|
| **TTL (Time-To-Live)** | Keep for X seconds | Cache for 10 mins |
| **LRU (Least Recently Used)** | Remove oldest unused item | Keep top 100 most-used results |
| **Manual Invalidation** | Clear cache when data changes | Clear when product price is updated |

## Types of Caching

## 1. In-Memory Cache

A **very fast** type of cache stored in **RAM (memory)**.
It's used to store temporary data that needs to be accessed frequently.

### Why It's Used:

- Reading from memory (RAM) is much faster than hitting a disk or a database.
- Perfect for **frequently used data** like:
    - API results
    - Session data
    - Computation results

### Tools Used:

- **Redis** (most common)
- **Memcached**

### Example:

You're an e-commerce website. Your home page shows "Top 10 Trending Products". Instead of querying the database every time, store this list in Redis for 5 minutes.

### Pros:

- Super fast
- Great for small, hot data

### Cons:

- Data is lost when server restarts (unless persistent Redis is used)

- Limited by available RAM

## 2. Disk-Based Cache

Caching on the **hard disk** (HDD/SSD) instead of RAM.
Slower than memory, but **can store much larger data** and is **persistent** (stays even after restart).

### Why It's Used:

- To cache **large files** (videos, images, or static HTML)
- When memory is not enough

### Tools Used:

- File system cache (OS-level)
- Local disk caching in browsers, apps, or even APIs

### Example:

A video streaming app caches recently watched videos on disk, so if the user replays it,
the app can load it from disk instead of downloading again.

### Pros:

- Bigger storage than memory
- Persistent

### Cons:

- Slower than in-memory
- Disk I/O can be a bottleneck if not managed

## 3. Browser Cache

Your web browser (like Chrome or Firefox) **automatically stores files** like:

- HTML
- CSS
- JavaScript
- Images

So that websites load **faster on repeat visits**.

**Why It's Used:**

- Avoid downloading the same content again
- Saves bandwidth
- Faster page load

**Technologies Used:**

- `Cache-Control` headers
- `ETags`
- Service workers

**Example:**

You open Instagram in your browser.
Your browser remembers and caches the logo image, styles, and scripts.
Next time, it shows those from cache — not the internet.

**Pros:**

- Saves network usage
- Very fast repeat visits

**Cons:**

- Can show outdated data if cache is not refreshed properly

## 4. CDN Cache (Content Delivery Network)

A **CDN (like Cloudflare, Akamai, or AWS CloudFront)** stores copies of your files (images, videos, scripts, etc.) **in servers around the world**.

**Why It's Used:**

- To serve static content **from the closest location to the user**
- Reduces **latency** and **server load**

**Common CDNs:**

- Cloudflare
- Akamai
- AWS CloudFront
- Fastly

**Example:**

Your website has users in India, US, and Europe.
Instead of sending all traffic to your server in the US,
CDN servers in each region serve the files directly.

**Pros:**

- Global performance boost
- Offloads traffic from your servers

**Cons:**

- Might require proper setup of headers (e.g., cache-control)
- Cache invalidation can be tricky

## 5. Application-Level Cache

Cache **built directly into your application logic**.

Developers decide **what to cache**, **how to cache**, and **when to invalidate**.

This is the **most flexible** type — often layered on top of Redis or memory.

**Why It's Used:**

- Customize what data gets cached
- Used in API response caching, function result caching, etc.

**Examples:**

- Django/Flask cache (Python)
- Spring Cache (Java)
- Guava Cache (Java)
- In-code dictionaries/variables

**Example:**

```
@cache_for(60)  # cache for 60 seconds
def get_top_news():
    return expensive_news_query()
```

**Pros:**

- Total control
- Great for computed data

**Cons:**

- Can get complex to manage
- Must manually handle expiry and invalidation

## 6. Database Cache

Caches **frequently queried results** so the database doesn't have to do the same work again and again.

Can be **inside the DB engine** or **external** (like Redis in front of the DB).

**Why It's Used:**

- Reduces query time
- Reduces load on DB servers

**Techniques Used:**

- Query cache
- Materialized views
- External cache (Redis/Memcached)

**Example:**

SELECT * FROM products WHERE category = 'laptops'
This is requested 1000s of times.
Store the result in Redis for 10 minutes.

