**[DevOps](https://www.geeksforgeeks.org/courses/devops-live) is like a teamwork approach for making and delivering computer programs**. It brings together the folks who create the **software** (**Development**) and the ones who make sure it runs **smoothly** (**Operations**). The main aim of **DevOps** is to make the process of building and delivering software faster and better. It's like a bridge that helps these two teams work together more easily, so the software is **created**, **tested**, and **delivered quickly** and without problems.

The goals of DevOps are:

* Faster and continuous software releases.
* Reduces manual errors through automation.
* Built-in Monitoring**,** detect failures or issues during delivery or in production.
* Automate testing throughout the software delivery process.

CI/CD stands for Continuous Integration and Continuous Deployment/Delivery. It is a core DevOps practice that automates the process of building, testing, and deploying code changes to production faster and more reliably.

CALMS is an acronym that stands for culture, automation, lean, measurement and sharing.

Learn continue  
eliminating the unwanted   
automation   
nurturing culture

Containerization is a method of packaging an application with all its dependencies (libraries, configuration files, runtime) into a single, portable unit called a Container. These containers run consistently across environments, whether on a developer’s laptop, a test server, or in production, without compatibility issues.

**5. DevOps in Data Platforms (1.5 min)**

Since you're in a data platform context, highlight:

* **DataOps**: Applying DevOps to data engineering.
* **Pipeline Automation**: ETL/ELT workflows with Airflow, dbt.
* **Testing Data**: Schema validation, data quality checks.
* **Observability**: Monitoring data freshness, lineage, and SLAs.
* **Security & Governance**: Integrating compliance checks into CI/CD.

**6. Challenges & Anti-Patterns (1 min)**

* **Tool Overload**: Too many tools, not enough integration.
* **Lack of Culture Shift**: DevOps fails without collaboration.
* **Ignoring Feedback Loops**: No monitoring = no improvement.
* **Manual Processes**: Automation is key, but often neglected.

**7. Conclusion & Call to Action (1 min)**

“DevOps is a journey, not a destination. Whether you're building microservices or managing data pipelines, the principles of DevOps—collaboration, automation, and continuous improvement—apply universally. Let’s focus on outcomes, not just tools.”

## 1. What is DevOps?

DevOps is a **collaborative approach** that integrates software development (Dev) and IT operations (Ops) to **accelerate delivery**, **improve quality**, and **enhance reliability** of software systems.

DevOps is a **collaborative methodology** that integrates:

* **Development**
* **Operations**
* **Networking**
* **Security**

Its goal is to **streamline software delivery**, improve **system reliability**, and foster a culture of **continuous improvement**.

**🔗 Key Objectives**

* **Speed**: Faster and more frequent releases.
* **Quality**: Automated testing and monitoring.
* **Reliability**: Stable deployments and quick recovery.
* **Collaboration**: Breaking silos between teams.

The goals of DevOps are:

* Faster and continuous software releases.
* Reduces manual errors through automation.
* Built-in Monitoring**,** detect failures or issues during delivery or in production.
* Automate testing throughout the software delivery process.

**3. CI/CD – The Heart of DevOps**

**CI/CD** stands for:

* **Continuous Integration**: Developers merge code frequently; automated builds and tests ensure stability.
* **Continuous Delivery/Deployment**: Code is automatically prepared for release or deployed to production.

**🛠 Benefits:**

* Early bug detection
* Faster feedback loops
* Reduced integration issues

**4. CALMS Framework**

A model to assess DevOps maturity:

* **C**ulture: Collaboration, trust, and shared responsibility.
* **A**utomation: Tools and scripts to streamline processes.
* **L**ean: Eliminate waste, focus on value.
* **M**easurement: Track metrics for improvement.
* **S**haring: Knowledge, tools, and practices across teams.

**5. DevOps in Data Platforms (1.5 min)**

Applying DevOps principles to data engineering—**DataOps**:

* 🔄 **Pipeline Automation**: ETL/ELT workflows using tools like **Apache Airflow**, **dbt**.
* 🧪 **Data Testing**: Schema validation, data quality checks.
* 📈 **Observability**: Monitor data freshness, lineage, SLAs.
* 🔐 **Security & Governance**: Embed compliance checks in CI/CD.

**6. Challenges & Anti-Patterns (1 min)**

* 🧩 **Tool Overload**: Too many tools, poor integration.
* 🧱 **Lack of Culture Shift**: DevOps fails without collaboration.
* 🔁 **Ignoring Feedback Loops**: No monitoring = no learning.
* 🖐️ **Manual Processes**: Automation is often neglected.

**7. Conclusion & Call to Action (1 min)**

“DevOps is a journey, not a destination. Whether you're building microservices or managing data pipelines, the principles of DevOps—**collaboration, automation, and continuous improvement**—apply universally. Let’s focus on **outcomes**, not just tools.”

**✅ Additional Points You Can Add**

* **Infrastructure as Code (IaC)**: Tools like Terraform and Ansible for managing infrastructure.
* **DevSecOps**: Integrating security into DevOps workflows.
* **Shift-Left Testing**: Testing earlier in the development cycle.
* **Blue-Green & Canary Deployments**: Safer release strategies.

**Example of DevOps Workflow**

**Scenario: Building and Deploying a Web Application**

1. **Development Phase**
   * Developers write code and push changes to a shared repository.
   * **Tool**: Git (version control), GitHub/GitLab/Bitbucket
2. **Continuous Integration (CI)**
   * Code is automatically built and tested when changes are pushed.
   * **Tool**: Jenkins, GitHub Actions, GitLab CI, CircleCI
3. **Continuous Delivery/Deployment (CD)**
   * After successful tests, code is deployed to staging or production.
   * **Tool**: Spinnaker, ArgoCD, Harness, Octopus Deploy
4. **Infrastructure as Code (IaC)**
   * Infrastructure is provisioned automatically using code.
   * **Tool**: Terraform, AWS CloudFormation, Pulumi
5. **Containerization**
   * Application is packaged into containers for consistency across environments.
   * **Tool**: Docker
6. **Orchestration**
   * Containers are managed and scaled automatically.
   * **Tool**: Kubernetes, OpenShift
7. **Monitoring & Logging**
   * System health, performance, and errors are tracked.
   * **Tool**: Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), Datadog
8. **Security Integration (DevSecOps)**
   * Security checks are embedded in CI/CD pipelines.
   * **Tool**: Snyk, Aqua Security, SonarQube, HashiCorp Vault
9. **Networking**
   * Load balancing, service discovery, and traffic routing.
   * **Tool**: NGINX, Istio (service mesh), HAProxy

**🧰 Popular DevOps Platforms & Tools by Category**

| **Category** | **Tools/Platforms** |
| --- | --- |
| **Version Control** | Git, GitHub, GitLab, Bitbucket |
| **CI/CD** | Jenkins, GitHub Actions, GitLab CI, CircleCI, Travis CI |
| **IaC** | Terraform, AWS CloudFormation, Ansible, Pulumi |
| **Containers** | Docker, Podman |
| **Orchestration** | Kubernetes, OpenShift, Docker Swarm |
| **Monitoring** | Prometheus, Grafana, ELK Stack, Datadog, New Relic |
| **Security** | Snyk, SonarQube, Aqua Security, HashiCorp Vault |
| **Networking** | NGINX, Istio, HAProxy, Traefik |
| **Artifact Repositories** | JFrog Artifactory, Nexus Repository, GitHub Packages |
| **Collaboration** | Slack, Microsoft Teams, Confluence, Jira |

**Infrastructure as Code (IaC)** is a DevOps practice where **infrastructure is provisioned and managed using code**, rather than manual processes. This allows teams to automate, version, and replicate infrastructure reliably across environments.

* ✅ **Automation**: Reduces manual setup and errors.
* 🔁 **Consistency**: Same configuration across dev, test, and prod.
* 📦 **Version Control**: Infrastructure changes are tracked like code.
* 🚀 **Speed**: Faster provisioning and scaling.
* 💰 **Cost Efficiency**: Optimized resource usage and reduced downtime.

**Cloud Agnostic** refers to the ability of a system, application, or infrastructure to **run on any cloud provider** without being tightly coupled to one specific platform like AWS, Azure, or Google Cloud.

**Terraform** is an **open-source Infrastructure as Code (IaC)** tool developed by **HashiCorp**. It allows you to **define, provision, and manage cloud infrastructure** using a declarative configuration language called **HCL (HashiCorp Configuration Language)**.

Terraform supports **multiple cloud providers** through its **provider plugins**, making it **cloud agnostic**. This means you can use **one tool and one language** to manage infrastructure across **different cloud platforms**, avoiding vendor lock-in.

With Terraform, you write code to:

* Create servers, databases, networks, storage, etc.
* Manage dependencies and relationships between resources.
* Automate provisioning and updates.

**✅ Benefits:**

* **Version-controlled infrastructure** (like Git for infra).
* **Repeatable deployments** across environments.
* **Automated rollbacks and updates**.
* **Auditability** and **collaboration**.

**Supported Providers Include:**

* **AWS**
* **Azure**
* **Google Cloud Platform (GCP)**
* **Oracle Cloud**
* **VMware**
* **Kubernetes**
* **Alibaba Cloud**
* **DigitalOcean**
* **On-prem tools** like vSphere, Docker, etc.

Terraform provides state management, Terraform uses a **state file** (terraform.tfstate) to **track the current state of your infrastructure**. This file acts as a **snapshot** of what resources Terraform has created, modified, or destroyed.

Allows users to keep track of the current state of the infrastructure and manage its lifecycle

Terraform can be used for both CI/CD Integration both.

A lambda function in python is a **small, one-line function** defined using the lambda keyword. It’s often used when you need a simple function for a short period and don’t want to formally define it using def.

**AWS CloudFormation** is Amazon Web Services’ **native Infrastructure as Code (IaC)** tool that allows you to **define and provision AWS infrastructure** using **YAML or JSON templates**.

CloudFormation lets you:

* Describe your infrastructure as **code**.
* Automate the **creation, update, and deletion** of AWS resources.
* Use **templates** written in YAML or JSON to define resources like EC2, S3, RDS, IAM, etc.

**Why Use YAML in CloudFormation?**

* **Human-readable** and cleaner than JSON.
* Easier to manage **nested structures**.
* Widely supported in DevOps tools and CI/CD pipelines.

**AWS Lambda** is a **serverless compute service** provided by Amazon Web Services (AWS). It lets you **run code without provisioning or managing servers**. You simply upload your function code, and Lambda takes care of everything else—scaling, patching, and running it in response to events.

**Scaling** refers to adjusting compute capacity based on demand.

**🔄 Types:**

* **Vertical Scaling**: Increase CPU/RAM of a single instance.
* **Horizontal Scaling**: Add more instances (e.g., more pods in Kubernetes).

**Patching** involves applying updates to fix bugs, security vulnerabilities, or improve performance.

**🔧 Approaches:**

* **Manual Patching**: Admins apply updates manually.
* **Automated Patching**: Scheduled or triggered updates via tools.

**Upgrading** means moving to a newer version of software, OS, or platform components.

**🔄 Examples:**

* Upgrading OS (e.g., Ubuntu 20.04 → 22.04)
* Upgrading Kubernetes clusters
* Upgrading application versions

A **RESTful application** is a web application that follows the principles of **REST (Representational State Transfer)** — a software architectural style used for designing **networked applications**, especially **web APIs**.

**Flask** is a lightweight and flexible **Python web framework** that is commonly used to build **RESTful applications and APIs**. It’s ideal for small to medium-sized projects and is known for its simplicity and ease of use.

**What Is Jenkins?**

Jenkins is an **open-source automation server** used for:

* **Continuous Integration (CI)**: Automatically building and testing code.
* **Continuous Delivery/Deployment (CD)**: Automating release processes.

**🐳 What Is Docker?**

Docker is a platform that allows you to **package applications and their dependencies** into lightweight, portable containers that run consistently across environments.

**Jenkins with Docker** refers to using **Docker containers** to run Jenkins itself, or to run **Jenkins jobs inside containers**. This combination is powerful in DevOps because it brings together **automation (Jenkins)** and **containerization (Docker)** for scalable, portable, and efficient CI/CD pipelines.

**What Is Jenkins and How Does It Operate?**

**Jenkins** is an **open-source automation server** used primarily for **Continuous Integration (CI)** and **Continuous Delivery/Deployment (CD)** in DevOps. It helps automate the process of building, testing, and deploying software, making development faster and more reliable.

**🧠 How Jenkins Operates**

Jenkins works by executing **jobs** or **pipelines** that define a series of steps to perform tasks like compiling code, running tests, packaging applications, and deploying them.

**🔄 Basic Workflow:**

1. **Code Commit**  
   A developer pushes code to a version control system (e.g., GitHub).
2. **Trigger**  
   Jenkins detects the change (via webhook or polling) and triggers a job.
3. **Build**  
   Jenkins pulls the latest code and compiles it.
4. **Test**  
   Automated tests are run to verify the code.
5. **Deploy**  
   If tests pass, Jenkins deploys the application to staging or production.
6. **Feedback**  
   Jenkins provides logs, reports, and notifications (e.g., via email or Slack).

Jenkins is an **automation tool** that works **with Git** and other tools to **automate tasks** like building, testing, and deploying your code.

**Typical Jenkins Workflow**

1. **Connect to a Git Repository**
   * Jenkins pulls code from GitHub, GitLab, Bitbucket, etc.
   * It does **not** store or edit code itself.
2. **Run a Pipeline**
   * Jenkins executes a series of steps defined in a Jenkinsfile:
     + Build the code
     + Run tests
     + Package the application
     + Deploy to staging or production
3. **Provide Feedback**
   * Jenkins shows logs, test results, and deployment status.
   * It can send notifications (email, Slack, etc.).

**Jira Workflow with Jenkins & Terraform Integration**

**1. Ticket Creation (Jira)**

* **Jira:** Ticket is created with requirements.
* **Jenkins/Terraform:** No action yet.

**2. Development Phase**

* **Jira:** Ticket moves to "In Progress".
* **Jenkins:**
  + Developers push code to Git (e.g., GitHub/GitLab).
  + Jenkins is triggered via webhook or polling.
  + Jenkins runs build jobs, unit tests, and static code analysis.
* **Terraform:**
  + If infrastructure changes are needed, Terraform code is written and committed.
  + Jenkins can run terraform plan to preview changes.

**3. Testing Phase**

* **Jira:** Ticket moves to "In QA".
* **Jenkins:**
  + Executes automated test suites (e.g., Selenium, JUnit).
  + Generates test reports and artifacts.
  + Can notify Jira via plugins (e.g., Jira plugin for Jenkins).
* **Terraform:**
  + Jenkins may run terraform apply in a staging environment to test infrastructure changes.

**4. UAT Phase**

* **Jira:** Ticket moves to "Ready for UAT".
* **Jenkins:**
  + Deploys application to UAT environment.
  + Can use blue-green or canary deployments.
* **Terraform:**
  + Manages UAT infrastructure provisioning.

**5. Deployment Phase**

* **Jira:** Ticket moves to "Ready for Release" → "Deployed".
* **Jenkins:**
  + Executes production deployment pipeline.
  + Includes build, test, artifact upload, and deployment steps.
  + Sends deployment status back to Jira.
* **Terraform:**
  + Jenkins runs terraform apply to provision/update production infrastructure.
  + Ensures infrastructure-as-code consistency.

**6. Closure**

* **Jira:** Ticket is verified and closed.
* **Jenkins/Terraform:** Logs and metrics are archived; monitoring tools may be triggered.

A **data warehouse** is a centralized system designed to store, manage, and analyze large volumes of structured data from multiple sources. It’s optimized for **querying and reporting**, not for transaction processing.

**🏢 1. On-Premises Data Warehouse**

**🔹 Architecture:**

* Hosted on physical servers within the organization's data center.
* Managed by internal IT teams.
* Data is stored in relational databases (e.g., Oracle, SQL Server).
* ETL tools are used to ingest and transform data.

**🔧 Technologies:**

* Oracle Exadata
* IBM Netezza
* Microsoft SQL Server DW
* Informatica, Talend (ETL tools)

**✅ Advantages:**

* Full control over data, security, and compliance.
* Custom hardware and configurations.
* Suitable for industries with strict data residency laws (e.g., banking, healthcare).

**❌ Limitations:**

* High capital expenditure (CapEx).
* Limited scalability (vertical scaling).
* Long provisioning cycles.
* Manual upgrades and patching.

**🧠 Use Cases:**

* Legacy enterprise systems.
* Highly regulated environments.
* Organizations with existing infrastructure investments.

**🌐 2. Hybrid Data Warehouse**

**🔹 Architecture:**

* Combines on-premises and cloud components.
* Data may reside in both environments.
* ETL/ELT tools synchronize data across systems.
* Often used during cloud migration or for data locality needs.

**🔧 Technologies:**

* Azure Synapse (hybrid capabilities)
* Snowflake (can connect to on-prem sources)
* Informatica Intelligent Cloud Services
* VPNs or private links for secure data movement

**✅ Advantages:**

* Flexibility to keep sensitive data on-prem.
* Gradual cloud adoption.
* Optimized cost and performance.
* Can leverage cloud compute for analytics while storing data locally.

**❌ Limitations:**

* Complex architecture and integration.
* Requires strong governance and monitoring.
* Latency and synchronization challenges.

**🧠 Use Cases:**

* Enterprises transitioning to cloud.
* Multi-region operations.
* Compliance-driven industries needing data locality.

**☁️ 3. Cloud Data Warehouse**

**🔹 Architecture:**

* Fully hosted and managed by cloud providers.
* Serverless or managed infrastructure.
* ELT-friendly: raw data is loaded and transformed in-place.
* Scales horizontally based on demand.

**🔧 Technologies:**

* Snowflake
* Google BigQuery
* Amazon Redshift
* Azure Synapse Analytics
* dbt (for transformation)
* Fivetran, Stitch (for ingestion)

**✅ Advantages:**

* Elastic scalability and high availability.
* Pay-as-you-go pricing (OpEx).
* Fast provisioning and deployment.
* Integrated with modern analytics and ML tools.

**❌ Limitations:**

* Ongoing operational costs.
* Vendor lock-in risks.
* Requires cloud security best practices.

**🧠 Use Cases:**

* Real-time analytics and dashboards.
* Big data and machine learning workloads.
* Startups and agile enterprises.
* Global teams needing remote access.

**🧭 Summary Comparison**

| **Feature** | **On-Premises** | **Hybrid** | **Cloud** |
| --- | --- | --- | --- |
| **Deployment** | Local servers | Mixed | Cloud provider |
| **Scalability** | Limited | Moderate | Elastic |
| **Cost Model** | CapEx | Mixed | OpEx |
| **Maintenance** | Manual | Shared | Automated |
| **Security Control** | Full | Partial | Shared |
| **Speed to Deploy** | Slow | Medium | Fast |
| **Integration** | Complex | Moderate | Easy |
| **Best For** | Legacy systems | Transitional setups | Modern analytics |

**What Are ETL and ELT?**

Both are data integration processes used to move data from source systems (like databases, APIs, files) into a **data warehouse** or **data lake**.

**🧪 ETL: Extract, Transform, Load**

**🔹 Workflow:**

1. **Extract**: Pull data from source systems.
2. **Transform**: Clean, enrich, and reshape data in a staging area (outside the target system).
3. **Load**: Push the transformed data into the data warehouse.