**Pros:**

- Boosts performance for read-heavy queries
- Saves DB resources

**Cons:**

- Cache invalidation must be handled carefully
- Write-heavy systems don't benefit much

A smart cache can **prevent cascading failure**:

- If your backend goes down:
    - Cached data can still serve users
    - Less panic on the system
    - Time to recover

| Pros | Cons |
|---|---|
| Fast | Can show outdated (stale) data |
| Reduces system load | Hard to invalidate on change |
| Cost-effective | Extra memory/storage needed |

| Tool | Use-case |
|---|---|
| **Redis** | In-memory key-value cache (most popular) |
| **Memcached** | Lightweight memory cache |
| **CDNs (Cloudflare, Akamai)** | Caches static files close to users |
| **Varnish** | Web accelerator caching |
| **Browser Storage** | Local caching (e.g., Service Workers) |
| **Spring Cache / Guava Cache** | Java apps |

| | |
|---|---|
| **Flask/Django Cache** | Python apps |

## What Is Gradual Deployment?

> **Gradual Deployment** means **releasing a new version of your app or feature slowly**, step-by-step, to a small portion of users before rolling it out to everyone.
>
> **"Don't launch to everyone at once — test on a few, fix issues, then go full."**

## Real-Life Analogy:

You own a pizza chain and want to try a **new recipe**.

- You **don't launch it in all stores at once.**
- First, test in 1 store
- If it works → roll out to 10 stores
- Still good → roll out to all

That's **gradual deployment**: Safe, step-by-step launch.

## Why Gradual Deployment?

| Full Deployment | Gradual Deployment |
|---|---|
| All users get changes instantly | Only a few users get it at first |
| One small bug can affect everyone | Bugs are limited to a small group |
| Harder to rollback | Easy to stop or adjust mid-way |

## Common Strategies for Gradual Deployment

### 1. Canary Releases

- Release new version to a **small % of users** (like 5%)
- Monitor for issues
- If all looks good, increase to 20%, 50%, 100%

**Example:**
"New search algorithm is shown only to 10% of users today."

### 2. Feature Flags / Toggles

- Code is **already deployed** but **hidden** behind a flag.
- Turn it **on only for selected users**, groups, or countries.

**Example:**
"Only beta testers in India see the new homepage."

Tools: **LaunchDarkly, Unleash, Firebase Remote Config**

### 3. Blue-Green Deployment

- Two environments: **Blue (old)** and **Green (new)**
- Traffic starts going to the new (Green) slowly
- If something breaks, switch back to Blue instantly

**Example:**
Run both versions side-by-side and **switch traffic gradually**

### 4. Traffic Splitting / Shadow Testing

- Route only **some traffic** to the new system
- Users don't know — but you're **testing in the background**

**Example:**
Search queries go to both old and new search engine, but only old one returns results.

## Where Is Gradual Deployment Used?

| Industry | Use-Case |
|---|---|
| **Web apps** | New homepage, new pricing, new design |
| **Mobile apps** | Rollout new version to 10% of users |
| **APIs** | New API version tested with internal users |
| **AI/ML systems** | Test new models silently on real data |
| **Games** | A/B test levels or features |

## Benefits of Gradual Deployment

| Benefit | Why It Matters |
|---|---|
| Safer rollouts | Catch bugs before full release |
| Better monitoring | Track metrics in small groups |
| A/B testing | Compare old vs new features |
| Easy rollback | If bug found, rollback is limited |
| Targeted control | Choose who sees what and when |

## Challenges

| Challenge | Solution |
|---|---|
| Data inconsistencies | Keep data format backward-compatible |
| Code complexity | Use centralized feature flag system |
| Hard to test all scenarios | Use good observability/logging |
| Requires DevOps setup | Use tools like Kubernetes, Istio, or LaunchDarkly |

## Popular Tools for Gradual Deployment

| Tool | Purpose |
|---|---|
| **LaunchDarkly** | Advanced feature flagging |
| **Unleash** | Open-source feature toggles |
| **Firebase Remote Config** | Remote feature control for mobile apps |
| **Istio / Envoy** | Traffic routing for microservices |
| **Kubernetes + Argo Rollouts** | Progressive delivery for container apps |
| **GitLab/GitHub Actions** | Pipeline support for staged releases |

Ab smart Coupling smjhenge !

## What is Coupling in Software?

**Coupling** refers to **how much one part of a system depends on another.**

### Think of it like building blocks:

- If blocks are **tightly glued**, one falling block breaks the whole wall.
- If blocks are **loosely connected**, one falling block doesn't disturb the rest.

So, in software:

| Type | Meaning |
|---|---|
| **Tight Coupling** | Components are heavily dependent on each other |
| **Loose Coupling** | Components can work more independently |

## So, What is Smart Coupling?

> **Smart Coupling** means **connecting components in a way that balances coordination and independence**:

- They **work together** smoothly,
- But **don't drag each other down** during failure or scaling.

"Be connected smartly — not blindly."

## How to Achieve Smart Coupling

Smart coupling is all about **connecting parts of your system wisely**, so they can work **together smoothly**, but also **fail independently** without taking the whole system down.

### 1. Use Asynchronous Communication (Queues, Events)

**Traditional way (tight coupling):**
Service A calls Service B directly.
If B is slow or down, A gets stuck or crashes.

**Smart way (loose coupling):**
Service A sends a message to a **queue or event stream**.
Service B reads and processes the message **when it's ready**.

**Tools:** RabbitMQ, Kafka, AWS SQS, Google Pub/Sub

**Why it's smart:**

- Services don't wait for each other
- Failures in one service don't affect the others
- You can retry failed messages
- Easy to scale independently

### 2. Use APIs and Interfaces — Not Internal Access

**Bad practice:**
One service directly accesses another's database or logic.

**Good practice:**
Each service should expose a **well-defined API** — like:

```
GET /user/123
```

That way, other services **only talk through that interface**, not through internals.

**Why it's smart:**

- Each service can evolve independently
- Internal changes don't break others
- Encourages clear boundaries and modular design

### 3. Use Circuit Breakers and Timeouts

When one service depends on another, things can go wrong.

So be **defensive**:

- Add **timeouts** (don't wait forever for a reply)
- Use a **circuit breaker**: if a service is failing repeatedly, stop calling it for a while

**Tools:** Resilience4j, Netflix Hystrix (legacy), Istio

**Why it's smart:**

- Avoids cascading failure
- Prevents your whole system from crashing just because one service is slow or broken
- Gives services time to recover

### 4. Use API Gateways or Proxies for Isolation

Instead of calling services directly, route traffic through an **API gateway** or **proxy**.

The gateway can:

- Route requests to the right service
- Add caching
- Handle failures
- Load balance traffic
- Enforce security

**Tools:** NGINX, Kong, AWS API Gateway, Istio

**Why it's smart:**

- You control how services interact
- Easier to monitor and debug
- Keeps services isolated and safer from outside traffic

### 5. Use Loose Data Contracts

Tightly coupled services often depend on **fixed data structures** — which makes changes risky.

Instead, use **loose contracts**:

- Version your APIs (`/v1/users`, `/v2/users`)
- Make fields optional when possible

- Use defaults for missing values

**Why it's smart:**

- You can update or add new fields without breaking older services
- Allows flexibility and safer evolution of the system

### 6. Design for Failure

Smart systems assume that **anything can fail at any time** — and prepare for it.

Good design includes:

- **Retries**: Try again if something temporarily fails
- **Fallbacks**: Use a cached response or backup plan
- **Partial results**: Show available data even if some parts failed

**Why it's smart:**

- Your app keeps running even when something goes wrong
- Users get a better experience
- Makes the system more robust and fault-tolerant

### 7. Use Domain-Driven Design (DDD)

DDD is a way of organizing your system into **independent "domains" or areas** (like `orders`, `users`, `billing`, `products`).

Each domain:

- Owns its own data
- Handles its own logic
- Communicates with others through clear contracts or events

**Why it's smart:**

- Reduces cross-dependencies
- Easier to scale and deploy parts of the system independently
- Teams can work on different domains without blocking each other

### Why Smart Coupling Is Important

| Without Smart Coupling | With Smart Coupling |
|---|---|
| One failure can break the whole system | Failures are contained |
| Hard to scale individual parts | Easier to scale piece by piece |
| Tight dependencies | Decoupled but still communicate efficiently |
| Difficult to change or update | Easier to evolve the system |

### Real-Life Example: Amazon

- Checkout service doesn't directly call the shipping service.
- It puts an "order placed" event in a queue.
- Shipping service **picks up the event** and handles it **asynchronously**.
- If shipping fails, it **retries** or alerts.
- **Checkout isn't affected.**

EOT

### Anomaly Detection in Distributed Systems

Detecting anomalies (or anomaly detection) means identifying data points, events, or patterns that deviate significantly from the expected behavior in a dataset. These "outliers" can signal critical issues, such as fraud, system failures, or cyberattacks.

- **Point anomalies**: A single data point is unusual (e.g., a ₹10,000 transaction in a ₹100/day pattern).
- **Contextual anomalies**: Unusual in a specific context (e.g., high CPU usage at 2 a.m.).

- **Collective anomalies**: A group of related unusual data points (e.g., a spike in errors followed by a drop in traffic).