**✅ Best For:**

* Traditional data warehouses (e.g., Oracle, SQL Server).
* Complex transformations.
* When compute resources are limited in the target system.

**🔧 Tools:**

* Informatica
* Talend
* Apache NiFi
* Azure Data Factory (can do ETL)
* AWS Glue

**⚡ ELT: Extract, Load, Transform**

**🔹 Workflow:**

1. **Extract**: Pull data from source systems.
2. **Load**: Push raw data directly into the data warehouse.
3. **Transform**: Use the data warehouse’s compute power to transform data in-place.

**✅ Best For:**

* Modern cloud data warehouses (e.g., Snowflake, BigQuery, Redshift).
* Large-scale data processing.
* Real-time or near-real-time analytics.

**🔧 Tools:**

* dbt (Data Build Tool)
* Fivetran
* Stitch
* Azure Synapse Pipelines

**🆚 ETL vs ELT: Detailed Comparison**

| **Feature** | **ETL** | **ELT** |
| --- | --- | --- |
| **Transformation Location** | External (before loading) | Internal (after loading) |
| **Performance** | Depends on external compute | Uses DW compute (faster in cloud) |
| **Latency** | Higher (due to staging) | Lower (faster load) |
| **Complexity** | More control over transformation | Simpler pipeline, but depends on DW |
| **Security** | Data transformed before entering DW | Raw data enters DW (needs governance) |
| **Use Case** | Legacy systems, compliance-heavy | Cloud-native, big data, real-time analytics |

**🧠 Real-World Example**

**ETL:**

* A bank extracts customer data from Oracle DB.
* Transforms it using Informatica (masking, joining, aggregating).
* Loads it into an on-prem data warehouse for reporting.

**ELT:**

* A startup uses Fivetran to load raw app data into Snowflake.
* dbt transforms the data using SQL models inside Snowflake.
* Dashboards are built in Looker or Power BI.

## What is a Data Lake?

A **data lake** is a centralized storage system designed to hold **raw data** in its native format—structured, semi-structured, and unstructured.

**🔹 Key Features:**

* Stores data at scale (e.g., logs, images, videos, sensor data).
* Schema-on-read: structure is applied when data is queried.
* Used for big data, machine learning, and analytics.
* Often built on cloud object storage (e.g., Amazon S3, Azure Data Lake Storage).

**☁️ What is Cloud Data Storage?**

**Cloud data storage** refers to storing any kind of data (files, databases, backups) in the cloud. It’s a **general-purpose** storage solution.

**🔹 Key Features:**

* Can store documents, media, backups, databases, etc.
* Not optimized for analytics or big data by default.
* Includes services like:
  + **Object storage** (e.g., S3, Blob Storage)
  + **Block storage** (e.g., EBS, Azure Disk)
  + **File storage** (e.g., Azure Files, Amazon EFS)

**📦 What is a Repository?**

A **repository** is a place where **code, data, or artifacts** are stored and managed—often with version control.

**🔹 Types of Repositories:**

* **Code Repositories**: GitHub, GitLab, Bitbucket
* **Artifact Repositories**: JFrog Artifactory, Azure Artifacts
* **Data Repositories**: For storing datasets (e.g., Kaggle, Dataverse)

**Managed cloud services** provide comprehensive management and support, making them ideal for businesses seeking ease of use and robust security.

You rent raw infrastructure (like virtual machines or storage), but **you manage everything**—OS, updates, security, scaling, backups, etc. example AWS EC2 (you manage the OS and software)

**✅ Pros:**

* Full control over configuration
* Customizable environments
* Suitable for legacy apps or specific compliance needs

**❌ Cons:**

* Requires skilled DevOps/IT teams
* Higher operational overhead
* Risk of misconfiguration or security gaps

**Unmanaged cloud services** offer greater control and customization at a potentially lower initial cost but require significant IT expertise and resources.

The cloud provider **manages the infrastructure and operations**—you just use the service. This includes updates, scaling, security, and availability.

**✅ Pros:**

* Faster deployment
* Lower maintenance
* Built-in scalability and security
* Ideal for modern DevOps and CI/CD workflows

**❌ Cons:**

* Less control over underlying infrastructure
* May have limitations on customization
* Potential vendor lock-in

Amazon redshift  
google bigQuery  
oracle Exadata  
azure data factory   
AWS Glue – ETL (extraction, transform and load)  
Talend – cloud service, flexibility, data integration, speed ELT, open source solution

**SAP** (Systems, Applications, and Products in Data Processing) is a leading enterprise software platform used by organizations to manage **business operations** and **customer relations**. SAP is widely used in large enterprises for **ERP (Enterprise Resource Planning)**, **CRM**, and **data analytics**.

**Deep SAP Integration** refers to the **tight and seamless connection** between SAP systems and other platforms, tools, or services—especially in **cloud**, **DevOps**, and **data ecosystems**.

### 📊 What is Data Visualization?

**Data visualization** is the process of converting **raw data** into **visual formats** like charts, graphs, maps, and dashboards to make the data easier to understand, analyze, and communicate.

* **Simplifies complex data**
* **Reveals patterns, trends, and outliers**
* **Supports decision-making**
* **Improves storytelling with data**

**🔧 Tools for Data Visualization**

* **Power BI**, **Tableau**, **Looker** – Business Intelligence platforms
* **Excel**, **Google Sheets** – Basic charting
* **Python (Matplotlib, Seaborn, Plotly)** – Custom visualizations

### 📈 What is Data Extrapolation in DevOps?

**Data extrapolation** in DevOps refers to the process of using **existing data trends** to **predict future outcomes** or **estimate unknown values**. It’s a form of **predictive analytics** that helps DevOps teams make proactive decisions.

# **🐧 What is Linux?**

**Linux** is an open-source operating system, just like Windows or macOS. It manages your computer's hardware and software, allowing you to run programs, manage files, and connect to networks.

* **Command-line friendly**: While it has graphical interfaces, many tasks are done via the **terminal**.

## 💻 What is the Command Prompt?

The **Command Prompt** (also called **Terminal** in Linux/macOS) is a text-based interface where you type commands to interact with the operating system.

Think of it as talking directly to the computer in its own language.

#### 🔹 **Linux Kernel**

* The **kernel** is the **core** of the Linux operating system.
* It directly interacts with **hardware**: CPU, memory, disk, network, etc.
* It provides services to programs via **system calls**.
* You typically don’t interact with the kernel directly unless you're doing **low-level programming** (e.g., writing device drivers or kernel modules).
* Written mostly in **C**.

#### 🔹 **Text Editors (**vi**,** nano**,** vim**, etc.)**

* These are **user-space applications**.
* They run **on top of the kernel**, not inside it.
* You use them to **write and edit code**, scripts, config files, etc.
* They don’t interact with hardware directly — they rely on the kernel to do that.

**🔍 File & Directory Navigation**

* pwd – Show current directory
* ls – List files in a directory
* cd – Change directory
* mkdir – Create a new folder
* rmdir – Remove a folder

**📁 File Operations**

* touch – Create a new empty file
* cp – Copy files
* mv – Move or rename files
* rm – Delete files
* cat – View file contents
* nano, vim – Edit files in terminal

**🔎 Searching & Filtering**

* grep – Search text in files
* find – Locate files
* locate – Quickly find files (uses a database)

**📦 Package Management (depends on distro)**

* apt – For Debian/Ubuntu (sudo apt install <package>)
* yum – For CentOS/RHEL
* dnf – For Fedora

**🛠️ System Info & Management**

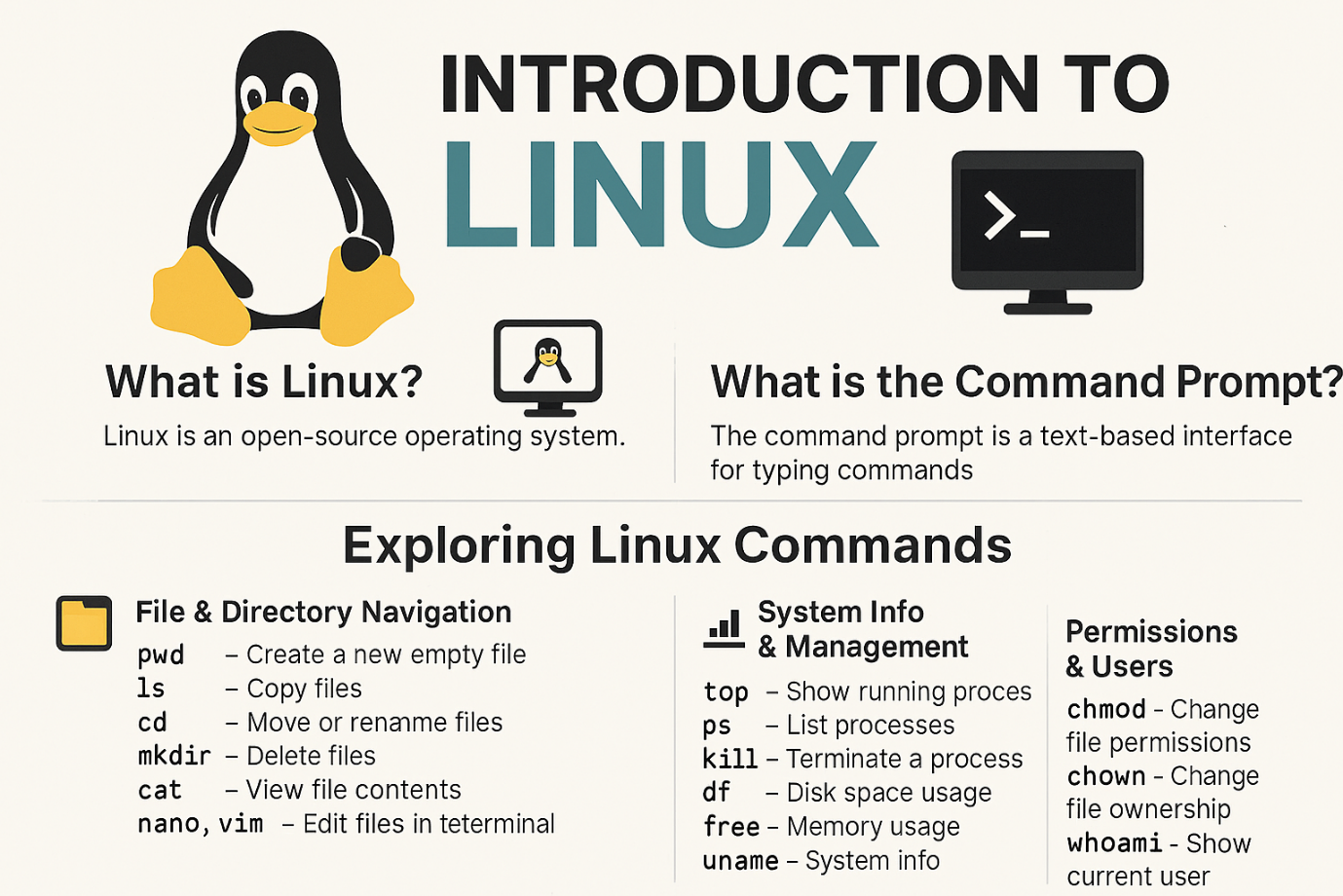
* top – Show running processes
* ps – List processes
* kill – Terminate a process
* df – Disk space usage
* free – Memory usage
* uname – System info

**🔐 Permissions & Users**

* chmod – Change file permissions
* chown – Change file ownership
* whoami – Show current user
* sudo – Run command as superuser

**🌐 Networking**

* ping – Test network connection
* ifconfig or ip – Show network interfaces
* netstat – Network statistics
* curl – Transfer data from/to a server



Test connectivity to a server: ping google.com

Show network interfaces and IP addresses: ip a

Shows the path packets take to reach a host: traceroute google.com

View network connections: netstat -tuln

ss -tuln

**🌐 VI editor**

The **vi editor** is a powerful text editor in Linux/Unix systems, operating in two primary modes: **command mode** (for navigation and editing) and **insert mode** (for text input). Below are some essential shortcuts to enhance your productivity.

**Basic Navigation**

h - Move left

j - Move down

k - Move up

l - Move right

w - Jump forward by a word

b - Jump backward by a word

0 - Jump to the beginning of the line

$ - Jump to the end of the line

gg - Go to the beginning of the file

G - Go to the end of the file

**Editing Commands**

i - Insert before the cursor

a - Append after the cursor

o - Open a new line below

dd - Delete the current line

yy - Yank (copy) the current line

p - Paste below the cursor

u - Undo last change

Ctrl + r - Redo last undone change

**Saving and Exiting**

:w - Save changes without exiting

:wq or :x - Save changes and exit

:q - Quit without saving (if no changes)

:q! - Force quit without saving changes

**Search and Replace**

/word - Search forward for "word"

?word - Search backward for "word"

n - Repeat search in the same direction

:%s/old/new/g - Replace all occurrences of "old" with "new"

:%s/old/new/gc - Replace with confirmation for each occurrence

**Visual Mode (Text Selection)**

v - Start visual mode for character selection

V - Start visual mode for line selection

Ctrl + v - Start block selection mode (visual block)

y - Yank (copy) selected text in visual mode

d - Delete selected text in visual mode

## 💻 Exercises

Count the number of words in notes.txt

wc -w notes.txt

**🔍 Explanation**

* wc = word count utility
* -w = tells it to count **words**
* notes.txt = the file you're analyzing

Search for the word “root” in /etc/passwd.

grep "root" notes.txt

**🔍 Explanation**

* grep = Global Regular Expression Print (used to search text)
* "root" = the word you're searching for
* notes.txt = the file you're searching in

Display the first 10 lines of /etc/passwd.

head -n <number> <filename>

**🔍 Explanation**

* -n → Specifies the number of lines to show.
* <number> → How many lines you want to display.
* <filename> → The file you want to read.

**🔍 simpler and easier Bash script** to rename all .txt files in a folder to .bak:

A white background with red text

Description automatically generated

### ✅ What It Does:

* Loops through all .txt files in the current folder.
* Uses mv to rename each file.
* ${file%.txt} removes the .txt extension.
* Adds .bak to the end.

## 📁 Linux Directory Structure Explained

### 1. / ****(Root Directory)****

* The top-level directory in Linux.
* Everything starts from here.
* All other directories are subdirectories of /.

### 2. /bin ****(Essential User Binaries)****

* Contains essential command-line utilities like ls, cp, mv, rm, cat, etc.
* These are needed for basic system functionality, even in single-user mode.

### 3. /boot ****(Boot Loader Files)****

* Contains files needed to boot the system, like the Linux kernel (vmlinuz) and bootloader config files (grub).
* Critical for system startup.

### 4. /dev ****(Device Files)****

* Contains files that represent hardware devices (e.g., /dev/sda for hard drives, /dev/tty for terminals).
* These are not regular files but interfaces to devices.

### 5. /etc ****(Configuration Files)****

* System-wide configuration files and shell scripts.
* Examples: /etc/passwd (user accounts), /etc/fstab (disk mounting), /etc/hostname.

### 6. /home ****(User Home Directories)****

* Each user gets a personal directory here, like /home/saketh.
* Contains user files, documents, downloads, etc.

### 7. /lib ****(Essential Shared Libraries)****

* Contains shared libraries needed by programs in /bin and /sbin.
* Similar to .dll files in Windows.

### 8. /media ****and**** /mnt ****(Mount Points)****

* Temporary mount points for external devices like USB drives or network shares.
* /media is often used for automatic mounting.

### 9. /opt ****(Optional Software Packages)****

* Used for installing third-party software.
* Example: /opt/google/chrome if you install Chrome manually.

### 10. /proc ****(Process Information)****

* Virtual filesystem that provides info about running processes and kernel.
* Example: /proc/cpuinfo, /proc/meminfo.

### 11. /root ****(Root User’s Home Directory)****

* Home directory for the root (superuser).
* Not to be confused with /, the root of the filesystem.

### 12. /run ****(Runtime Data)****

* Stores volatile runtime data like process IDs and system info.
* Cleared on reboot.

### 13. /sbin ****(System Binaries)****

* Contains system administration commands like shutdown, mount, fsck.
* Typically used by the root user.

### 14. /srv ****(Service Data)****

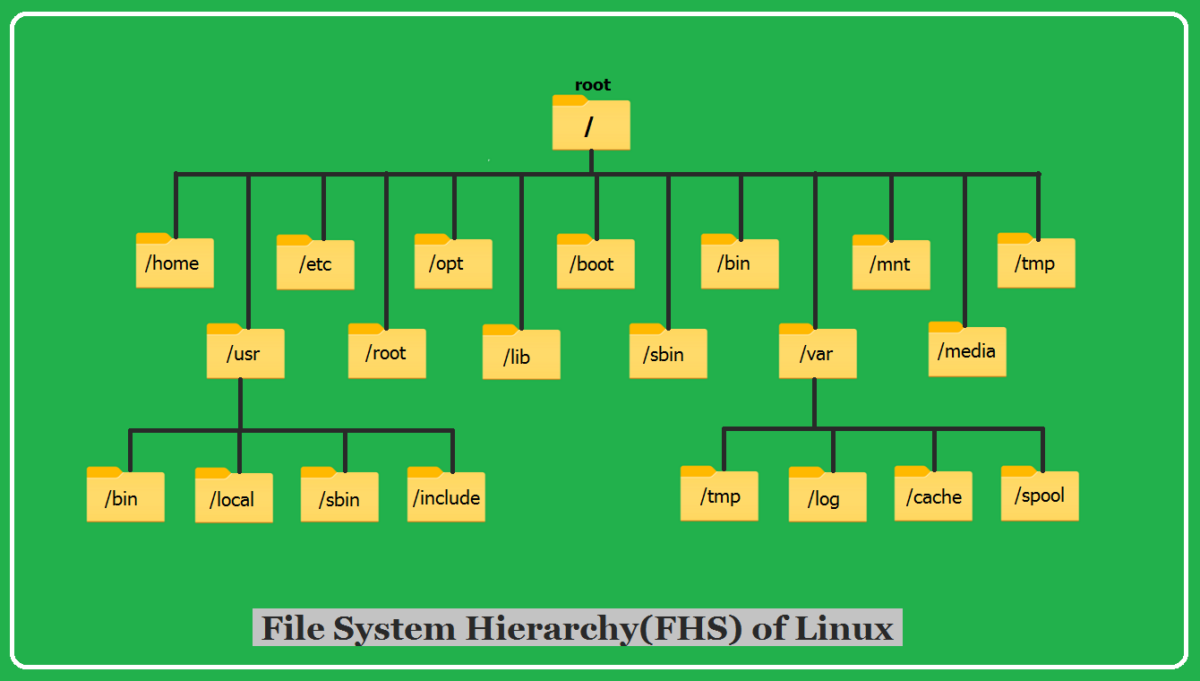
* Contains data for services like web servers or FTP.
* Example: /srv/www for web content.

### 15. /tmp ****(Temporary Files)****

* Used for temporary storage by applications.
* Often cleared on reboot.

### 16. /usr ****(User Programs and Data)****

* Contains user-installed software and libraries.
* Subdirectories:
  + /usr/bin: Non-essential user binaries
  + /usr/lib: Libraries
  + /usr/share: Shared data



The first character in the output indicates the file type:

* - → regular file
* d → directory
* l → symbolic link
* c → character device
* b → block device
* p → pipe
* s → socket

## 🧨 1. ****Shebang (****#! /bin/bash****)****

### What is it?

The **shebang** is the first line in a shell script that tells the system **which interpreter** to use to run the script.