## Where Anomaly Detection Fits in System Design

1. **Monitoring and Observability**
    - Metrics (latency, error rate, throughput)
    - Logs (unusual error patterns)
    - Traces (detecting abnormal request paths)
2. **Alerting Systems**
    - Dynamic thresholds (e.g., based on baseline)
    - Rate-of-change detection (e.g., sudden spike in API calls)
3. **Auto-healing / Auto-scaling**
    - Use anomalies to trigger container restarts, scale resources, or re-route traffic
4. **Rate Limiting / Abuse Detection**
    - Detect IPs or users doing something unusual (e.g., 1000 requests/min)
5. **Data Pipelines**
    - Detect schema drift, unexpected nulls, out-of-order or missing data

| Aspect | Client-Side Anomaly | Server-Side Anomaly |
|---|---|---|
| **Where it happens** | On the **user's side** (browser, mobile app, etc.) | On the **server or backend** (APIs, DB, compute) |
| **Example issues** | - App crash- Slow UI- Too many requests sent- Repeated login attempts | - High CPU- Memory leak- Service down- Spike in 500 errors |
| **Who detects it** | Often reported by user or captured by frontend monitoring tools (e.g., Sentry, Datadog RUM) | Detected by backend monitoring tools (e.g., Prometheus, CloudWatch) |

| | | |
|---|---|---|
| **Cause example** | Bug in frontend code → sends wrong request format | Bug in backend service → throws error on valid input |
| **Impact** | One user or subset affected | Can impact everyone (especially if service is shared) |

## How Do You Fix Server-Side Anomalies?

### Step 1: Know That Something Is Wrong

- A user complains: “The app is stuck!”
- You get an **alert** from a monitoring tool
- You see something weird in your system dashboard (if there is one)

### Step 2: Find Out What Went Wrong

- Did the server crash?
- Is it using too much memory or CPU?
- Is the database working?
- Is one service calling another service too many times?

You usually use:

- **Logs** → Like the server’s diary; it writes down everything
- **Metrics** → Numbers that tell you how the system is doing (like speed, errors)
- **Traces** → A way to follow what happened in the system, step by step

But don’t worry — even checking one log file is enough to start.

### Step 3: Fix the Issue

Examples:

| Problem | Simple Fix |
|---|---|
| Server crashed | Restart it |
| Using too much memory | Restart, or fix memory usage in code |

| | |
|---|---|
| DB query too slow | Improve the query (ask less data) |
| One service is overloaded | Add more copies of it (called “scaling”) |
| Wrong config | Fix the mistake and restart the service |

You don’t need to know tools like Kubernetes yet — just understand the idea:

> "Look at what’s broken and restart or fix it"

### Step 4: Make Sure It’s Really Fixed

- Try the app or site again
- See if the error goes away
- Watch the logs — are new errors coming?
- Ask: is it still slow or crashing?

### Step 5: Prevent It from Happening Again

Once it’s fixed, ask:

- Why did this happen?
- Can we catch it earlier next time?
- Can we add better logs?
- Can we add an alert that warns us before it breaks?

## What Are Isolation Trees?

**Isolation Trees** (used in **Isolation Forests**) are a machine learning technique designed **specifically to detect anomalies**.

Instead of learning what "normal" is, it tries to **isolate each data point**.

- **Anomalies** are rare and different → **easier to isolate** (need fewer splits)
- **Normal data** is common and similar → takes more steps to isolate

Think of it like:

If something is weird, it gets caught quickly.
If something is normal, it takes longer to find what makes it different.

## Why Use Isolation Trees for Anomalies?

- Works well with **high-dimensional data**
- No need for labels (unsupervised)
- Fast and scalable
- Good for both **batch jobs** and **real-time detection**

## Example

Let's say you want to find **anomalous users** based on their behavior:

| User | Login Count | Time Spent (min/day) |
|---|---|---|
| U1 | 5 | 30 |
| U2 | 4 | 45 |
| U3 | 6 | 25 |
| U4 | 1 | 1 ← Anomaly |
| U5 | 7 | 40 |

### Step 1: Install Required Library

```
pip install scikit-learn
```

### Step 2: Sample Code

```
from sklearn.ensemble import IsolationForest

import numpy as np
```

```
import pandas as pd

# Sample user behavior data
data = pd.DataFrame({
    "login_count": [5, 4, 6, 1, 7],
    "time_spent": [30, 45, 25, 1, 40]
})
# Train Isolation Forest
model = IsolationForest(contamination=0.2, random_state=42)
model.fit(data)

# Predict anomalies
data['anomaly'] = model.predict(data)

# -1 = Anomaly, 1 = Normal
print(data)
```