## 🧾 3. ****Taking Input –**** read ****Command****

### Usage:

echo "Enter your name:"  
read name  
echo "Hello, $name!"

* read waits for user input and stores it in a variable.
* You can read multiple inputs too:

read first last  
echo "First: $first, Last: $last"

## 🧾 4. ****Current Date –**** date ****Command****

 date +"%T" → Time only (HH:MM:SS)

 date +"%A" → Day name (Thursday)

 date +"%B" → Month name (September)

## 🧾 5. **Top 5 largest files** in your **home directory**

A screenshot of a computer

Description automatically generated

## 🧾 6. **Top 5 largest files** in your **home directory**

du -ah ~ | sort -rh | head -n 5

## 🧠 Explanation:

### 🔧 du -ah ~

* du = Disk Usage
* -a = Show sizes of **all files and directories**
* -h = Human-readable format (KB, MB, GB)
* ~ = Your home directory

This lists the size of every file and folder inside your home directory.

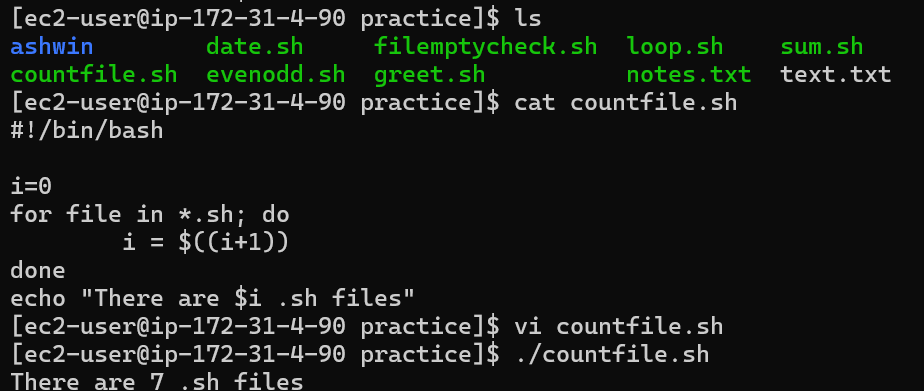
### 🔧 sort -rh

* sort = Sorts the output
* -r = Reverse order (largest first)
* -h = Human-readable sort (understands KB, MB, GB)

### 🔧 head -n 5

* Shows the **top 5** results from the sorted list.

## 🧾 7. Write a command to count the number of .sh scripts in the current directory



### ✅ 1. ls -1 \*.sh 2>/dev/null | wc -l

* ls -1 \*.sh: Lists all .sh files in the current directory, one per line.
* 2>/dev/null: Suppresses error messages (e.g., if no .sh files exist).
* | wc -l: Counts the number of lines (i.e., number of .sh files).

### ✅ 2. find . -maxdepth 1 -type f -name "\*.sh" | wc -l

* find .: Starts searching from the current directory.
* -maxdepth 1: Limits the search to the current directory only.
* -type f: Filters for regular files.
* -name "\*.sh": Matches files ending in .sh.
* | wc -l: Counts the number of matching files

## 🧾 8. Extract all usernames from /etc/passwd and save them to users.txt.

A black screen with white text

Description automatically generated

### ✅ How It Works:

* cat /etc/passwd: Reads the file line by line.
* cut -d':' -f1: Extracts the first field (username).
* echo "$username": Prints the username.
* >> users.txt: Appends the username to the file.

**✅ 1. Using cut**

**Shell**

cut -d: -f1 /etc/passwd > users.txt

**🔍 Explanation:**

* cut: A command used to extract specific fields from each line of a file.
* -d:: Sets the delimiter to a colon (:), which is used in /etc/passwd to separate fields.
* -f1: Selects the **first field**, which is the **username**.
* /etc/passwd: The file that contains user account information.
* >: Redirects the output.
* users.txt: The file where the usernames will be saved.

**✅ 2. Using awk (Alternative)**

**Shell**

awk -F*: '{print $1}'* /etc/passwd > users.txt

**🔍 Explanation:**

* awk: A powerful text-processing tool.
* -F:: Sets the field separator to a colon (:).
* {print $1}: Prints the first field (username).
* /etc/passwd: Source file.
* >: Redirects the output.
* users.txt: Destination file.

**9.Display Top 5 CPU-Consuming Processes**

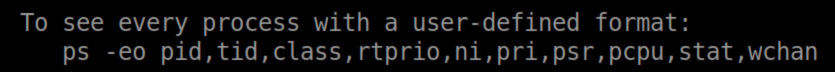
Use the ps command instead:

**Shell**

ps -eo pid,comm,%cpu --sort=-%cpu | head -n 6

**🔍 Explanation:**

* ps -eo pid,comm,%cpu: Lists process ID, command name, and CPU usage.
* --sort=-%cpu: Sorts by CPU usage in descending order.
* head -n 6: Shows top 5 processes **plus the header line**.



A computer screen with white text

Description automatically generated

**🧾 10.** **Write a script that automatically backs up a directory to a .tar.gz file with today’s date.**

**Shell**

*#!/bin/bash*

tar -czf backup\_$(date +%F).tar.gz /path/to/directory

**🧠 Explanation:**

**✅ Line-by-line:**

1. #!/bin/bash
   * This is the **shebang**. It tells the system to use the Bash shell to run the script.
2. tar -czf backup\_$(date +%F).tar.gz /path/to/directory
   * tar: A command used to archive files.
   * -c: Create a new archive.
   * -z: Compress the archive using gzip.
   * -f: Specify the filename of the archive.
   * backup\_$(date +%F).tar.gz: Creates a filename like backup\_2025-09-12.tar.gz using today’s date.
   * /path/to/directory: Replace this with the actual directory you want to back up.

**30. Write a script that monitors free disk space and warns if usage exceeds 80%**

A screenshot of a computer program

Description automatically generated

 df /: Shows disk usage statistics for the root directory.

 awk 'NR==2 {print $5}': Picks the **second line** of the output and extracts the **5th column**, which is the **percentage used** (e.g., 85%).

* awk is a powerful text-processing tool.
* NR==2 means: "If it's the second line of input..."
* {print $5} means: "...print the 5th column."
* In the example above, the 5th column is 80%, which is the disk usage percentage.

 sed 's/%//': Removes the % sign so we can compare it as a number.

* sed is a stream editor used to modify text.
* 's/%//' means: "Substitute (remove) the % character."
* So 80% becomes just 80, which is easier to compare as a number.

 usage=...: Stores the numeric value (e.g., 85) in the variable usage.

**31. Monitor the log files and directory**

A screen shot of a computer code

Description automatically generated

#### 🔹 inotifywait -m /path/to/dir -e create

* inotifywait: A command-line tool that waits for filesystem events.
* -m: Monitor mode — keeps running and watching continuously.
* /path/to/dir: Replace this with the actual directory you want to monitor.
* -e create: Only triggers when a **new file is created**.

#### 🔹 | while read path action file; do

* This reads the output from inotifywait line by line.
* Each line contains:
  + path: the directory where the event occurred
  + action: the type of event (e.g., CREATE)
  + file: the name of the new file

#### 🔹 echo "$(date): $file was created in $path" >> dir\_monitor.log

* Logs the event with a timestamp.
* Appends it to a file called dir\_monitor.log.

**34. Delete the files which are above 100 MB**

A black background with white text

Description automatically generated

* find: Searches for files and directories.
* /path/to/dir: The directory to search in.
* -type f: Limits the search to files only.
* -size +100M: Finds files larger than 100MB.
* -delete: Deletes the files that match the criteria.
* -print: This prints the full path of each file before it's deleted.
* -delete: Deletes the file after printing.

**35. Parse an Apache/Nginx access log and find the top 10 IP addresses**

A black background with white text

Description automatically generated

* Extracts the **first field** from each line of the log file.
* In Apache/Nginx logs, the first field is usually the **client IP address**.
* Sorts the IP addresses alphabetically.
* Required before using uniq -c to group identical IPs together.
* Counts how many times each unique IP appears.
* Output format: count IP\_address
* Sorts the results **numerically** (-n) and in **reverse order** (-r).
* So the IPs with the **highest request count** come first.

**36. Write a script that extracts email addresses from a text file**

A computer screen with a green line and white text

Description automatically generated

#### 🔍 What it does:

* grep: Searches for patterns in text.
* -E: Enables **extended regular expressions**.
  + -E tells grep to use **Extended Regular Expressions (ERE)**.
  + This allows you to use more powerful pattern matching features like:
  + + (one or more)
  + ? (zero or one)
  + () (grouping)
  + | (OR)
* -o: Prints **only the matching part** (not the whole line).
* "[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-z]{2,}": This is the **regex pattern** to match email addresses.
* file.txt: The file to search in.

#### 🧠 Regex Breakdown:

* [a-zA-Z0-9.\_%+-]+: Matches the **username** part of the email.
* @: Matches the **@** symbol.
* [a-zA-Z0-9.-]+: Matches the **domain name**.
* \.[a-z]{2,}: Matches the **top-level domain** (like .com, .org, etc.)

**36. Write a script to count how many times each shell (bash, zsh, etc.) is used in /etc/passwd**

A screenshot of a computer program

Description automatically generated

**🧠 Explanation**

1. **awk -F: '{print $7}' /etc/passwd**
   * awk: Text processing tool.
   * -F:: Sets the **field separator** to : (colon), which is how /etc/passwd is structured.
   * {print $7}: Prints the **7th field**, which is the user's **default shell** (e.g., /bin/bash, /bin/zsh, etc.).
2. **sort**
   * Sorts the list of shells alphabetically.
3. **uniq -c**
   * Counts how many times each unique shell appears.
4. **sort -nr**
   * Sorts the results **numerically** and in **reverse order**, so the most-used shells appear first.
5. cut -d: -f7: Cuts out the 7th field using : as the delimiter.
6. uniq -c: Counts occurrences.

37. Write a script to monitor failed login attempts from /var/log/auth.log.

### Permission to make it Executable:

chmod +x myscript.sh

### ✅ What is chmod?

chmod stands for **change mode**. It’s used to **change the permissions** of a file or directory.

### 🔤 Permission Flags

Linux permissions are represented by **three types of access** for three types of users:

A screenshot of a computer

Description automatically generated A screenshot of a list of categories

Description automatically generated

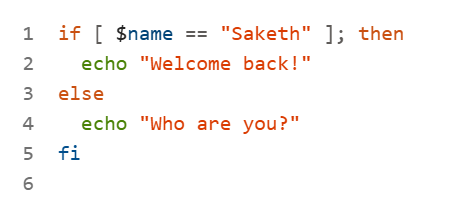
## 🧠 Bonus: Useful Bash Scripting Concepts

### 🔁 Loops

A white background with black and orange text

Description automatically generated

### 📋 Conditions



# **💻 What is Cloud ?**

**Cloud computing** is the delivery of computing services—like servers, storage, databases, networking, software, analytics, and intelligence—over the **internet** (“the cloud”) to offer faster innovation, flexible resources, and economies of scale.

Delivery of computing services over the internet, allowing for scalable resources and flexible management without physical infrastructure.

Physical servers are not feasible cause of the more starting capital and high maintenance.

Transition from CapEx to OpEx shift : capital expenditure to operational expenditure allows businesses to pay for only the resources they use, eliminating the large upfront investments in hardware.

Agility

Enhances responsiveness

Elasticity

Flexible resources

Cost saving through efficient resource

Global deployment

Reliability

Robust – cloud

Types of cloud : IaaS (infrastructure as service) (Amazon Ec2), PaaS (platform as a service) (Microsoft Azure App Services), SaaS (Software as a service) (Gmail, Microsoft 365)

### 🧾 ****Amazon EC2 Payment Types Explained****

Amazon EC2 offers multiple pricing models to help you optimize costs based on your workload type and usage patterns:

#### **1. On-Demand Instances**

* **What it is**: Pay for compute capacity by the hour or second (minimum 60 seconds).
* **Use case**: Best for short-term, unpredictable workloads or when you're testing or developing.
* **Pros**: No upfront payment, flexible.
* **Cons**: Most expensive option if used continuously.

#### **2. Savings Plans**

* **What it is**: Commit to a consistent amount of usage (e.g., $100/month) for 1 or 3 years.
* **Use case**: Ideal for steady workloads like web servers or databases.
* **Pros**: Up to **72% savings** compared to On-Demand.
* **Cons**: Requires commitment; less flexible.

#### **3. Spot Instances**

* **What it is**: Buy unused EC2 capacity at **up to 90% discount**.
* **Use case**: Great for fault-tolerant, flexible workloads like batch jobs, big data, or CI/CD.
* **Pros**: Very cheap.
* **Cons**: Can be interrupted by AWS with short notice.

#### **4. Dedicated Hosts**

* **What it is**: Physical servers dedicated to your use.
* **Use case**: Useful for licensing requirements (e.g., Windows Server, Oracle).
* **Pros**: Full control over hardware, compliance benefits.
* **Cons**: Higher cost, less flexibility.

#### **5. On-Demand Capacity Reservations**

* **What it is**: Reserve capacity in a specific Availability Zone.
* **Use case**: For critical workloads that must run when needed.
* **Pros**: Guaranteed capacity.
* **Cons**: You pay even if you don’t use the reserved capacity.

**🏢 Availability Zones (AZs)**

* **Definition**: AZs are isolated locations within a region that contain one or more data centers.
* **Purpose**: They provide **high availability** and **fault tolerance**. If one AZ goes down, others can take over.
* **Example**: The AWS Mumbai Region (ap-south-1) has multiple AZs like ap-south-1a, ap-south-1b, etc.
* **Use case**: You can deploy applications across multiple AZs to ensure redundancy and failover.
* If one AZ is down the another AZ will backup foe example if the AZ’s of Asia- Pacific if Mumbai AZ is failed then the Hyderabad AZ will support for continuous

**🌐 Edge Locations**

* **Definition**: These are **content delivery network (CDN)** endpoints used by **Amazon CloudFront** and other services.
* **Purpose**: They cache content closer to users to reduce latency and improve performance.
* **Example**: If a user in Pune accesses a website hosted in the US, the content can be served from an edge location in India.
* **Use case**: Ideal for static content, video streaming, and APIs that need fast response times.

AZs and edge locations is essential for designing applications that require fault tolerance, low latency.

AWS can be accessed by 3 modes: management console, command line, console

#### **1. AWS Management Console**

* **Type**: Web-based GUI (Graphical User Interface)
* **Use case**: Ideal for beginners or for visual management of services.
* **Features**:
  + Easy navigation
  + Dashboard views
  + Service wizards
* **Example**: Launching an EC2 instance via the browser.

#### **2. AWS Command Line Interface (CLI)**

* **Type**: Text-based command tool
* **Use case**: Great for automation, scripting, and quick access.
* **Features**:
  + Execute commands like aws ec2 describe-instances
  + Supports all AWS services
  + Can be used in shell scripts

#### **3. AWS SDKs / APIs**

* **Type**: Programmatic access via code
* **Use case**: For developers building apps that interact with AWS.
* **Features**:
  + SDKs available for Python (Boto3), Java, JavaScript, etc.
  + RESTful APIs for direct HTTP requests
  + Full control and automation

AWS login url: <https://ashwin-aws-try.signin.aws.amazon.com/console>

IAM username: team-1

Password: Saketh@team1

For creating another user (IAM) in the cloudshell:

aws iam create-user --user-name Team-1.2

aws iam create-login-profile \

  --user-name Team-1.1 \

  --password 'YourSecurePassword123!' \

  --password-reset-required

**🔍 What Each Part Means**

| **Part** | **Description** |
| --- | --- |
| aws iam create-login-profile | Tells AWS CLI to create a login profile for an IAM user. |
| --user-name Team-1.1 | Specifies the IAM user you're creating the login profile for. |
| --password 'YourSecurePassword123!' | Sets the initial password for the user to log in to the AWS Console. |
| --password-reset-required | Forces the user to change their password the first time they log in (recommended for security). |

**🌍 What is Terraform?**

**Terraform** is an **open-source Infrastructure as Code (IaC) tool** developed by HashiCorp. It allows you to **define, provision, and manage cloud infrastructure** using a **declarative configuration language** called **HCL (HashiCorp Configuration Language)**.

Terraform's modeling language, known as **HashiCorp Configuration Language (HCL)**, is a declarative language designed to define and manage infrastructure as code (IaC). It allows users to describe the desired state of infrastructure resources, and Terraform ensures that the actual state matches the defined configuration. HCL is both human-readable and machine-friendly, making it ideal for DevOps workflows.

**🧠 Key Concepts**

**✅ 1. Infrastructure as Code (IaC)**

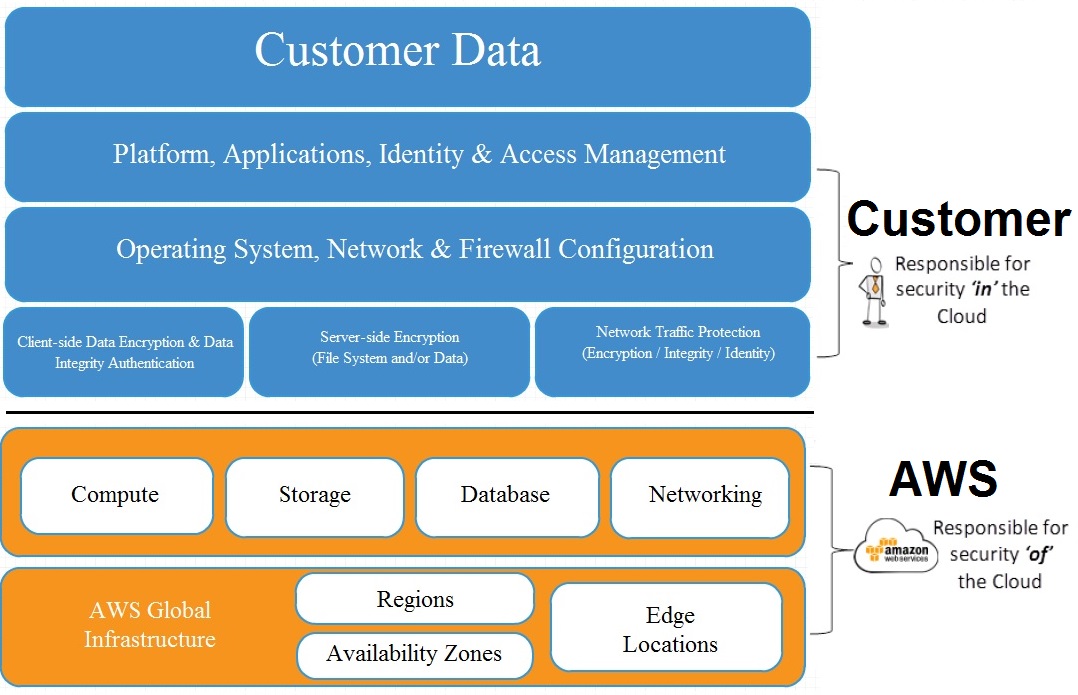
* You write code (in HCL) to describe your infrastructure.
* This code can be version-controlled, reused, and shared — just like application code.

**✅ 2. Declarative Language**

* You **declare** what resources you want (e.g., EC2 instance, S3 bucket).
* Terraform figures out **how** to create or update them.

**🛠️ What Can Terraform Do?**

* Create and manage resources in:
  + **AWS**
  + **Azure**
  + **Google Cloud**
  + **Kubernetes**
  + **VMware**, and many more.
* Automate:
  + EC2 instance creation
  + VPC setup
  + IAM roles and policies
  + S3 buckets
  + Load balancers
  + DNS records (via Route 53)



**🔐 1. AWS Shared Responsibility Model**

This is **fundamental** to cloud security. It defines **who is responsible for what** between AWS and **you (the customer)**.

**✅ AWS is responsible for:**

* **Security *of* the cloud**
  + Physical infrastructure (data centers)
  + Hardware, networking, and facilities
  + Global infrastructure (Regions, AZs, Edge Locations)

**✅ You are responsible for:**

* **Security *in* the cloud**
  + Configuring IAM roles and policies
  + Securing your EC2 instances, S3 buckets, databases
  + Managing encryption, patching, firewall rules

🔄 Think of it like renting an apartment: AWS secures the building, but you lock your own door.

**📜 2. AWS Compliance Programs**

AWS complies with **global standards and regulations** to help customers meet their own compliance needs.

**🔒 Examples of AWS Compliance Certifications:**

* **ISO 27001, 27017, 27018** – Information security standards
* **SOC 1, SOC 2, SOC 3** – System and organization controls
* **PCI DSS** – For handling credit card data
* **HIPAA** – For healthcare data in the U.S.
* **GDPR** – For data protection in the EU
* **FedRAMP** – For U.S. government cloud services

**✅ AWS provides:**

* **Audit reports**
* **Compliance documentation**
* **Artifact portal** for downloading compliance evidence

**🛡️ 3. Security Features in AWS**

AWS offers many built-in tools to help you secure your environment:

| **Feature** | **Purpose** |
| --- | --- |
| **IAM** | Manage users, roles, and permissions |
| **KMS** | Key Management Service for encryption |
| **CloudTrail** | Logs all API calls for auditing |
| **GuardDuty** | Threat detection and monitoring |
| **Security Hub** | Centralized security view |
| **WAF & Shield** | Protect against web attacks and DDoS |

**🖥️ 1. EC2 (Elastic Compute Cloud)**

* **Purpose**: Virtual servers in the cloud.
* **Use Case**: Run applications, host websites, backend services.
* **Key Feature**: You choose the OS, instance type, storage, and networking.
* **Analogy**: Like renting a virtual machine.

**📦 2. S3 (Simple Storage Service)**

* **Purpose**: Object storage for files, backups, media, etc.
* **Use Case**: Store images, videos, logs, static websites.
* **Key Feature**: Highly durable (99.999999999%) and scalable.
* **Analogy**: Like a cloud-based hard drive or file cabinet.

**☸️ 3. EKS (Elastic Kubernetes Service)**

* **Purpose**: Managed Kubernetes service.
* **Use Case**: Run containerized applications using Kubernetes without managing the control plane.
* **Key Feature**: AWS handles the Kubernetes master nodes, scaling, and security.
* **Analogy**: Like a container orchestration manager for your microservices.

**🤖 4. SageMaker**

* **Purpose**: Build, train, and deploy machine learning models at scale.
* **Use Case**: Data science, AI/ML workflows, model training and inference.
* **Key Feature**: Fully managed Jupyter notebooks, built-in algorithms, and model hosting.
* **Analogy**: Like a data scientist’s lab in the cloud.

**🗄️ 5. AWS Database Services (AWS DB)**

This includes **multiple database offerings**, such as:

| **Service** | **Type** | **Use Case** |
| --- | --- | --- |
| **RDS** | Relational (MySQL, PostgreSQL, etc.) | Traditional apps needing SQL |
| **DynamoDB** | NoSQL | High-speed, low-latency apps |
| **Aurora** | Relational (MySQL/PostgreSQL-compatible) | High-performance managed DB |
| **Redshift** | Data warehouse | Analytics and BI workloads |
| **DocumentDB** | Document-based (MongoDB-compatible) | JSON document storage |

**🌐 What is an Internet Gateway in AWS?**

An **Internet Gateway (IGW)** is a **networking component** in AWS that allows resources in your **VPC (Virtual Private Cloud)** to:

* **Connect to the internet**
* **Receive traffic from the internet**

It acts as a **bridge between your VPC and the public internet**.

Internet gateways provide target in your VPC route table for internet routable traffic

## 🌐 ****What is Amazon CloudFront?****

**Amazon CloudFront** is a **Content Delivery Network (CDN)** service offered by AWS. It helps deliver content (like websites, videos, APIs, and software) to users **quickly and securely**, by caching it at **Edge Locations** around the world.

**🗄️ AWS Managed Database Services**

**🔹 1. Amazon RDS (Relational Database Service)**

* **Type**: Relational (SQL)
* **Engines Supported**:
  + MySQL
  + PostgreSQL
  + MariaDB
  + Oracle
  + SQL Server
* **Features**: Automated backups, replication, scaling, patching.
* **Use Case**: Traditional applications needing structured data.

**🔹 2. Amazon Aurora**

* **Type**: Relational (MySQL/PostgreSQL-compatible)
* **Features**: High performance, auto-scaling, fault-tolerant.
* **Use Case**: Enterprise-grade apps needing speed and reliability.

**🔹 3. Amazon DynamoDB**

* **Type**: NoSQL (Key-Value and Document)
* **Features**: Serverless, auto-scaling, millisecond latency.
* **Use Case**: Real-time apps, gaming, IoT, mobile apps.

**🔹 4. Amazon ElastiCache**

* **Type**: In-memory caching
* **Engines**: Redis, Memcached
* **Use Case**: Speed up database queries, session storage, caching layers.

**🔹 5. Amazon Redshift**

* **Type**: Data warehouse (columnar storage)
* **Features**: Fast querying for analytics, BI tools integration.
* **Use Case**: Big data analytics, reporting, dashboards.

**🔹 6. Amazon DocumentDB**

* **Type**: Document database (MongoDB-compatible)
* **Use Case**: JSON-based applications, content management systems.

**🔹 7. Amazon Neptune**

* **Type**: Graph database
* **Use Case**: Social networks, recommendation engines, fraud detection.

**🔹 8. Amazon Timestream**

* **Type**: Time-series database
* **Use Case**: IoT, telemetry, sensor data, time-based analytics.

**🔹 9. Amazon QLDB (Quantum Ledger DB)**

* **Type**: Ledger database
* **Use Case**: Immutable, cryptographically verifiable records (e.g., financial systems).

6 Pillars of AWS:

* Operational Excellence
* Security
* Reliability (ability to recover from failure and meet customer demands without interruption)
* Performance efficiency
* Cost Optimization
* Sustainability

**🗄️ Microsoft Azure**

**1. IaaS (Infrastructure as a Service)**

**What it provides:**  
Virtualized computing resources over the internet — like servers, storage, and networking.

**You manage:**  
Operating systems, applications, runtime, and data.

**Cloud provider manages:**  
Hardware, virtualization, storage, and networking.

**Examples:**

* Microsoft Azure Virtual Machines
* Amazon EC2
* Google Compute Engine

**Use case:**  
You want full control over your environment but don’t want to manage physical hardware.

**2. PaaS (Platform as a Service)**

**What it provides:**  
A platform allowing you to develop, run, and manage applications without dealing with infrastructure.

**You manage:**  
Applications and data.

**Cloud provider manages:**  
Runtime, middleware, OS, servers, storage, and networking.

**Examples:**

* Azure App Service
* Google App Engine
* Heroku

**Use case:**  
You want to focus on coding and deploying apps without worrying about infrastructure or OS updates.

**3. SaaS (Software as a Service)**

**What it provides:**  
Fully functional software delivered over the internet.

**You manage:**  
Just the usage — everything else is handled by the provider.

**Cloud provider manages:**  
Everything — infrastructure, platform, and software.

**Examples:**

* Microsoft 365
* Google Workspace
* Salesforce

**Use case:**  
You need ready-to-use software without any setup or maintenance.

Integration within the Microsoft ecosystem is easier for better collaboration.

Some of the features:

* Attribute Log reader – logs all the data
* Azure Virtual machines – similar to AWS EC2
* Managed disks are similar to EB’s
* rage are similar to S3
* Azure DB – overview of analytics and big services
* Similar to AWS budget Azure has cost Management
* Azure can be integrated with IoT, AI and Kubernetes

**🗄️ Google Cloud Platform**

Google Cloud Platform (GCP) is a suite of cloud computing services offered by **Google**, designed to help businesses and developers build, deploy, and scale applications, websites, and services on the same infrastructure that Google uses internally.

#### **1. Compute Services**

* **Compute Engine**: Virtual machines (VMs) for running workloads.
* **App Engine**: Platform-as-a-Service (PaaS) for deploying applications without managing infrastructure.
* **Cloud Functions**: Serverless functions triggered by events.
* **Kubernetes Engine (GKE)**: Managed Kubernetes for container orchestration.

#### **2. Storage & Databases**

* **Cloud Storage**: Object storage for unstructured data.
* **Persistent Disks**: Block storage for VMs.
* **Cloud SQL**: Managed relational databases (MySQL, PostgreSQL, SQL Server).
* **Firestore & Datastore**: NoSQL document databases.
* **Bigtable**: Scalable NoSQL database for large analytical workloads.
* **Gcp’s deep integration of big data and ML helps a lot and better than any other platform for analysing the data**

#### **3.Big Data & Machine Learning**

* **BigQuery**: Serverless data warehouse for analytics.
* **Dataflow**: Stream and batch data processing.
* **Dataproc**: Managed Spark and Hadoop.
* **Vertex AI**: End-to-end platform for building and deploying ML models.

| **Category** | **Description** | **AWS** | **Azure** | **GCP** |
| --- | --- | --- | --- | --- |
| **Compute** | Virtual Machines | EC2 (Elastic Compute Cloud) | Virtual Machines (VMs) | Compute Engine |
|  | Serverless Functions | Lambda | Functions | Cloud Functions |
|  | Container Orchestration | EKS (Elastic Kubernetes Service) | AKS (Azure Kubernetes Service) | GKE (Google Kubernetes Engine) |
|  | Platform-as-a-Service (PaaS) | Elastic Beanstalk | App Service | App Engine |
|  | Serverless Containers | Fargate | Container Instances | Cloud Run |
| **Storage** | Object Storage | S3 (Simple Storage Service) | Blob Storage | Cloud Storage |
|  | Block Storage | EBS (Elastic Block Store) | Managed Disks | Persistent Disk |
|  | File Storage | EFS (Elastic File System) | Azure Files | Filestore |
|  | Archive Storage | S3 Glacier | Blob Storage Archive Tier | Cloud Storage Archive |
| **Databases** | Relational (SQL) | RDS (MySQL, PostgreSQL, etc.) | SQL Database | Cloud SQL |
|  | NoSQL (Document) | DynamoDB | Cosmos DB (Document API) | Firestore |
|  | NoSQL (Key-Value) | DynamoDB | Cosmos DB (Table API) | Firestore / Bigtable |
|  | Data Warehouse | Redshift | Synapse Analytics | BigQuery |
|  | In-Memory Cache | ElastiCache | Cache for Redis | Memorystore |
| **Networking** | Virtual Network | VPC (Virtual Private Cloud) | VNet | VPC |
|  | Load Balancer | ELB / ALB / NLB | Load Balancer | Cloud Load Balancing |
|  | DNS Management | Route 53 | Azure DNS | Cloud DNS |
|  | CDN | CloudFront | Azure Front Door / CDN | Cloud CDN |
|  | Direct Connect | Direct Connect | ExpressRoute | Cloud Interconnect |
| **Identity & Security** | Identity & Access Mgmt | IAM | Azure AD + IAM | Cloud IAM |
|  | Secrets Management | Secrets Manager | Key Vault | Secret Manager |
|  | DDoS Protection | AWS Shield | Azure DDoS Protection | Cloud Armor |
|  | Web Application Firewall | AWS WAF | Application Gateway WAF | Cloud Armor |
| **Management & Monitoring** | Monitoring & Logging | CloudWatch | Azure Monitor | Cloud Operations (Monitoring & Logging) |
|  | Infrastructure as Code | CloudFormation | ARM Templates | Deployment Manager |
|  | Config & Compliance | AWS Config | Azure Policy | Security Command Center |
| **Big Data & Analytics** | Data Pipelines (ETL) | Glue | Data Factory | Dataflow |
|  | Stream Processing | Kinesis | Stream Analytics | Dataflow / Pub/Sub |
|  | Real-Time Messaging | SQS / SNS | Service Bus / Event Grid | Pub/Sub |
| **AI/ML** | Machine Learning Platform | SageMaker | Azure ML | Vertex AI |
|  | Vision AI | Rekognition | Computer Vision | Vision AI |
|  | Speech AI | Transcribe | Speech Services | Speech-to-Text / Text-to-Speech |
|  | Language AI | Comprehend | Text Analytics | Natural Language AI |
| **DevOps** | Container Registry | ECR (Elastic Container Registry) | Azure Container Registry | Artifact Registry |
|  | CI/CD Pipelines | CodePipeline | Azure DevOps Pipelines | Cloud Build |
|  | Source Repositories | CodeCommit | Azure Repos | Cloud Source Repositories |

**What is an LLM?**

**LLM** stands for **Large Language Model**.

**In simple terms:**

* It’s a type of artificial intelligence (AI) model designed to understand and generate human language.
* LLMs are trained on massive amounts of text data (like books, articles, websites) to learn patterns, grammar, facts, and even reasoning.

**Key features:**

* **Scale:** “Large” refers to the huge number of parameters (think: settings or “neurons”)—often billions or even trillions.
* **Capabilities:** LLMs can answer questions, write essays, summarize text, translate languages, generate code, and much more.
* **Examples:** OpenAI’s GPT-4, Google’s Gemini, Meta’s Llama, and others.

**How do they work?**

* They use deep learning (specifically, neural networks called transformers).
* When you give them a prompt, they predict the most likely next words based on their training.

**1. ChatGPT (by OpenAI)**

* Based on the GPT (Generative Pre-trained Transformer) architecture.
* Versions: GPT-3, GPT-3.5, GPT-4, etc.
* Used for chatbots, writing assistants, coding help, and more.

**2. Gemini (by Google)**

* Previously known as Bard.
* Built on Google’s advanced AI models.
* Integrated with Google Search and other Google services.

**3. Llama (by Meta/Facebook)**

* Open-source LLM series: Llama, Llama 2, Llama 3.
* Used for research, chatbots, and enterprise solutions.

**4. Claude (by Anthropic)**

* Focuses on safety and helpfulness.
* Used in business, customer support, and productivity tools.

**5. Mistral (by Mistral AI)**

* European LLM, open-source and efficient.
* Known for its speed and performance.

**What is an AI Agent?**

An **AI agent** is a software program or system that can:

* **Perceive** its environment (through data, sensors, or user input)
* **Decide** what actions to take (using rules, logic, or machine learning)
* **Act** to achieve specific goals or tasks

**Key Points:**

* **Autonomous:** AI agents can operate on their own, making decisions without constant human guidance.
* **Goal-oriented:** They are designed to accomplish specific objectives (like booking a meeting, answering questions, or controlling a robot).
* **Interactive:** Many AI agents can communicate with users or other systems.

**Examples:**

* **Chatbots:** Like ChatGPT, which can answer questions and have conversations.
* **Virtual Assistants:** Such as Siri, Alexa, or Google Assistant.
* **Robotic Agents:** Robots that navigate spaces, pick up objects, or perform tasks.
* **Software Agents:** Programs that monitor emails, automate workflows, or trade stocks.

**Cloud computing**

Remote service which we can pay-as-we-use.

All the cloud companies have their own data centers

**🔍 What Does a Data Center Do?**

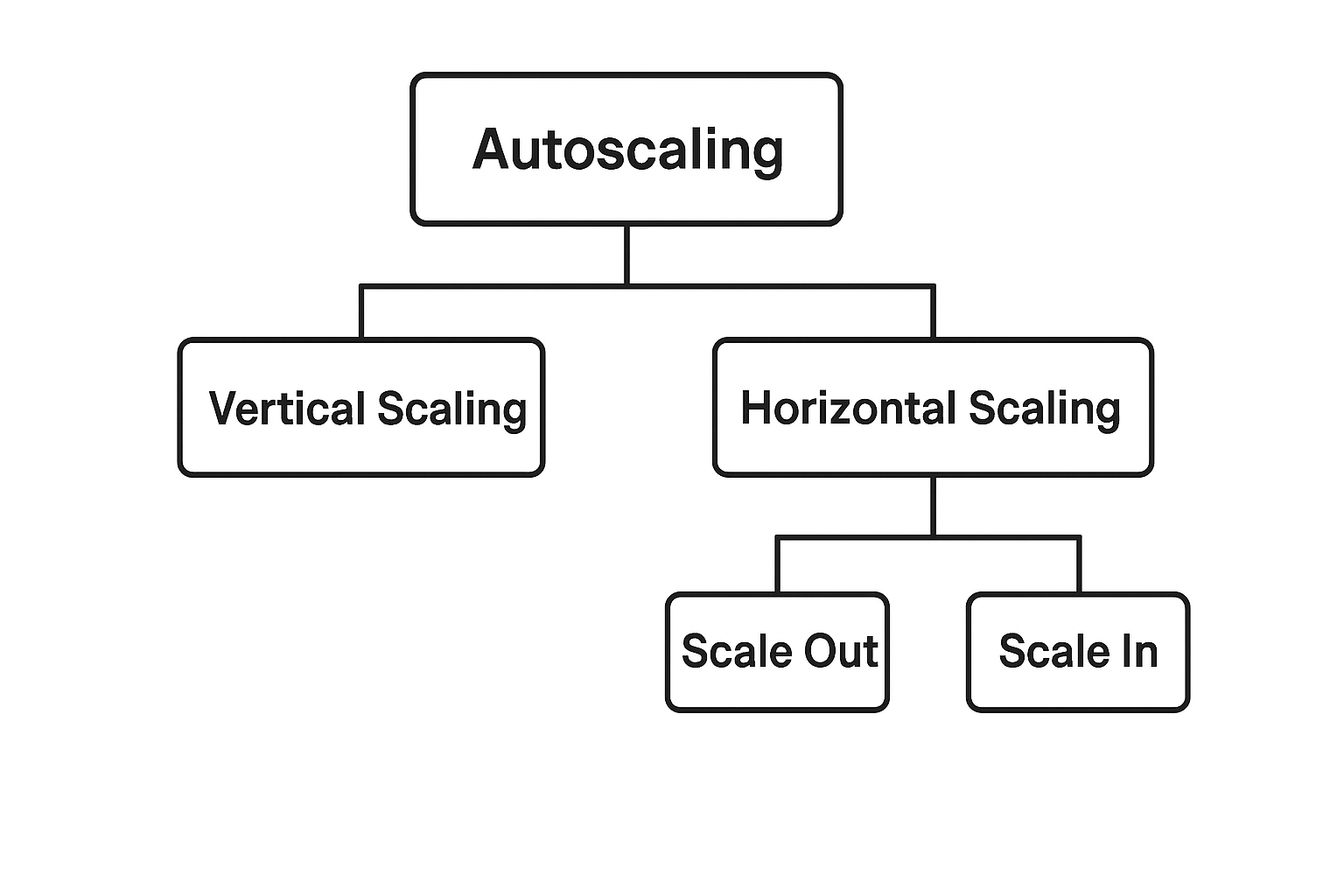
It provides the infrastructure needed to:

* Store and process large amounts of data
* Run applications and services (like websites, cloud platforms, AI models)
* Support enterprise IT operations

**🏢 Types of Data Centers**

1. **Enterprise Data Centers** – Owned and operated by companies for internal use.
2. **Colocation Data Centers** – Businesses rent space and resources.
3. **Cloud Data Centers** – Operated by cloud providers like AWS, Microsoft Azure, Google Cloud.
4. **Edge Data Centers** – Smaller centers located closer to users for faster response times.