**Output:**

| login_count | time_spent | anomaly |
|---|---|---|
| 5 | 30 | 1 |
| 4 | 45 | 1 |

| | | |
|---|---|---|
| 6 | 25 | 1 |
| 1 | 1 | **-1** ← Anomaly |
| 7 | 40 | 1 |

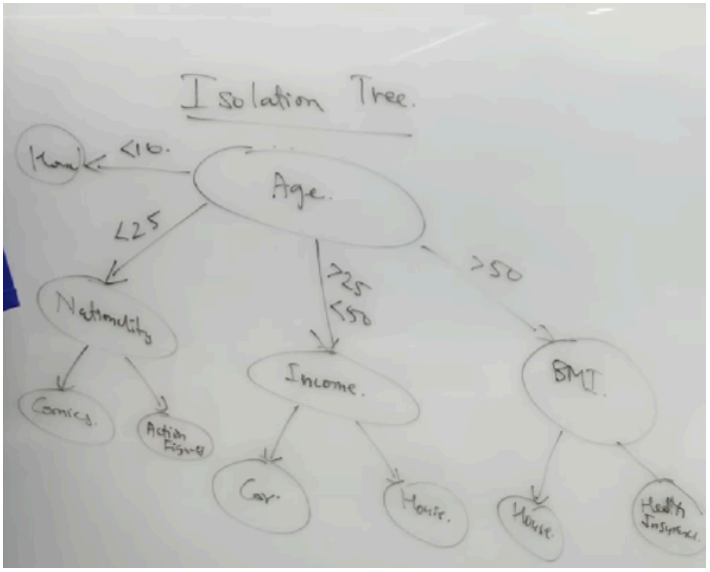
You just detected a weird user behavior using Isolation Trees!

## How It Works Internally

- Isolation Forest builds **multiple trees**.
- Each tree randomly selects a feature (like login count or time)
- Then picks a random split value
- Points that are isolated in fewer splits → are likely **anomalies**
- Each data point gets an **anomaly score**

## When to Use

- Server metrics (e.g., sudden spike in CPU)
- User behavior (e.g., unusual usage pattern)
- Transaction monitoring (e.g., fraud detection)
- Sensor data (e.g., out-of-range values)




We are more forgiving towards false positives

jo data point jaldi isolate ho jaaye, wo anomaly ho sakta hai.

Yaha data ko **Age**, **Income**, **BMI**, etc. jaise features ke basis pe split raha hai — taaki **alag-alag**

**logon ke patterns** samjhe jaa sakein.

## What Are Production Alerts?

**Production alerts** are **automatic warnings** sent when something goes wrong (or seems wrong) in your **live system** — also called the “production environment”.

They help **engineers know** there’s a problem **before users notice** or get affected.

## What Do Production Alerts Monitor?

| What’s Monitored | Example Alert |
|---|---|
| **Error Rate** | "More than 5% of API calls are failing!" |
| **Latency** (Speed) | "Login API is taking more than 2 seconds!" |
| **System Load** | "CPU usage on server is above 90%!" |
| **Crashes** | "Service A restarted 5 times in 10 minutes!" |
| **Traffic Drop** | "Sudden drop in user logins in last hour!" |
| **Disk Space** | "Database disk is 95% full!" |

## Tools That Create Production Alerts

| Tool | What It Does |
| --- | --- |
| **Prometheus + Alertmanager** | Most popular open-source monitoring/alerting |
| **Datadog** | Cloud monitoring with smart alerts |
| **AWS CloudWatch** | Used in AWS-based systems |
| **Grafana** | Visual dashboards + alert rules |
| **New Relic**, **Splunk**, etc. | Full-stack observability platforms |

EOT

## Distributed Caching , Cache Policies , Global cache

Distributed caches are caching systems that store **data across multiple nodes** (servers) in a **distributed environment**, rather than on a single machine. They're used to **speed up data access** in large-scale systems by keeping frequently accessed data closer to the application.

### What is a Distributed Cache?

- Imagine a cache that spans **multiple servers** instead of just one.
- These servers **work together** to store and retrieve key-value data efficiently.

- Common tools: **Redis Cluster**, **Memcached**, **Hazelcast**, **Apache Ignite**, **Ehcache**, etc.

### Why Use Distributed Caches?

- To **handle large datasets** that don't fit on one server.
- To ensure **low latency** even as your application scales.
- To **reduce load on databases** by storing frequently accessed data in memory.

### Example

Let's say your app has 10 million users. Every time a user logs in, you fetch their profile info from the database. That causes a bottleneck.

Instead:

- Store user profiles in a **distributed cache**.
- When User A logs in, the system checks the cache first.
- If not present (a "cache miss"), it fetches from DB and stores it in cache for next time.