On demand service :- Autoscaling there are 2 types vertical scaling and horizontal scaling and further horizontal scaling is classified into scale up and scale down



Whenever the traffic increases the we increase the number of servers – this called scale up

Whenever the traffic decreases the additional servers are removed – this is called scale down

A **load balancer** is a system or device that distributes incoming network traffic across multiple servers. Its main goal is to ensure:

* **High availability**
* **Reliability**
* **Efficient resource utilization**
* **Scalability**

**🔧 How Load Balancers Work**

Imagine you have a website with millions of users. Instead of sending all requests to one server (which could crash), a load balancer spreads the traffic across several servers.

**📊 Types of Load Balancing**

1. **Round Robin** – Requests are distributed evenly in order.
2. **Least Connections** – Sends traffic to the server with the fewest active connections.
3. **IP Hash** – Uses the client’s IP address to determine which server to route to.
4. **Weighted Distribution** – Servers with more capacity get more traffic.

**🔼 Vertical Scaling (Scale Up)**

Also known as **"scaling up"**, this means increasing the capacity of a single server or resource.

**✅ How it works:**

* Add more **CPU**, **RAM**, **storage**, or **power** to an existing machine.
* You’re making one server stronger.

**📌 Pros:**

* Simple to implement.
* No need to change application architecture.

**❌ Cons:**

* There's a **limit** to how much you can upgrade one machine.
* Can be **expensive**.
* If the server fails, everything goes down.

**🔁 Horizontal Scaling (Scale Out / Scale In)**

Also known as **"scaling out/in"**, this means adding or removing multiple servers to handle load.

**✅ How it works:**

* Add more machines (scale out) to distribute the load.
* Remove machines (scale in) when demand drops.

**📌 Pros:**

* Highly **scalable** and **resilient**.
* Supports **load balancing** and **fault tolerance**.
* Ideal for cloud-native applications.

**❌ Cons:**

* More complex to manage.
* Requires distributed architecture and often stateless design.

### 🧠 Summary Table

| **Feature** | **Vertical Scaling** | **Horizontal Scaling** |
| --- | --- | --- |
| Method | Upgrade existing server | Add/remove servers |
| Complexity | Low | High |
| Scalability | Limited | Virtually unlimited |
| Cost | Can be expensive | Cost-effective at scale |
| Fault Tolerance | Low | High |

**🔢 What Are Port Numbers?**

A **port number** is a numerical identifier used in networking to specify a particular process or service on a device.

* Think of an IP address as a **building**, and port numbers as **doors** to specific rooms (services) inside.
* Port numbers range from **0 to 65535**.

**📂 Port Number Categories:**

| **Range** | **Type** | **Description** |
| --- | --- | --- |
| 0–1023 | **Well-known ports** | Reserved for common protocols (e.g., HTTP, FTP) |
| 1024–49151 | **Registered ports** | Used by software applications |
| 49152–65535 | **Dynamic/private ports** | Temporary ports for client-side communication |

**🌐 What Are Protocols?**

A **protocol** is a set of rules that define how data is transmitted over a network.

**Common Protocols:**

| **Protocol** | **Port** | **Description** |
| --- | --- | --- |
| **HTTP** | 80 | Web traffic (unsecured) |
| **HTTPS** | 443 | Secure web traffic |
| **FTP** | 21 | File transfer |
| **SSH** | 22 | Secure remote login |
| **SMTP** | 25 | Sending emails |
| **DNS** | 53 | Domain name resolution |
| **Telnet** | 23 | Remote login (not secure) |
| **POP3** | 110 | Receiving emails |
| **IMAP** | 143 | Email retrieval and storage |

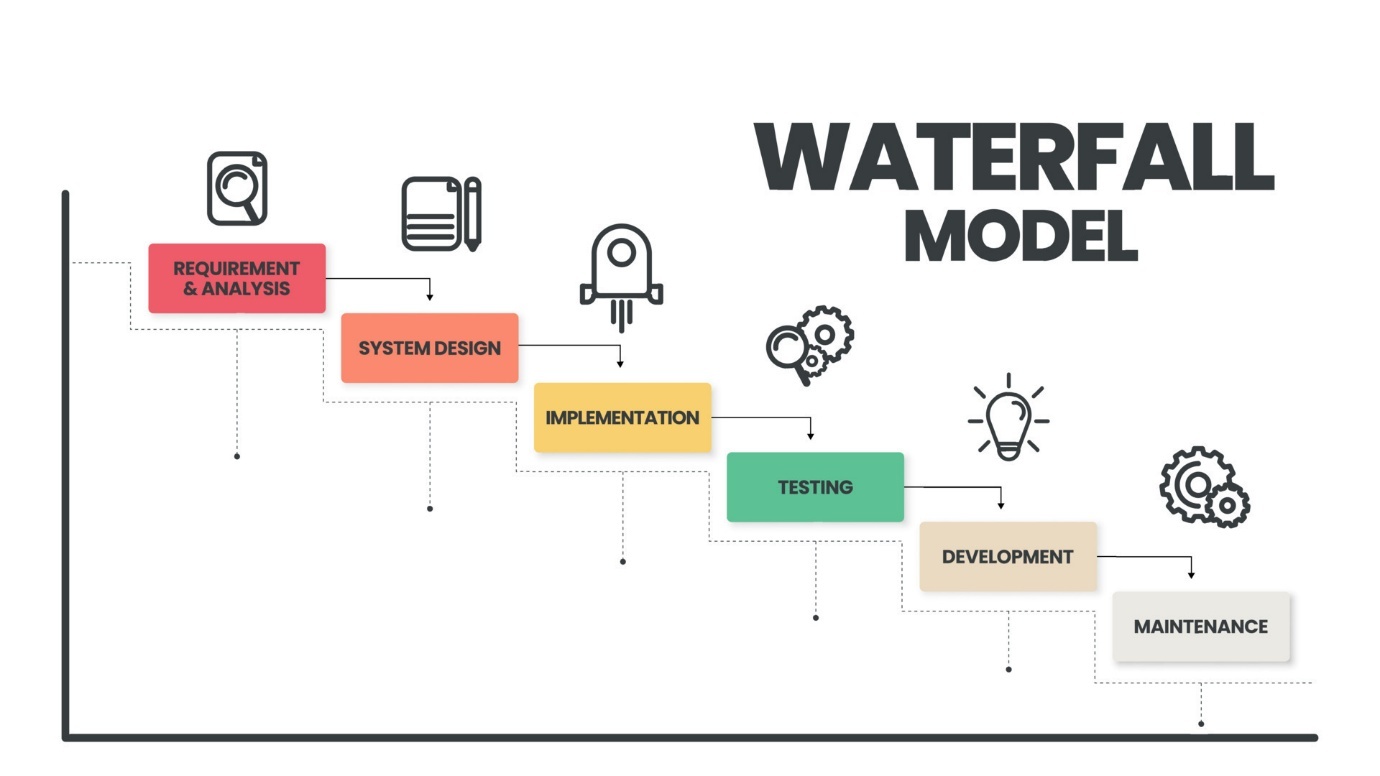
In SDLC there are 6 process:

* Requirement gathering
* Designing
* Implementation
* Testing
* Deployment
* Maintenance

**Waterfall Model**

* **Definition:**  
  The Waterfall model is a **linear and sequential** approach to software development. Each phase must be completed before the next begins, and there’s little to no overlap between phases.
* **Phases:**
  1. Requirement Gathering
  2. System Design
  3. Implementation (Coding)
  4. Testing
  5. Deployment
  6. Maintenance
* **Key Characteristics:**
  1. **Rigid structure:** Progress flows in one direction (“downward” like a waterfall).
  2. **Documentation-heavy:** Each phase produces detailed documents.
  3. **Best for:** Projects with well-understood, unchanging requirements.
* **Pros:**
  1. Simple to understand and manage.
  2. Well-suited for smaller projects with clear requirements.
* **Cons:**
  1. Difficult to accommodate changes once a phase is completed.
  2. Late discovery of issues (testing happens after coding is complete).
  3. Dependency – each team is depended on each other. Unless a previous team is completed the next step can not move forward
  4. Additional changes can not done
  5. Customer don’t know what we are implementing
  6. Customer don’t have any idea what we are developing until the end
  7. It doesn’t fit for larger applications

To overcome all these disadvantages we are moving forward with the agile methodology.

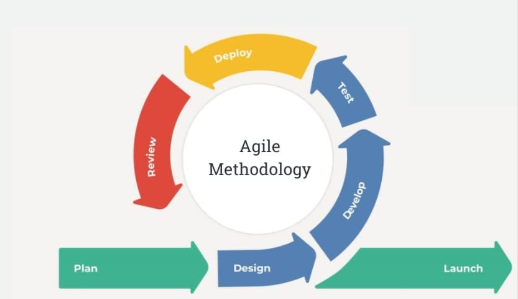


**Agile Methodology**

* **Definition:**  
  Agile is an **iterative and incremental** approach to software development. Work is divided into small cycles called **sprints** (usually 1–4 weeks), and teams continuously collaborate and adapt to changes.
* Agile is an iterative approach or incremental approach. In agile there are two important frameworks 1. Sprint 2. Scrum
* Scrum is an Agile framework for managing and completing complex projects. It divides work into fixed-length iterations called **sprints** (usually 2–4 weeks).
  + Scrum is a framework where we will understand the status of the tasks.
* A **sprint** is a time-boxed period (commonly 2 weeks) in which a specific set of work must be completed and made ready for review.
  + Every application is divided into the modules. Every module (feature of the application) is going to be implemented with sprint (a time period)
  + The maximum sprint duration is 1-4 weeks
* **Key Characteristics:**
  + **Flexible and adaptive:** Requirements and solutions evolve through collaboration.
  + **Continuous feedback:** Regular reviews and adjustments.
  + **Best for:** Projects where requirements may change or are not fully known at the start.
* **Pros:**
  + Responds quickly to changes.
  + Frequent delivery of working software.
  + High customer involvement and satisfaction.
* **Cons:**
  + Requires close collaboration and communication.
  + Less predictability in scope, cost, and timeline.

Every application is divided into the modules. Every module (feature of the application) is going to be implemented with sprint (a time period)

The maximum sprint duration is 1-4 weeks



What is an artifact ?

It is an executable file where we deploy the server

Devops is the combination of cultural philosophies, practices and tools.

DevOps is divided into 2 parts Dev team and operation

To fill the gap between the Dev team and Ops team the DevOps teams comes.

**Dev Team (Development)**

* **Responsibilities:**
  + **Planning:** Understanding requirements, creating user stories, and defining features.
  + **Designing:** Architecting the solution, designing UI/UX, and technical design.
  + **Development:** Writing code, building features, and integrating components.
  + **Testing:** Unit testing, integration testing, and sometimes initial QA.

**Ops Team (Operations)**

* **Responsibilities:**
  + **Infrastructure Setup:** Provisioning servers, networks, and storage.
  + **Deployment:** Releasing applications to production or staging environments.
  + **Backup:** Ensuring data and system backups are in place.
  + **Monitoring:** Keeping track of application health, performance, and uptime.
  + **Operational Support:** Handling incidents, troubleshooting, and maintenance.

**DevOps: Bridging the Gap**

* **Why DevOps?**
  + Traditionally, Dev and Ops worked in silos, leading to slow releases, miscommunication, and “it works on my machine” issues.
  + **DevOps** is a culture and set of practices that **integrates development and operations** to enable:
    - Faster, more reliable releases
    - Better collaboration and communication
    - Automation of build, test, and deployment processes
* **DevOps Team Responsibilities:**
  + Automating CI/CD pipelines
  + Infrastructure as Code (IaC)
  + Continuous monitoring and feedback
  + Security integration (DevSecOps)
  + Collaboration across the software lifecycle

the **four main environments** in the software development life cycle, along with a bit of extra detail for clarity:

**1. Development Environment (Dev)**

* **Purpose:**  
  Where developers write code and test basic functionality.
* **Users:**  
  Developers only.
* **Key Points:**
  + Frequent code changes.
  + Not stable; used for initial testing and debugging.

is a server where primarily the developer test the minimum functionality of the application

**2. Test Environment**

* **Purpose:**  
  Where the testing team validates the application.
* **Users:**  
  QA/Testers.
* **Key Points:**
  + Used for manual and automated testing.
  + Tests include functional, non-functional, stress, load, and cross-platform testing.
  + More stable than Dev, but not customer-facing.
* once we deploy the environment in the testing environment, there is a team called testing team where they are going to test application using manual testing, automation testing to understand how the application nbehaving along with that they are going to check the functional test cases and non functional test cases, stress testing load testing cross platform testing.

**3. UAT Environment (User Acceptance Testing)**

* **Purpose:**  
  Also called **pre-production**; where clients or end-users validate the application before release.
* **Users:**  
  Business users, clients, or stakeholders.
* **Key Points:**
  + Mimics production as closely as possible.
  + Used to ensure the application meets business requirements.
  + Final approval before going live.

(user acceptance test) UAT also called as a pre production environment. It is a customer copy of the application. We are going to showcase to the client before we deploy the application in the main environment

**4. Production Environment**

* **Purpose:**  
  The live environment where the application is accessible to end-users/customers.
* **Users:**  
  Actual customers or users.
* **Key Points:**
  + Highest stability and security.
  + Real data and real usage.
  + Monitored and maintained by operations/support teams.

where we deploy the application and we provide the access to the customer.

To **run an application**, there are typically **four primary requirements** you need to consider:

**1. Platform Dependency**

* **What it means:**  
  The underlying operating system and hardware (e.g., Windows, Linux, macOS, cloud environment) required to run your application.
* **Examples:**
  + Windows Server 2019
  + Ubuntu 22.04
  + ARM vs. x86 architecture

**2. Application Dependency**

* **What it means:**  
  Other software or services your application relies on to function.
* **Examples:**
  + Web servers (e.g., Apache, Nginx, IIS)
  + Application servers (e.g., Tomcat, JBoss)
  + Middleware (e.g., message brokers, caching servers)

**3. Library Files**

* **What it means:**  
  External libraries, frameworks, or packages your application needs.
* **Examples:**
  + Java JAR files
  + Python packages (from requirements.txt)
  + Node.js modules (node\_modules)
  + .NET DLLs

**4. Databases**

* **What it means:**  
  The data storage system your application uses to store and retrieve information.
* **Examples:**
  + MySQL, PostgreSQL, Oracle, SQL Server
  + MongoDB, Redis, Cassandra (NoSQL databases)

**🛠️ Build Tools**

Build tools automate the process of compiling source code, running tests, and packaging applications (compile the source code into a single package) .

| **Language** | **Build Tool** | **Output Format** |
| --- | --- | --- |
| **Java** | Maven | .jar / .war |
| **Python** | PyPI (via setuptools, wheel) | .whl / .tar.gz |
| **Node.js** | NPM | Bundled JS files |

* **Maven**: A powerful Java build tool that handles dependencies, compilation, testing, and packaging.

For Every programming language we use a different build tools

* for java to build a .jar file or .war file we are using maven.
* For building python applications we are using pypi
* For building Node js applications we use NPM

**🐳 Docker**

* **Purpose**: Create **portable**, **platform-independent** applications by packaging code with its dependencies into containers.
* Docker is a tool we are using to create a portable applications ( the code will work anywhere), platform independent applications.
* **Benefits**:
  + Consistent environments across development, testing, and production.
  + Easy to scale and deploy.
* **Limitations**:
  + No built-in **load balancing**, **self-healing**, or **auto-scaling**.

**📦 Docker Hub**

* A **cloud-based registry** where Docker images are stored and versioned.
* Developers can **push** and **pull** images for deployment.
* Hop.docker.com -> it is a repository where we can store different versions of the applications.

**☸️ Kubernetes**

* A **container orchestration tool** that manages Docker containers at scale.
* To overcome all this disadvantages of the docker we are moving towards Kubernetes. It is an orchestration tool which is going to manage all the containers. If any container goes down, it will re-create the same container
* **Features**:
  + **Self-healing**: Automatically restarts failed containers.
  + **Load balancing**
  + **Auto-scaling**
  + **Rolling updates**

**🔁 CI/CD (Continuous Integration / Continuous Delivery)**

* **CI**: Automatically integrates code changes from multiple developers into a shared repository.
* **CD**: Automatically delivers and deploys code to production or staging environments.
* CI/CD: continuous integration/ continuous delivery : CI/CD tools are part of the devops. When ever the software development team deploys the code it will pull the code
* **Tool Example**: **Jenkins**
  + Automates the entire pipeline: code pull → build → test → package → deploy.

**🔄 DevOps Workflow Summary**

1. **Dev team** pushes code to **GitHub**
2. **SonarQube** checks code quality
3. **Maven** builds the application
4. **Docker** containerizes the app
5. Image is pushed to **Docker Hub**
6. **Kubernetes** orchestrates deployment
7. **Jenkins** automates the CI/CD pipeline

**✅ 7 C’s of Jenkins in DevOps**

Jenkins helps automate the entire software delivery process. The **7 C’s** represent the stages Jenkins can manage:

1. **Continuous Integration**
   * Automatically merge code changes into a shared repository.
   * Detect integration issues early.
2. **Continuous Building**
   * Compile the code and generate build artifacts (e.g., .jar, .war, .exe).
3. **Continuous Testing**
   * Run automated tests (unit, integration, regression) to ensure code quality.
4. **Continuous Delivery**
   * Prepare code for release to production at any time.
   * Manual approval may be required before deployment.
5. **Continuous Deployment**
   * Automatically deploy code to production without manual intervention.
6. **Continuous Monitoring**
   * Track application performance, resource usage, and errors post-deployment.
7. **Continuous Improvement**
   * Use feedback from monitoring and testing to improve code, processes, and performance.

**📊 What is Monitoring?**

**Monitoring** is the process of observing and analyzing how an application behaves in real-time.

**Key Metrics Monitored:**

* **CPU usage**
* **Memory consumption**
* **Disk I/O**
* **Network traffic**
* **Application errors**
* **User activity**

**Tools Used:**

* **Prometheus**: Collects and stores metrics.
* **Grafana**: Visualizes metrics through dashboards.
* **Alerting**: Set thresholds to trigger alerts (e.g., high CPU usage).

Monitoring is nothing but to understand how the application is behaving. How much resources are consuming like CPU utilization memory utilization how much traffic is coming and setting alerts like using gphana and promotheus

**What is Jira?**

**Jira** is a popular **project management and issue tracking tool** developed by Atlassian. It’s widely used in **Agile software development** to help teams plan, track, and manage work efficiently.

**Key Features of Jira:**

1. **Issue Tracking**
   * Track bugs, tasks, stories, and epics.
   * Assign issues to team members and monitor progress.
2. **Agile Boards**
   * Supports **Scrum** and **Kanban** boards.
   * Visualize work in progress and manage sprints.
3. **Project Management**
   * Create roadmaps, backlogs, and sprint plans.
   * Set priorities and deadlines.
4. **Custom Workflows**
   * Define how tasks move through different stages (e.g., To Do → In Progress → Done).
5. **Integration**
   * Works with tools like **Confluence**, **Bitbucket**, **GitHub**, **Slack**, **Jenkins**, and more.
6. **Reporting**
   * Generate burndown charts, velocity reports, and sprint summaries.

**Common Use Cases:**

* **Software Development Teams:**  
  Plan sprints, track bugs, and manage releases.
* **IT Support Teams:**  
  Handle service requests and incidents.
* **Business Teams:**  
  Manage tasks, approvals, and workflows.

**Jira in DevOps:**

In a DevOps pipeline, Jira is often used to:

* Link code commits to user stories or bugs.
* Trigger CI/CD pipelines based on issue status.
* Track deployment progress and incidents.

### 🧾 What is pom.xml?

pom.xml stands for **Project Object Model** XML. It’s a configuration file used by **Apache Maven**, which is a **build automation tool** primarily for **Java projects**.

It defines what kind of artifacts we have to use and it is going tot define the dependencies and plugs of the application.

**🔧 In Simple Terms:**

Imagine you're building a Java project. You need:

* Libraries (like Spring, Hibernate),
* Build instructions (how to compile, test, package),
* Version control (which version of Java or dependencies to use).

Instead of doing all this manually, you use **Maven**, and Maven reads the pom.xml file to know:

* What your project is,
* What dependencies it needs,
* How to build and deploy it.

**📦 What’s Inside pom.xml?**

Here’s what you typically define in it:

* **Project info**: name, version, description.
* **Dependencies**: external libraries your project needs.
* **Plugins**: tools to compile, test, package, etc.
* **Build settings**: how to build the project.
* **Repositories**: where to download dependencies from.

**🧠 Why Is It Important in DevOps?**

In a DevOps pipeline:

* Jenkins can use Maven to build Java projects.
* Jenkins reads pom.xml to know how to compile, test, and package the code.
* It ensures **consistent builds** across environments.

YAML (short for **YAML Ain’t Markup Language**) is a human-readable data serialization format that's commonly used for configuration files and data exchange between languages with different data structures. It’s designed to be easy to read and write, making it a popular choice in DevOps, cloud deployments, and software development.

**🔧 Key Features of YAML:**

* **Readable syntax**: Uses indentation (like Python) instead of brackets or braces.
* **Supports complex data structures**: Lists, dictionaries (maps), nested structures.
* **Language-agnostic**: Can be used with many programming languages (Python, JavaScript, Go, etc.).
* **Widely used in tools**: Kubernetes (.yaml manifests), GitHub Actions (.yml workflows), Ansible, Docker Compose, etc.

**🧩 YAML Data Types**

* **Scalars**: Strings, numbers, booleans
* **Sequences**: Lists (arrays)
* **Mappings**: Key-value pairs (dictionaries)

**✅ Rules to Remember**

* Indentation is **critical** (use spaces, not tabs)
* No quotation marks needed unless the string has special characters
* Comments start with #

**🛠️ Common Use Cases**

* **Configuration files**: e.g., .travis.yml, docker-compose.yml
* **Infrastructure as Code**: e.g., Kubernetes manifests
* **CI/CD pipelines**: e.g., GitHub Actions workflows
* **Data exchange**: Between services or APIs

Ansible is a powerful **open-source automation tool** used extensively in **DevOps** for tasks like configuration management, application deployment, and orchestration. It helps teams automate repetitive tasks, manage infrastructure efficiently, and ensure consistency across environments.

**WHY Ansible:**

* Complete automation
* Increase uptime
* Improve performance
* Ensure compliance
* Prevent errors
* Reduce cost
* Uses python
* Push mechanism
* Relies on ssh

Before ansibles we have many tools chef puppet and salt. These tools are not agentless, complicated setup pull mechanism, lot of learning required and huge overhead of the infrastructure setup.

**🔧 1. Configuration Management**

**Definition**:  
Configuration management is the process of **maintaining systems in a desired state**. This includes installing software, setting up services, managing files, users, permissions, and ensuring consistency across environments.

* Method through which we can automate admin tasks
* Turns your code into infrastructure
* Testable repeatable and version able

**In Ansible**:

* You write **playbooks** (in YAML) that describe how a system should be configured.
* Ansible applies these configurations to one or many servers.
* If a server already matches the desired state, Ansible **does nothing** (this is called *idempotency*).

**Pain Points:**

* Managing users and group accounts
* Dealing with packages
* Taking backups
* Deploying all kinds of applications
* Configure services

**🏗️ 2. Provisioning**

**Definition**:  
Provisioning is the process of **setting up infrastructure** — like creating virtual machines, containers, cloud resources (AWS EC2, Azure VMs), networks, etc.

**In Ansible**:

* You use **cloud modules** (e.g., amazon.aws.ec2, azure.azcollection.vm) to create and configure infrastructure.
* It can automate the entire setup from scratch — including OS installation, networking, and software.

**Example Tasks**:

* Launch EC2 instances on AWS
* Create a Kubernetes cluster
* Set up Docker containers
* Configure cloud storage or databases

**✅ Ansible in a DevOps CI/CD Pipeline**

Ansible plays a **key role in automation**, especially when integrated with tools like **Docker** and **Kubernetes**, and it uses **YAML** as its configuration language.

**🔄 How It All Fits Together**

**🧱 Ansible + Docker**

* Automates the creation and management of Docker containers.
* Can build Docker images, run containers, and configure them.
* Example: Deploy a microservice inside a Docker container using an Ansible playbook.

**☸️ Ansible + Kubernetes**

* Manages Kubernetes resources (pods, deployments, services) using Ansible modules.
* Can automate cluster setup, app deployment, and scaling.
* Example: Deploy an app to a Kubernetes cluster using Ansible.

**🔁 CI/CD Automation**

* **Jenkins**, **GitLab CI**, or **GitHub Actions** can trigger Ansible playbooks as part of the pipeline.
* Ansible handles:
  + Provisioning infrastructure (cloud VMs, containers)
  + Deploying applications
  + Configuring environments
  + Rolling back on failure

**🧩 Traditional vs Modern Deployment**

**🚫 Traditional Deployment (e.g., Tomcat)**

* You deploy Java apps (WAR files) to a **Tomcat server**.
* Requires manual setup or scripts to configure the server, deploy the app, and manage updates.
* Often runs on a VM or physical server.

**✅ Modern Deployment with Ansible + Docker**

* You **containerize** the app using **Docker** (e.g., build a Docker image with Tomcat + your app).
* Use **Ansible** to automate:
  + Building the Docker image
  + Running the container
  + Configuring the environment
* This is faster, more portable, and easier to scale.

**🏗️ Infrastructure Setup**

**1. Create 3 EC2 Instances**

* **Purpose**: To simulate a real-world setup where one machine (Ansible control node) manages multiple target machines (host nodes).
* **Naming**:
  + ansible-server: Control node where Ansible is installed.
  + host-server-1 and host-server-2: Target nodes to be managed.
* **Amazon Linux Kernel 6.12**: Chosen for compatibility and stability.
* **Instance Type t3.small**: Provides enough memory (2 GB) for lightweight automation tasks.

**🔐 Key Pair (.ppk)**

* **Purpose**: To securely SSH into EC2 instances.
* **.ppk Format**: Used with PuTTY on Windows systems for authentication.

**⚙️ Installing Ansible on Control Node**

**2. sudo yum update -y and sudo dnf update -y**

* **Why**: Updates all system packages to the latest versions to ensure compatibility and security.
* **yum vs dnf**: dnf is the newer package manager; running both ensures coverage across Amazon Linux versions.

**3. sudo dnf install -y ansible**

* **Why**: Installs Ansible on the control node so it can manage other servers.

**4. sudo curl -o /etc/ansible/ansible.cfg ...**

* **Why**: Downloads a sample configuration file to customize Ansible's behavior.

**5. ansible --version**

* **Why**: Verifies that Ansible is installed correctly and shows its version.

**6. cd /etc/ansible**

* **Why**: Navigates to Ansible’s default configuration directory.

**📝 Configuration Files**

**7. Edit ansible.cfg**

[defaults]

inventory = /etc/ansible/hosts

host\_key\_checking = False

retry\_files\_enabled = False

sudo\_user = root

* **Why**:
  + inventory: Tells Ansible where to find the list of target machines.
  + host\_key\_checking: Disables SSH key verification to avoid manual prompts.
  + retry\_files\_enabled: Prevents clutter from retry files.
  + sudo\_user: Specifies the user to run commands as (root).

**8. Create hosts File**

[webservers]

<private-ip-of-host-server-1>

<private-ip-of-host-server-2>

* **Why**: Defines the target machines Ansible will manage, grouped under webservers.

**👤 User Setup**

**9. Create ansible User**

useradd ansible

* **Why**: Creates a dedicated user for running Ansible tasks securely.

**10. su - ansible**

* **Why**: Switches to the ansible user to test permissions and environment.

**11. Grant Sudo Privileges**

ansible ALL=(ALL) NOPASSWD: ALL

* **Why**: Allows the ansible user to run commands as root without needing a password.

**🔁 Repeat on Host Servers**

* **Why**: Ensures the same user and permissions exist on target machines for seamless SSH and command execution.

**🔐 SSH Configuration**

**12. Edit /etc/ssh/sshd\_config**

* Change:

PermitRootLogin yes

* **Why**:
  + Enables root login and password-based SSH authentication.
  + Required for Ansible to connect using the ansible user credentials.

**13. sudo systemctl restart sshd**

* **Why**: Applies the SSH configuration changes.

**🔗 Establish Connection**

**14. su - ansible**

* **Why**: Switches to the Ansible user to initiate connections and run playbooks.

**15. Test Connection**

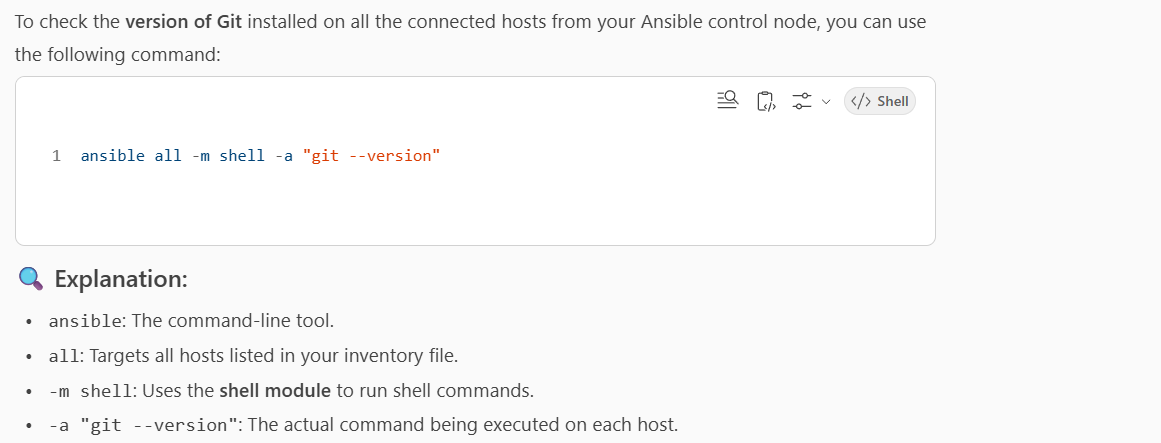
ansible all -m ping

* **Why**: Verifies that Ansible can reach and communicate with the host servers.

**✅ End Goal**

You’ve built a **fully functional Ansible environment** where:

* One control node (ansible-server) can manage multiple target nodes (host-server-1, host-server-2).
* You can now automate tasks like installing software, configuring services, or deploying applications across all servers using simple YAML playbooks.



**⚡️ Ad Hoc Commands in Ansible**

**🔍 What Are They?**

**Ad hoc commands** are **one-time, quick commands** you run directly from the command line to perform tasks on remote hosts — without writing a full playbook.

Ad hoc commands are used to:

* Run **simple tasks** quickly
* Test connectivity
* Perform **administrative actions** like installing packages, restarting services, or checking system info

They are written in **a single line** and executed from the Ansible control node.

A screenshot of a computer

AI-generated content may be incorrect.

**🧠 Why Use Them?**

* For **quick tasks** like checking connectivity, restarting a service, installing a package, or gathering system info.
* Ideal for **testing** or **small-scale operations**.

**🧵 Single-Line Commands**

**🔍 What Are They?**

Single-line commands are **ad hoc commands written in one line** — so technically, **all ad hoc commands are single-line commands**, but the term emphasizes their simplicity and brevity.

**🧩 Linux Families in the Context of Ansible**

Linux distributions are broadly classified into two major families:

**🔴 1. Red Hat Family**

Includes:

* **Red Hat Enterprise Linux (RHEL)**
* **CentOS**
* **Fedora**
* **Amazon Linux**
* **Oracle Linux**
* **SUSE Linux** (though SUSE is sometimes considered separate, it shares similarities)

**🛠️ Package Manager: yum or dnf**

* yum: Older package manager
* dnf: Modern replacement for yum (used in Fedora, Amazon Linux 2+)

**📦 Example Commands:**

Shell

sudo yum update -y

sudo yum install git -y

Show more lines

**📥 Downloading Modules or Files:**

* Use wget or curl to download files from the internet.

Shell

wget https://example.com/file

Show more lines

**🟢 2. Debian Family**

Includes:

* **Debian**
* **Ubuntu**
* **Linux Mint** (based on Ubuntu)

**🛠️ Package Manager: apt or apt-get**

* apt-get: Traditional command-line tool
* apt: More user-friendly wrapper around apt-get

**📦 Example Commands:**

Shell

sudo apt-get update -y

sudo apt-get install git -y

Show more lines

* apt-get update: Refreshes the package list from repositories
* apt-get install: Installs the specified package

**🧰 Why This Matters in Ansible**

When writing **Ansible playbooks or ad hoc commands**, you need to use the **correct module** based on the Linux family:

| **Linux Family** | **Package Manager** | **Ansible Module** |
| --- | --- | --- |
| Red Hat | yum / dnf | yum |
| Debian | apt / apt-get | apt |

**🧾 Example Ansible Ad Hoc Commands:**

**For Red Hat-based systems:**

Shell

ansible all -m yum -a "name=git state=present"

Show more lines

**For Debian-based systems:**

Shell

ansible all -m apt -a "name=git state=present update\_cache=yes"

Show more lines

To install **Apache HTTP Server (httpd)** on **Amazon Linux**, you can use the following command:

A white background with black and white clouds

AI-generated content may be incorrect.**✅ Explanation:**

* sudo: Runs the command with admin privileges.
* yum: Package manager for Amazon Linux (Red Hat-based).
* install httpd: Installs the Apache web server.
* -y: Automatically confirms the installation.

**🔄 After Installation**

To start and enable the service:

A white background with black and white clouds

AI-generated content may be incorrect.

Alternative command: ansible webservers -a "sudo service httpd start"

To check its status:

A white background with black and white clouds

AI-generated content may be incorrect.

To stop the server: ansible webservers -a "sudo service httpd stop"

To restart the server: ansible webservers -a "sudo service httpd restart"

Create a new user account using red hat command Capgemini

Create a file Automate.ssh file

A close-up of a logo

AI-generated content may be incorrect.

**✅ Explanation of the corrected command:**

* -m service: Use the Ansible **service module** to manage services.
* -a "name=capgemini state=started":
  + name=capgemini: The name of the service.
  + state=started: Ensure the service is running.
* --become: Run with elevated privileges (required for managing services).

**🟦 JavaScript / Node.js**

**1. npm**

* **Purpose**: Manages Node.js packages.
* **Usage**: Install, update, and remove packages from the npm registry.
* **Example**: npm install express

**2. yarn**

* **Purpose**: Alternative to npm, faster and more secure.
* **Usage**: Manages Node.js packages with better caching and parallel installs.
* **Example**: yarn add react

**3. bower *(Deprecated)***

* **Purpose**: Used to manage front-end packages (HTML, CSS, JS).
* **Usage**: Was popular before npm/yarn took over.
* **Example**: bower install jquery

**🟥 Python**

**4. pip**

* **Purpose**: Installs Python packages from PyPI.
* **Usage**: Most common Python package manager.
* **Example**: pip install requests

**5. easy\_install *(Deprecated)***

* **Purpose**: Older Python package installer.
* **Usage**: Replaced by pip.
* **Example**: easy\_install flask

**6. pip\_package\_info**

* **Purpose**: Retrieves metadata about installed pip packages.
* **Usage**: Useful for auditing or checking versions.

**🟨 Ruby**

**7. gem**

* **Purpose**: Installs Ruby packages (gems).
* **Usage**: Core tool for Ruby package management.
* **Example**: gem install rails

**8. bundler**

* **Purpose**: Manages Ruby project dependencies via Gemfile.
* **Usage**: Ensures consistent environments across machines.
* **Example**: bundle install

**🟩 PHP**

**9. composer**

* **Purpose**: Dependency manager for PHP.
* **Usage**: Manages libraries and packages via composer.json.
* **Example**: composer install

**🟪 Perl**

**10. cpanm**

* **Purpose**: Installs Perl modules from CPAN.
* **Usage**: Lightweight and fast installer.
* **Example**: cpanm JSON

**🟫 Java**

**11. maven\_artifact**

* **Purpose**: Downloads artifacts (JARs) from Maven repositories.
* **Usage**: Useful in Java projects for dependency resolution.
* **Example**: Used in build automation and CI/CD pipelines.

**🟧 PHP Extensions**

**12. pear**

* **Purpose**: Installs PHP extensions and libraries.
* **Usage**: Older system, less common now due to Composer.
* **Example**: pear install HTTP\_Request2

**🔹 What are single module commands in Ansible?**

Single module commands are **ad-hoc commands** that use a specific Ansible module to perform a task on a group of hosts **without writing a playbook**. These are useful for quick tasks like installing packages, restarting services, copying files, etc.

The general syntax is:

A white background with blue and red text

AI-generated content may be incorrect.**✅ 1. Using the yum module:**

* **What it does**: Uses Ansible's built-in yum module to ensure the git package is installed on all hosts in the webservers group.
* **Advantages**:
  + Idempotent: Won’t reinstall if already present.
  + Structured: Uses Ansible's module system.
  + Portable: Works across systems with yum (like RHEL/CentOS).
  + Uses --become to run with elevated privileges (like sudo).