## What Data Inconsistencies Can Occur in Distributed Caching?

### 1. Stale Data

- **What happens:** You read an old value from the cache that doesn't reflect the latest update in the database or another cache node.
- **Why it happens:**
    - The cache wasn't **invalidated** or updated after a DB write.
    - Data hasn't yet **replicated** to all nodes (in eventual consistency models).

- **Example:** User A updates their profile picture, but the app still shows the old one from cache.

### 2. Race Conditions

- **What happens:** Two or more processes write to the same key at nearly the same time, causing a **write conflict**.
- **Why it happens:** No locking or version control on the key during updates.
- **Example:**
    - Process A writes `value=10`, then Process B writes `value=20`.
    - If order flips, you get the wrong final value.

### 3. Replication Lag

- **What happens:** An update happens on the **primary node**, but the **replica nodes lag** behind before syncing the change.
- **Why it happens:** Asynchronous replication is used for performance.
- **Example:** One user sees updated product inventory, another still sees the old stock level.

### 4. Partition Brain (Split-Brain Scenario)

- **What happens:** The system splits into two or more **isolated partitions** (e.g., due to a network failure), and each believes it is the master.
- **Why it happens:** Lack of coordination between nodes.
- **Example:** Both Node A and Node B accept writes for the same key, then conflict on reconnection.

### 5. Write Skew

- **What happens:** Two transactions read the same old value and write **conflicting updates** based on outdated data.
- **Why it happens:** Missing transactional control or no check for version/timestamp.
- **Example:** Two users withdraw from the same bank account at once, both think balance is ₹10,000 — total withdrawn becomes ₹20,000.

### 6. TTL Inconsistency

- **What happens:** Time-to-live (expiration) is inconsistent across nodes.
- **Why it happens:** One node's TTL expired, another still holds the key.
- **Example:** Node A expires a product offer, Node B still serves it for a while.

### 7. Cache Stampede (Thundering Herd)

- **What happens:** Many clients try to read a missing or expired key at the same time, overloading the database.
- **Why it happens:** Expired data is not proactively refreshed or locked.
- **Not a direct inconsistency**, but can lead to **temporary inconsistent behavior** due to overload or fallback.

## Why is Consistency Important in Distributed Caching?

In a **distributed cache**, data is stored across **multiple nodes or servers**. If multiple copies of data exist, or data is updated frequently, we must ensure that:

- **Everyone sees the correct value** (not outdated or conflicting ones).
- The cache and the **original data source (like DB)** don't go out of sync.

## Key Techniques to Manage Consistency in Distributed Caches

### 1. Cache Invalidation

When the source data (e.g., database) changes, the corresponding cache entry must be **removed (invalidated)** or **updated**, so that stale data isn't served.

**Types of Invalidation:**

- **Manual Invalidation**: App explicitly deletes/updates cache key after DB write.

- **Auto Expiry (TTL)**: Each cache key expires after a time period.
- **Event-Driven (Pub/Sub)**: DB or app publishes a change event to update/invalidate cache in all nodes.

Helps prevent stale data
Risky if invalidation isn't reliable or is delayed

### 2. Write-Through Cache

- Data is **written to both the cache and the database** at the same time.
- Cache is always in sync with DB.

```
def update_profile(user_id, new_data):
    db.update(user_id, new_data)
    cache.set(user_id, new_data)
```

Strong consistency
Slightly slower writes

### 3. Write-Behind (Write-Back) Cache

- Data is **written to the cache first**, then **asynchronously written to DB** later.

Fast writes
Risk of data loss if cache crashes before syncing

### 4. Read-Through Cache

- App only reads from cache.
- If data is missing, the **cache fetches from DB automatically**, stores it, and returns.

Centralized cache control
First read is slow (cache miss)

### 5. Replication Consistency

In distributed caches, data may be **replicated** across multiple nodes for fault tolerance.

**Challenges:**

- If replication is **asynchronous**, some nodes may hold **stale copies**.
- Use **synchronous replication** or **quorum-based consistency** to ensure safety.

Prevents data loss
Can slow down writes

### 6. Eventual Consistency vs Strong Consistency

| Model | Description | Used When |
|---|---|---|
| **Strong Consistency** | All nodes see the same value immediately after a write | Financial systems, critical state |
| **Eventual Consistency** | Nodes may see different values temporarily; will sync later | High-performance apps (e.g., social media) |

Most distributed caches like Redis Cluster or Memcached go for **eventual consistency** to maximize speed.

### 7. Versioning / CAS (Check and Set)

- Every cache value is tagged with a **version number or timestamp**.
- When updating, the system checks the version — if changed, the update is rejected.