**⚠️ 2. Using a shell command:**

* A close-up of a white background

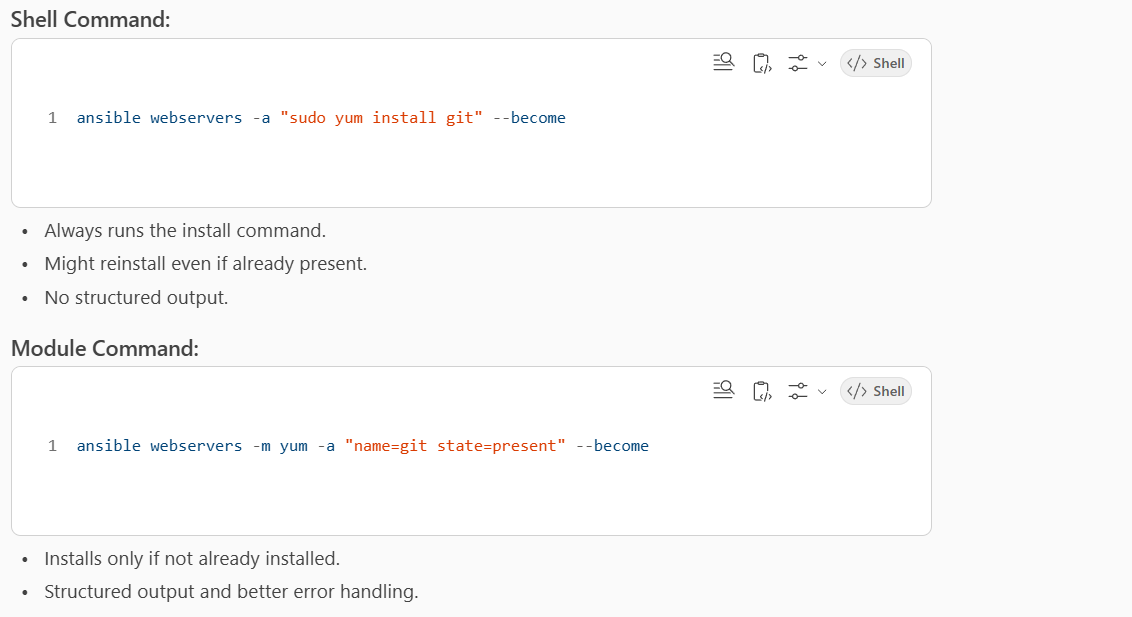
  AI-generated content may be incorrect.**What it does**: Runs a raw shell command on the remote hosts.
* **Drawbacks**:
  + Not idempotent: Will run every time, even if the package is already installed.
  + Less structured: No module logic or error handling.
  + Requires sudo to be configured properly on the remote host.
  + Doesn’t use Ansible modules, so lacks features like check mode, diff mode, etc.

**🔍 Difference Between Shell Commands and Module Commands in Ansible**

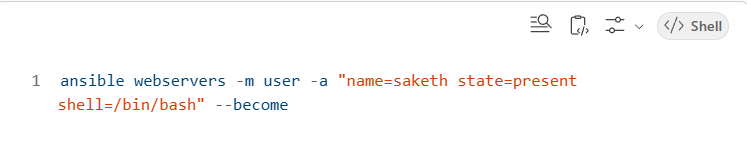
| **Feature** | **Shell Command (-a "command")** | **Module Command (-m <module> -a "args")** |
| --- | --- | --- |
| **Purpose** | Executes raw shell commands | Uses Ansible's structured modules |
| **Idempotency** | ❌ Not idempotent (runs every time) | ✅ Idempotent (runs only if needed) |
| **Error Handling** | Basic (depends on shell exit code) | Advanced (module returns structured output) |
| **Portability** | Depends on shell and OS | More portable across systems |
| **Logging & Output** | Raw stdout/stderr | Structured JSON output |
| **Check Mode Support** | ❌ No support | ✅ Supports --check mode |
| **Why Use It?** | Quick one-off tasks | Reliable, repeatable automation |

**✅ Why Use Module Commands Instead of Shell?**

1. **Idempotency**: Modules like yum, user, copy, etc., ensure that the desired state is achieved **only if needed**. For example, state=present won’t reinstall a package if it’s already installed.
2. **Error Handling**: Modules return structured data (JSON), making it easier to debug and log.
3. **Cross-Platform Compatibility**: Modules abstract away OS-specific details. For example, package module works across yum, apt, dnf, etc.
4. **Check Mode & Dry Runs**: Modules support --check to simulate changes without applying them.
5. **Better Integration**: Modules integrate with Ansible's inventory, facts, roles, and playbooks more cleanly.



to **create a user** named saketh on all hosts in the webservers group using Ansible's user module:

**🔍 Breakdown of the Command:**

* **ansible webservers**: Targets all hosts in the webservers group (defined in your inventory).
* **-m user**: Specifies the use of the user module.
* **-a "name=saketh state=present shell=/bin/bash"**:
  + name=saketh: The username to create.
  + state=present: Ensures the user exists (creates if not).
  + shell=/bin/bash: Sets the login shell for the user.
* **--become**: Runs the command with elevated privileges (like sudo), which is required for user creation.

**✅ What This Command Does:**

* Checks if the user saketh exists.
* If not, it creates the user with /bin/bash as the shell.
* If the user already exists, it ensures the shell is set correctly.
* It does **not** set a password, home directory, or group unless explicitly specified.

**✅ Example: Create an empty file**

Shell

ansible webservers -m file -a "path=/tmp/saketh.txt state=touch mode=0644" --become

Show more lines

**🔍 Explanation:**

* **-m file**: Specifies the use of the file module.
* **path=/tmp/saketh.txt**: The full path of the file to be created.
* **state=touch**: Ensures the file exists (creates it if it doesn’t).
* **state=delete**: to delete the file or **state=absent**
* **mode=0644**: Sets file permissions.
* **--become**: Runs the command with elevated privileges.

**📝 Notes:**

* This creates an **empty file**, similar to the touch command in Linux.
* If the file already exists, it won’t be modified unless you change other attributes like mode, owner, etc.

**✅ 1. Copy a file from control node to remote host — using copy module**

A close-up of a computer code

AI-generated content may be incorrect.**🔍 Explanation:**

* **src**: Path to the file on your **local machine** (control node).
* **dest**: Path on the **remote host**.
* **mode**, **owner**, **group**: Set file permissions and ownership.
* **--become**: Required if writing to system directories.

**🧾 Basic Structure of an Ansible Playbook**

Target section: where we can perform the tasks (which is nothing but the hosts)

Variable section: depends upon the client’s use case

Task section: the number of tasks are defined

Every ansible’s playbook starts with --- (3 hyphen)

Every playbook starts with: ---

This indicates the beginning of a YAML document.

**🔹 1. Target Section (hosts)**

Defines which machines the playbook will run on.

- hosts: webservers

hosts: Inventory group or hostnames.

A screenshot of a computer program

AI-generated content may be incorrect.

**✅ Workflow: Using VS Code, Git, and Ansible for Automation**

**🔹 1. Write Ansible Playbook in VS Code**

* VS Code provides syntax highlighting, linting, and extensions for YAML and Ansible.
* You write your playbook (e.g., example-test.yaml) with tasks to automate server setup.

**🔹 2. Install Git on Your Local Machine**

* Git tracks changes to your playbook and other files.
* You can version control your automation scripts.

**🔹 3. Integrate Git with VS Code**

* Use the built-in Git panel in VS Code to:
  + Stage changes (git add)
  + Commit changes (git commit -m "message")
  + Push to remote (git push origin <branch>)

**🔹 4. Clone the Git Repo on the Server**

* Use git clone <github-repo-link>
* On the server (or via Ansible), pull the latest changes: git pull

**🔹 5. Run the Playbook**

* You can run the playbook with different modes:

A close-up of a white background

AI-generated content may be incorrect.

**✅ Color Indicators in Ansible Output**

* **Yellow**: Task was skipped or already in desired state (no change).
* **Green**: Task was executed and changes were made.
* **Red**: Task failed.

there are 3 stages in the git lifecycle:

The key stages include the **Working Directory**, **Staging Area**, **Local Repository**, and **Remote Repository**.

**Working Directory**

The **Working Directory** is where files are created or modified. At this stage, Git is aware of the files but does not track them yet. To check the status of files, you can use:

git status

**Staging Area**

The **Staging Area** is a temporary space where changes are prepared for a commit. You can selectively add files to the staging area using:

git add <filename> # Add a specific file

git add . # Add all modified files

This step allows you to organize changes before committing them.

**Commit Changes**

Once files are staged, they can be committed to the **Local Repository**. A commit represents a snapshot of the project at a specific point in time. Each commit includes metadata such as the author, timestamp, and a message describing the changes. To commit changes, use:

git commit -m "Your commit message"

This saves the changes locally, creating a version that can be referenced later.

**Push to Remote Repository**

After committing changes locally, you can share them with collaborators by pushing them to a **Remote Repository** (e.g., GitHub, GitLab). This step uploads your commits to a central repository, making them accessible to others. To push changes, use:

git push origin <branch-name>

Replace <branch-name> with the name of the branch you want to push to (e.g., main or develop).

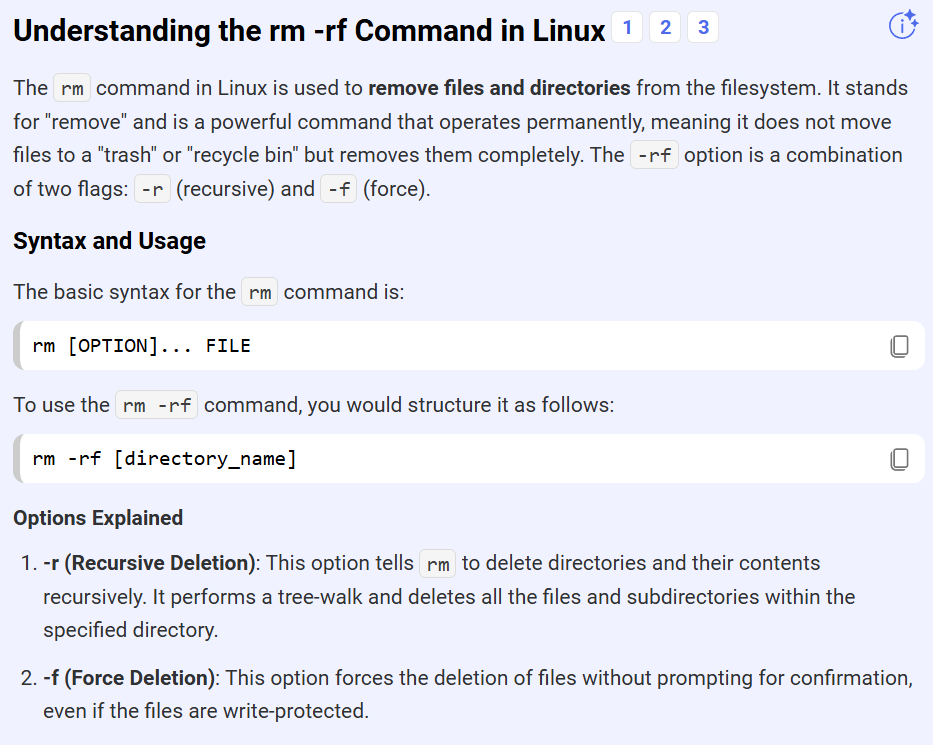
**Pull Before Push**

Before pushing, it is recommended to pull the latest changes from the remote repository to ensure your local branch is up-to-date. This avoids conflicts and ensures a smooth collaboration process:

git pull origin <branch-name>

**Summary**

The Git lifecycle ensures a systematic approach to version control. The **commit** operation saves changes locally, while the **push** operation shares them with others via a remote repository. By following this workflow, developers can efficiently track changes, collaborate, and maintain a clean project history.



Task

Write a playbook to install java, python , webserver , start webserver, spot , restart and install mysql

Clone code from github repo using playbook and deploy the webserver in httpd using playbook

**✅ What is yum?**

* **YUM** stands for **Yellowdog Updater Modified**.
* It's the traditional package manager for **RHEL-based systems** (Red Hat, CentOS, Amazon Linux).
* It handles package installation, updates, and dependency resolution.

**✅ What is dnf?**

* **DNF** stands for **Dandified YUM**.
* It is the **next-generation replacement for yum**, introduced in **RHEL 8 and Fedora**.
* It offers better performance, cleaner dependency management, and a more robust API.

**🔍 Key Differences**

| **Feature** | **yum** | **Dnf** |
| --- | --- | --- |
| **Introduced in** | RHEL 5/6/7 | RHEL 8+, Fedora 22+ |
| **Performance** | Slower | Faster and more efficient |
| **Dependency Handling** | Basic | Improved and more accurate |
| **Python API** | Legacy | Modern and extensible |
| **Backward Compatible** | Yes | Mostly compatible with yum |

**✅ What is RHEL?**

**RHEL** stands for **Red Hat Enterprise Linux**. It is:

* A **commercial Linux distribution** developed by **Red Hat Inc.**
* Widely used in **enterprise environments** for servers, cloud infrastructure, and data centers.
* Known for its **stability**, **security**, and **long-term support**.
* Comes with **subscription-based support**, including updates, patches, and certified software.

RHEL is the upstream source for other popular distributions like **CentOS**, **Rocky Linux**, and **AlmaLinux**.

**✅ Is dnf an Evolution of yum?**

Yes — **dnf is the modern replacement for yum**.

**🔍 Why dnf was introduced:**

* yum had limitations in performance and dependency resolution.
* dnf was built to be **faster**, **more reliable**, and **easier to maintain**.

**🔸 Key Improvements in dnf over yum:**

| **Feature** | **yum** | **Dnf** |
| --- | --- | --- |
| Dependency resolution | Basic | More accurate and efficient |
| Performance | Slower | Faster and more optimized |
| API support | Legacy Python API | Modern Python API |
| Plugin system | Limited | More flexible and modular |
| CLI consistency | Sometimes inconsistent | More predictable and robust |

**✅ What is GPG?**

**GPG** stands for **GNU Privacy Guard**. It is a tool used for **encryption**, **signing**, and **verifying data and communications**. In the context of Linux and package management, GPG is primarily used to:

* **Sign software packages**.
* **Verify the authenticity and integrity** of those packages before installation.

**🔍 Why is GPG important in Ansible and Jenkins setup?**

When you install Jenkins (or any software) from a third-party repository, you want to make sure:

1. The package **has not been tampered with**.
2. It **comes from a trusted source**.

This is done using a **GPG key** provided by the software vendor (like Jenkins). The system uses this key to verify the signature of the package before installing it.

**✅ 1. service Module**

**🔹 Purpose:**

Manages services (like httpd, jenkins, nginx, etc.) using the system's init system (like systemd or SysV).

**🔍 What it does:**

* Starts the jenkins service.
* If it's already running, it does nothing (idempotent).
* You can also use state: stopped, restarted, or reloaded.

**✅ 2. systemd Module**

**🔹 Purpose:**

Specifically interacts with **systemd**, the modern init system used in most Linux distributions (RHEL 7+, CentOS 7+, Ubuntu 16.04+).

**🔍 What it does:**

* Ensures the jenkins service is **enabled to start on boot**.
* You can also manage service state (started, stopped, etc.), but service is more general-purpose.

**🔸 When to use systemd over service:**

* When you need **systemd-specific features** like enabled, masked, daemon\_reload, etc.
* For newer systems where systemd is the default.

In **Ansible**, a **handler** is a special type of task that is **triggered only when notified by another task**. Handlers are typically used for actions that should only happen if something changes — for example, restarting a service after a configuration file is updated.

**✅ Why Use Handlers?**

* To avoid unnecessary actions (like restarting a service when nothing changed).
* To ensure tasks are executed **only when needed**.
* To improve performance and idempotency.

**🔹 How Handlers Work**

1. A regular task includes a notify directive.
2. If that task makes a change, it **notifies** the handler.
3. The handler runs **at the end of the play**.

**✅ Why Use a Handler Here?**

* You might want to **restart Docker only if it was changed** (e.g., newly started or reconfigured).
* This avoids unnecessary restarts, which is good for performance and stability.

**📝 Summary**

| **Component** | **Purpose** |
| --- | --- |
| notify: Restart Docker | Tells Ansible to run the handler if the task changes something. |
| handlers: section | Defines what to do when notified (restart Docker). |
| Execution timing | Handlers run **at the end of the play**, not immediately. |

TASK:

1. Write a ansible playbook to understand the when condition, I have to install a specific package on host like git or java or python if the package is already installed in the host machine we have to skip the task and we have to execute the next task.

**🔐 What is Ansible Vault?**

**Ansible Vault** is a feature that allows you to **securely store sensitive data** such as:

* Passwords
* API keys
* SSH credentials
* Any confidential variables or files

It encrypts files using a password so that only authorized users can view or use them.

**🔧 Other Useful Vault Commands**

| **Command** | **Purpose** |
| --- | --- |
| ansible-vault create <file> | Create and encrypt a new file |
| ansible-vault edit <file> | Edit an encrypted file |
| ansible-vault view <file> | View contents of an encrypted file |
| ansible-vault decrypt <file> | Decrypt a file |
| ansible-playbook <playbook>.yaml --ask-vault-pass | Run a playbook that uses encrypted variables |
| ansible-vault rekey <playbook.yml> | Change the password |
| ansible-vault encrypt <playbook.yml> | Encrypt an existing playbook (verify by opening playbook) |

**🛡️ Why Use Vault?**

* Keeps secrets out of plain text files.
* Prevents accidental exposure of credentials in version control (like Git).
* Integrates seamlessly with Ansible playbooks and roles.

# **Infrastructure:**

Infrastructure is an environment where we can deploy the application and it Combination of people process network

**✅ AWS Lambda**

**🔹 What is it?**

* **Lambda** is a **serverless compute service** by AWS.
* You run code **without provisioning or managing servers**.
* You just upload your function, and AWS handles the rest — scaling, availability, and infrastructure.

**🔍 Key Features:**

* **Event-driven**: Triggered by events (e.g., S3 uploads, API Gateway calls).
* **Pay-per-use**: You only pay for the time your code runs.
* **Supports multiple languages**: Python, Node.js, Java, Go, etc.

**🔸 DevOps Use Case:**

* Automate tasks like image processing, log analysis, or CI/CD hooks.
* Lightweight microservices or backend logic.

**✅ AWS Elastic Beanstalk**

**🔹 What is it?**

* A **Platform-as-a-Service (PaaS)** offering from AWS.
* It lets you **deploy and manage web applications** without worrying about the underlying infrastructure.

**🔍 Key Features:**

* Supports multiple platforms: Java, Python, Node.js, PHP, .NET, etc.
* Automatically handles:
  + Load balancing
  + Auto-scaling
  + Monitoring
  + Updates

**🔸 DevOps Use Case:**

* Ideal for deploying full-stack web applications.
* Simplifies deployment pipelines.
* Great for teams that want to focus on code, not infrastructure.

Cache memory – storing the temporary memory

## ✅ Networking Options on AWS

These services help manage **traffic flow**, **connectivity**, and **security** across your cloud infrastructure:

**1. VPC (Virtual Private Cloud)**

* Your own isolated network within AWS.
* You define subnets, route tables, gateways, and security groups.
* Foundation for secure and scalable infrastructure.

**2. Route 53**

* AWS’s **DNS service**.
* Maps domain names to IP addresses.
* Supports traffic routing, health checks, and domain registration.

**3. API Gateway**

* Manages and routes **HTTP requests to backend services** (like Lambda, EC2).
* Ideal for building and exposing RESTful APIs.
* Supports throttling, caching, and authorization.

**4. AWS App Mesh**

* A **service mesh** for microservices.
* Controls and configures traffic between services.
* Enables observability, retries, and traffic shifting.

**5. CloudFront**

* AWS’s **Content Delivery Network (CDN)**.
* Distributes static and dynamic content globally with low latency.
* Works well with S3, EC2, and custom origins.

### VPC – virtual private cloud

* Isolated Cloud Resource. Deploying the server in a secure network.
* We create to protect the network from hacker.
* If we wan to deploy any application in any cloud without creating a VPC we will never deploy any application in any cloud .
* Aws services -> VPC -> your VPC-> create VPC -> enter name tag -> default or dedicated tenancy -> create VPC -> Subnets -> subnet name ->

**✅ What is DHCP?**

**DHCP** stands for **Dynamic Host Configuration Protocol**.

It is a **network management protocol** used to **automatically assign IP addresses and other network configuration parameters** (like subnet mask, gateway, DNS) to devices on a network.