Prevents **lost updates** and **race conditions**

### 8. Distributed Locks / Semaphores

- When updating cache or DB, use a **distributed lock** (e.g., Redis Lock) to ensure only one writer at a time.

Avoids race conditions
Adds complexity and latency

### 9. Quorum-Based Read/Writes

- Use algorithms like **Raft** or **Paxos**, or systems like **Cassandra**, to ensure that:
    - Writes are successful only if a **majority of nodes acknowledge**
    - Reads return the **latest confirmed value**

Stronger guarantees
Slower and more resource-intensive

The **placement of a cache** relative to the server (or application) has a **major impact on speed, consistency, and scalability**. Let's explore how **close or far** a cache can be and what trade-offs come with each choice.

## Cache Placement Options: How Close or Far?

### 1. In-Process Cache (Closest)

- **Inside the application process** (e.g., in-memory using `HashMap`, `Guava Cache`, or `Caffeine` in Java).
- **Fastest access**: nanoseconds to microseconds.
- **Not shared** between instances.
- No fault tolerance—data is lost on app restart.

**Use When:**

- You need ultra-fast access.
- Data is small and app is single-node or doesn't require sharing cache across servers.

### 2. Local Node Cache

- Runs **on the same machine** as the application, but in a separate process.
- Very fast (microseconds to low milliseconds).
- Shared between multiple threads or processes on the same machine.
- Not visible to other machines in a cluster.

**Use When:**

- You want performance and isolation, but still need fast local access.

### 3. Remote Distributed Cache (Clustered)

- Cache is **hosted on separate servers** (can be nearby or far in the network).
- Tools: **Redis**, **Memcached**, **Hazelcast**, **Apache Ignite**, etc.
- Access times: low to mid milliseconds (depending on network distance).
- Shared across multiple application servers.
- Scalable and fault-tolerant.
- Slightly slower due to network latency.

**Use When:**

- You run multiple application nodes (microservices, cloud-native systems).
- You want **scalability, high availability, and replication**.

### 4. Geo-Distributed (Far-Edge) Cache

- Cache is placed **in data centers across the globe**.
- Used to serve users **closer to their geographic location**.
- Example: **CDNs (Content Delivery Networks)** like Cloudflare, Akamai.
- Latency depends on distance from user — still faster than going to origin DB/server.

**Use When:**

- You have **global users**.
- You're caching static assets or rarely changing data (e.g., images, product catalogs).
- You want **low latency for global access**.

CACHE POLICIES

## What Are Cache Policies?

Cache policies are **strategies or rules** used to:

1. **Decide what to store in cache**
2. **Determine how long to keep it**
3. **Choose what to evict when the cache is full**

These help improve **performance**, **memory efficiency**, and **data freshness**.

## Types of Cache Policies

### 1. Eviction Policies

Used when the cache is **full** and a new item needs to be added.

#### a) LRU (Least Recently Used)

- Evicts the **least recently accessed** item.
- Assumes recently used data is likely to be used again soon.

Example: If you haven't opened a chat in 2 weeks, it gets evicted before the one you just opened.

#### b) LFU (Least Frequently Used)

- Removes the item that has been used the **least number of times**.

Example: A news article read only once gets evicted before one read 50 times.

#### c) FIFO (First In First Out)

- Evicts the **oldest inserted item**, regardless of usage.

Simple but may evict frequently accessed items unfairly.

#### d) Random Replacement

- Removes a **random item**.
- Fast, but not very intelligent.

### 2. Expiration Policies

Control **how long an item stays in cache**.

#### a) TTL (Time to Live)

- Each item has an expiration time.
- After that, it is **automatically removed**.

Example: Cache a weather report for 10 minutes.

#### b) Idle Timeout (Sliding Expiration)

- Item expires if **not accessed for a certain time**, even if TTL hasn't hit.

Example: An item expires 5 minutes after last access.

## 3. Write Policies

Define how data is written to cache and backend storage.

#### a) Write-Through

- Write to cache and database **simultaneously**.
- Safe, but slower.

#### b) Write-Behind (Write-Back)

- Write to cache **first**, then sync to DB **later** (async).
- Faster, but risky.

#### c) Write-Around

- Writes **go directly to DB**, not cache.
- Cache is updated only on read.
- Prevents pollution of cache with rarely-used data.

## 4. Read Policies

Control how reads are handled.

#### a) Read-Through

- App reads from cache.
- If not found, **cache fetches from DB**, stores it, and returns it.

#### b) Read-Around

- App checks cache.

- If not found, it **reads directly from DB**, but **does not update the cache**.

### Best Practices

- Use **LRU** for user sessions or recent activity.
- Use **TTL** for time-sensitive data like API responses or news.
- Use **Write-Through** for critical data consistency.
- Use **LFU** for data with long-term popularity (e.g., search queries).

| Goal | Use This Policy |
|---|---|
| Reduce DB load | Read-through + TTL |
| Avoid stale data | Write-through + TTL |
| Improve latency | LRU with in-memory caching |
| Critical data sync | Write-through or Write-behind |
| Rarely-read data | Write-around to avoid polluting cache |

## What is a Global Cache?

A **global cache** is a **shared cache system** that is **accessible to multiple applications, servers, or services**, often across **different regions or zones**. It provides a **centralized caching layer** for the whole infrastructure.

**Think of It Like:**

Instead of each app or server having its **own separate cache**, you have **one big cache** that **everyone can read/write** to.

## Key Characteristics of a Global Cache:

| Feature | Description |
|---|---|
| **Shared** | Used by multiple app instances or microservices |
| **Distributed** | Often deployed across multiple data centers or nodes |
| **Consistent** | Needs to manage consistency and replication |
| **Centralized or Geo-distributed** | Can be in one region or span globally |

## Example:

### E-commerce Platform

- You have services like **Product Catalog**, **Pricing**, **Inventory**, and **User Sessions**.
- All services **access the same global cache** to retrieve product details, stock levels, etc.
- This reduces duplicate cache entries and improves **coordination between services**.

## How is a Global Cache Created?

A **global cache** is created by using a **distributed caching system** (like Redis Enterprise, Hazelcast, AWS ElastiCache Global Datastore) that:

1. **Runs across multiple nodes or regions** (data centers)
2. **Shares a common dataset** across these locations
3. Uses **replication and sharding** to balance and distribute data

### Example Architecture

Let's say you use **Redis Global Datastore**:

- You have one **primary region** (e.g., Mumbai 🇮🇳)
- You add **replica regions** (e.g., Frankfurt 🇩🇪, Singapore 🇸🇬)
- All regions can **read**, and only the primary **writes**
- Data is **replicated asynchronously** to replicas

## Why Was There a Need for Global Cache?

### Problem with Local Caches:

| Issue | Explanation |
|---|---|
| Multi-region latency | Users far from your server experience delays |
| Data fragmentation | Same data cached multiple times across services |
| Inconsistent views | One node shows old data, another shows new |
| Duplicate work | Multiple services cache same DB queries separately |

### Why Global Cache Solves It:

| Benefit | How It Helps |
|---|---|
| Faster global access | Users in Europe get data from a nearby cache (not India) |

| | |
|---|---|
| Consistent shared data | Product info, stock levels stay in sync |
| Reduced DB load | Everyone shares the same cache instead of hitting DB |
| Easier coordination | One source of truth for session, product, config, etc. |

## When to Use Global Cache?

| Use Case | Why Use Global Cache |
|---|---|
| Microservices | Shared data across services |
| Multi-region apps | Keep users in sync globally |
| Session management | One place to store user sessions |
| Cross-app coordination | Avoid cache duplication between apps |

## Challenges of Global Caching:

| Challenge | Description |
|---|---|
| Consistency | Hard to keep all nodes perfectly in sync |

| | |
|---|---|
| Latency | Accessing global cache from far regions adds delay |
| Partitioning | Needs careful sharding or partitioning logic |
| Cost | Global infrastructure and data replication are expensive |

## How Are Conflicts Solved in Global Caching?

This is the **hard part** in distributed systems. Let's break it down:

### Types of Conflicts:

1. **Concurrent Writes** to the same key from different regions
2. **Replication Delays** causing stale reads
3. **Split Brain**: Network partition causes two nodes to think they are the leader

### Strategies to Solve Conflicts:

| Strategy | How It Works |
|---|---|
| **Single Writer Principle** | Only **one region (primary)** is allowed to write; others are read-only |
| **Last Write Wins (LWW)** | Each value is timestamped; latest timestamp wins |
| **Distributed Locks** | Lock key before write using tools like **Redis Redlock** |
| **Quorum Writes** | Only update cache if **majority of nodes agree** (used in Cassandra, Hazelcast) |

| | |
|---|---|
| **Versioning (CAS)** | Use version numbers to detect & avoid overwrite conflicts |
| **Pub/Sub or Change Data Capture** | Services subscribe to changes and update cache accordingly |

### Example:

Imagine a global shopping app:

- **User in India** adds item to cart — write goes to India region cache.
- **User travels to Dubai**, continues shopping — cart is served from the **nearest replica**.
- All updates are **replicated from India to Dubai** behind the scenes.
- If both India and Dubai try to update the same key, only **India's update wins** (single-writer region).

EOT