* Every VPC must be under a region

**✅ What is a Subnet Mask?**

A **subnet mask** is a 32-bit number used in IP networking to **divide an IP address into two parts**:

1. **Network portion**: Identifies the network.
2. **Host portion**: Identifies individual devices (hosts) within that network.

**✅ Public IP Address**

* A **public IP** is an address that is **accessible over the internet**.
* It is assigned by **DHCP (Dynamic Host Configuration Protocol)** in most cloud environments.
* **Dynamic**: It can change when the instance is stopped and started again.
* Used when you want to **expose a service or application to the internet** (e.g., a website, API).

**🔸 Problem:**

Every time you restart your instance, the public IP may change.

**✅ Solution: Elastic IP**

* An **Elastic IP** is a **static public IP** provided by AWS.
* You can **associate it with any EC2 instance**.
* It remains the same even if the instance is stopped or restarted.
* Ideal for applications that need a **consistent public endpoint**.

**✅ Private IP Address**

* A **private IP** is used **within a VPC or local network**.
* It is **not accessible from the internet** directly.
* Assigned by DHCP but remains **constant** for the life of the instance (unless manually changed).
* Used for internal communication between services (e.g., database, backend services).

CIDR- Classless inter domain routing. Automatically assigns the IP to the server.

Default tenancy – multiple people sharing the same hardware.

**✅ What is a Subnet?**

A **subnet** (short for **subnetwork**) is a **logical subdivision of an IP network**. It helps organize and manage network traffic more efficiently by breaking a large network into smaller, more manageable segments.

* To **allocate IP addresses** more efficiently.
* To **control access** between different parts of a network.
* every subnet will have a unique IP address.

Subnet is a logical sub division of logical VPS’s IP address. Each subnet is confined with to a single Availability Zone.

**🔹 Types of Subnets in AWS**

**1. Public Subnet**

* Has access to the **internet** via an **Internet Gateway**.
* Used for resources that need to be publicly accessible (e.g., web servers).
* Instances in public subnets typically have **public IP addresses**.

**2. Private Subnet**

* **No direct internet access**.
* Used for internal resources like databases, backend services, etc.
* More secure because they are **isolated from the public internet**.
* if we want to get internet access to the private subnet we use NAT gateways

every sub net will associate with a routing table

routing table: which traffic we want to re direct to the VPC or our application.

In **DevOps**, **tunneling** refers to the technique of creating a secure, encrypted connection between two endpoints, often to **access internal resources** (like servers, databases, or services) that are not directly exposed to the public internet.

**🔐 What is Tunneling?**

Tunneling encapsulates one network protocol within another. It’s commonly used to:

* **Secure communication** over untrusted networks (like the internet)
* **Bypass firewalls or NAT** restrictions
* **Access private resources** in a VPC or internal network

## ✅ Database Services on AWS

These services store and manage data for your applications:

**1. RDS (Relational Database Service)**

* Managed relational databases (MySQL, PostgreSQL, Oracle, SQL Server).
* Stores **structured data** in tables.
* Handles backups, patching, scaling, and replication.

**2. DynamoDB**

* AWS’s **NoSQL database**.
* Stores **unstructured or semi-structured data** (key-value or document-based).
* Highly scalable, low-latency, and serverless.

SNS: simple notification service

Containers: elastic Kubernetes service, Red Data Openshift Services on AWS

For shifting of the service from on premise to cloud we can use migration and transfer.

## ✅ AWS DevOps Tools Overview

These services help automate the **build**, **test**, **release**, and **deployment** phases of your application lifecycle.

**1. CodeCommit**

* A **fully managed Git-based source control** service.
* Stores your code securely.
* Supports versioning, branching, and collaboration.

**2. CodeBuild**

* A **build service** that compiles source code, runs tests, and produces deployable artifacts.
* Scales automatically and integrates with CodeCommit, S3, and CodePipeline.

**3. CodeDeploy**

* Automates **deployment of applications** to EC2, Lambda, or on-prem servers.
* Supports rolling updates, blue/green deployments, and rollback.

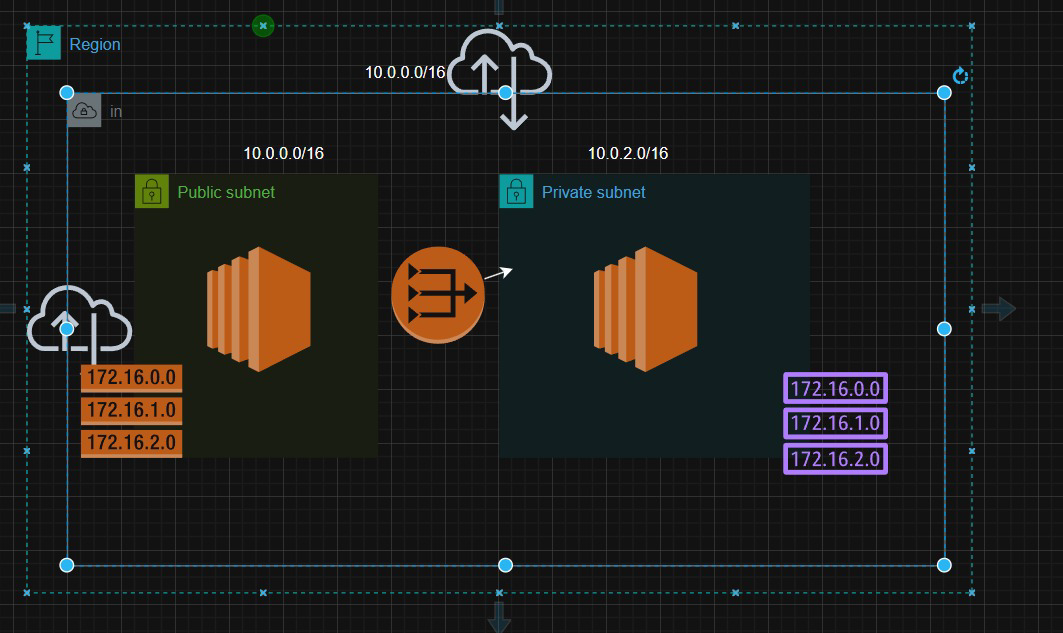
**4. CodePipeline**

* A **CI/CD orchestration tool**.
* Connects all stages: source → build → test → deploy.
* Integrates with CodeCommit, CodeBuild, CodeDeploy, and third-party tools.

**5. CodeArtifact *(optional but useful)***

* A **package management service** for storing and sharing software packages (npm, Maven, PyPI).
* Useful for dependency management in builds.

# **Task-Pvt & Pub**



Create VPC -> name Server-2 -> IPv4 – 10.0.0.0/16

Create subnets -> public -> subnet CIDR 10.0.1.0/24

Private ->subnet CIDR 10.0.2.0/24

Create route table -> public -> under VPC server-2

private -> under VPC server-2

internet gateway -> to VPC server-2

create NAT gateway -> link the public subnet

A screenshot of a computer

AI-generated content may be incorrect.

In the server

For amazon linux

Yum install httpd -y

For ubuntu:

apt-get install apache2 -y

for cloning the git repo in linux or ubuntu

git clone -b master <git-link>

copy the files to the html file

cp -r . /var/www/html

now to run the file in the ubuntu server (which is a private):

if we deploy any application in private servers, if we want to access them to the web browser there is only one way through the load balancer.

Load balancer will create in a public subnets.

# **Terraform**

## AWS Cloud formation VS Terraform

| **Feature** | **Terraform** | **AWS CloudFormation** |
| --- | --- | --- |
| **Cloud Support** | Multi-cloud (AWS, Azure, GCP, etc.) | AWS only |
| **Language** | HCL (HashiCorp Configuration Language) | YAML or JSON |
| **State Management** | External state file (e.g., stored in S3) | Managed by AWS |
| **Modularity** | Modules | Nested stacks |
| **Change Preview** | terraform plan | Change Sets |
| **Community Support** | Large open-source community | AWS ecosystem |
| **CLI Tooling** | terraform CLI | aws cloudformation CLI |
| **Learning Curve** | Easier syntax, more flexible | AWS-specific, more rigid |
| **Third-party Integrations** | Extensive | Limited to AWS services |
| **Error Handling** | Manual rollback | Automatic rollback on failure |

Terraform has terraform state file which store all the data about the file.

All the values must be in the form of variables (dynamic). Cause in various environment there would be various environments

Terraform validate :- checks if the code syntax is correct or not

When terraform plan is used then only for few seconds the terraform.tfstate file is created to check if all the files are present.

## Terrafrom Variables:

In **Terraform**, **variables** are used to make your infrastructure code **reusable**, **dynamic**, and **easier to manage**. Instead of hardcoding values (like region names, instance types, or IP addresses), you define variables and pass values to them.

**🧠 Why Use Variables?**

* Avoid duplication
* Make modules reusable
* Easily switch environments (dev, staging, prod)
* Centralize configuration

**📦 Types of Variables**

Terraform supports several types:

* string
* number
* bool
* list
* map
* object
* tuple

**🔹 Terraform Modules**

A **module** in Terraform is a container for multiple resources that are used together. It’s essentially a way to **group and reuse code**.

**Types of Modules:**

1. **Root Module**: The main module in your Terraform configuration (usually in the working directory).
2. **Child Module**: A module called from another module (can be local or remote).

**Why Use Modules?**

* **Reusability**: Define infrastructure once and reuse it across environments.
* **Organization**: Break complex setups into manageable pieces.
* **Maintainability**: Easier to update and version control.

**🔹 1. Terraform Modules (Deep Dive)**

**✅ What Are Modules?**

Modules in Terraform are like reusable components. They group related resources together so you can reuse them across projects or environments.

**✅ Why Use Modules?**

* **Reusability**: Write once, use many times.
* **Maintainability**: Easier to manage and update.
* **Organization**: Clean separation of concerns (e.g., VPC, EC2, RDS as separate modules).

A screenshot of a computer

AI-generated content may be incorrect.

**🔹 2. terraform.tfvars File**

**✅ What Is It?**

The terraform.tfvars file is used to **assign values to variables** declared in variables.tf. It helps separate configuration from code.

**✅ Why Use It?**

* Keeps your code clean.
* Makes it easy to switch environments (e.g., dev, staging, prod).
* Avoids hardcoding values in .tf files.

# **Docker:**

**Docker** is an open-source platform that helps developers **build, ship, and run applications** in a consistent environment using **containers**.

**Key Concepts:**

* **Containers**: Lightweight, standalone, and executable packages that include everything needed to run a piece of software—**code, runtime, libraries, and dependencies**. Think of them as mini virtual machines, but much more efficient.
* **Images**: A Docker image is a **blueprint** for a container. It defines what’s inside the container, like the OS, application code, and dependencies.
* **Docker Engine**: The runtime that builds and runs containers.
* **Dockerfile**: A text file with instructions to build a Docker image. It defines the environment and steps to set up your application.

**Why Use Docker?**

* **Consistency**: Run the same application across different environments (dev, test, prod) without "it works on my machine" issues.
* **Isolation**: Each container runs independently, avoiding conflicts between apps.
* **Portability**: Containers can run on any system that supports Docker.
* **Efficiency**: Containers are lightweight and start quickly compared to traditional virtual machines.

## monolithic application

A **monolithic application** is a type of software architecture where **all the components of the application are built and deployed as a single unit**.

**🔍 Key Characteristics of a Monolithic Application:**

1. **Single Codebase**: All the functionality—UI, business logic, and data access—is part of one large codebase.
2. **Tightly Coupled**: Components are interconnected and interdependent, making it harder to isolate and modify individual parts. Especially during the upgrades V1 – V2
3. **Single Deployment**: The entire application is deployed at once. Even a small change requires rebuilding and redeploying the whole system.
4. **Shared Resources**: Often uses a single database and shared memory space.
5. **Autoscaling**: always Horizontal scaling is preferred such as Scale up and Scale down.
6. New Release:

**🧱 Example:**

Imagine an e-commerce application. In a monolithic architecture, the following modules would all be part of one codebase and deployed together:

* User authentication
* Product catalog
* Shopping cart
* Payment processing
* Order management

**✅ Advantages:**

* **Simplicity**: Easier to develop and test in the early stages.
* **Performance**: Function calls within the same process are fast.
* **Deployment**: Only one deployment unit to manage.

## Microservices

**Microservices** is an architectural style where a **large application is broken down into smaller, independent services**, each responsible for a specific business function.

Each microservice:

* **Runs independently**
* **Has its own database (ideally)**
* **Communicates with other services via APIs (usually HTTP/REST or messaging queues)**

**Key Characteristics:**

| **Feature** | **Description** |
| --- | --- |
| **Independence** | Each service can be developed, deployed, and scaled independently. |
| **Specialization** | Each service focuses on a single business capability (e.g., user service, payment service). |
| **Technology Agnostic** | Different services can be written in different programming languages or use different databases. |
| **Resilience** | Failure in one service doesn’t necessarily bring down the whole system. |
| **Scalability** | You can scale individual services based on demand instead of scaling the entire application. |

**Example: E-commerce Application**

Instead of one big monolithic app, you might have:

* **User Service** – handles user registration and login
* **Product Service** – manages product catalog
* **Order Service** – processes orders
* **Payment Service** – handles transactions
* **Notification Service** – sends emails or SMS

Each of these can be deployed and updated independently.

**Benefits:**

* Faster development and deployment
* Easier to maintain and scale
* Better fault isolation
* Enables use of different tech stacks
* Roll back: upgrade V1 – V2 in the production environment. If a version fails then we can easily go back to the previous version.
* Simple database: each micro service has it’s own database
* Autoscaling: during the times the memory or the user capacity can be increased such as with Flipkart during the Big Billion days sale at the End user Scale up and Scale Down occurs.
* Better alignment with Buisness
* Loose Coupled

**Challenges:**

* Complexity in managing multiple services
* Requires robust DevOps and monitoring
* Network latency and inter-service communication issues
* Data consistency can be tricky

### 🔌 What is an API?

**API** stands for **Application Programming Interface**. It's like a **messenger** that allows different software systems (or parts of a system) to **talk to each other**.

In the context of microservices or even monolithic apps, an API is how one part of your system **exposes functionality** to other parts or to external systems.

**🧩 How APIs Fit into Microservices**

In a **microservices architecture**, each service is **independent** and often runs on its own server or container. These services **communicate via APIs**, usually over **HTTP** using **REST** or **GraphQL**.

For example:

* Your **frontend** (React/Angular) calls the **User Service API** to log in.
* It then calls the **Product Service API** to fetch product listings.
* Later, it calls the **Order Service API** to place an order.

Each of these services is a separate codebase, possibly in different languages, but they all expose **APIs** to interact with each other.

**🔄 Is it like converting code into an API?**

Yes, in a way! Here's how it works:

1. **You write some logic** (e.g., a function to get user data from a database).
2. You **wrap that logic in an API endpoint** using a web framework (like Flask for Python, Express for Node.js, Spring Boot for Java).
3. Other parts of your system (or external systems) can now **call that endpoint** over HTTP to use your logic.

## 🆚 Monolithic vs Microservices

| **Feature** | **Monolithic** | **Microservices** |
| --- | --- | --- |
| Architecture | Single unit | Multiple independent services |
| Deployment | One deployment | Independent deployments |
| Scalability | Entire app scales together | Individual services can scale separately |
| Technology Stack | Usually one | Can use different stacks per service |
| Maintenance | Harder as app grows | Easier to manage smaller services |

**🧱 Monolithic Application**

Think of it like a **single big project** in VS Code where:

* All your **frontend**, **backend**, **business logic**, and **database access code** are in **one codebase**.
* You **build and deploy** the entire application as **one unit**.
* If you want to update just the login feature, you still have to redeploy the whole app.

🔧 **Example**:  
A Java Spring Boot application with everything—user login, product catalog, payment, etc.—in one .jar or .war file.

**🧩 Microservices Architecture**

Now imagine breaking that big app into **smaller, independent services**, each doing one job well:

* One service handles **user authentication**.
* Another handles **product catalog**.
* Another handles **payments**.
* Each service can be:
  + Written in **different languages** (e.g., Node.js for one, Python for another).
  + Deployed **independently** (e.g., using Docker containers).
  + Hosted on **different platforms** (e.g., AWS Lambda for one, EC2 for another).
  + Communicate via **APIs** (usually REST or gRPC).

🔧 **Example**:  
Your frontend (React) calls a Node.js API for login, which talks to a MongoDB on AWS. Meanwhile, your order service is in Python and deployed on a different server.

**🚢 How Docker & Containerization Fit In**

**🔄 The Problem Before Containers**

Before Docker and containers, deploying applications was messy:

* Developers would build apps on their machines, but they might not work the same in production due to **different environments**.
* Scaling required spinning up **entire virtual machines (VMs)**, which are **heavy**, slow to start, and resource-intensive.
* Managing dependencies and versions across environments was painful.

**🧱 Enter Containerization**

**Containerization** is a way to **package your application and all its dependencies** into a single, lightweight unit called a **container**.

* Think of a container as a **mini virtual machine**, but much more efficient.
* It shares the host OS kernel, so it’s **faster and lighter** than a VM.
* You can run **multiple containers** on the same machine, each isolated from the others.

**🐳 Docker’s Role**

**Docker** is the most popular tool for containerization. It helps you:

1. **Build** containers using a Dockerfile.
2. **Run** containers using the Docker Engine.
3. **Share** containers via Docker Hub or private registries.
4. **Deploy** containers on any environment (local, cloud, CI/CD).

**🔗 How APIs and Docker Work Together**

Let’s say you have a **User Service API** written in Node.js.

1. You **wrap the code in an API** using Express.
2. You **containerize** it using Docker:
   * Write a Dockerfile that installs Node.js, copies your code, and runs the server.
3. You **run the container** locally or deploy it to the cloud.
4. Other services (like frontend or Order Service) **call the API** via HTTP.

This makes your service **portable**, **scalable**, and **easy to deploy**.

**📈 Scaling with Containers**

When traffic increases, you need to **scale**:

* With Docker, you can **spin up multiple containers** of the same service.
* Tools like **Kubernetes** or **Docker Swarm** help manage this:
  + **Auto-scaling**: Add more containers when load increases.
  + **Load balancing**: Distribute traffic across containers.
  + **Self-healing**: Restart failed containers automatically.

Containers can be **terminated** to save resources after requirement.

**🧱 Docker Image = Blueprint**

* A **Docker image** is like a **template** or **blueprint** of your application.
* It includes:
  + Your **application code**
  + All **dependencies** (libraries, runtimes)
  + Configuration files
  + Environment variables
  + Instructions on how to run the app (via CMD or ENTRYPOINT)

**🚀 Container = Running Instance**

* A **container** is a **running instance** of a Docker image.
* It’s **isolated**, **lightweight**, and **portable**.
* You can run **many containers** from the same image.

**⚙️ No Manual Resource Allocation**

* Docker containers **share the host OS kernel**, so:
  + No need to install a full guest OS like in virtual machines.
  + They start **very fast** (in milliseconds).
  + They use **only the resources they need**, and you can **limit CPU/memory** if required.

**🧠 Why This Changed the Game**

Before Docker:

* You had to configure servers manually.
* Apps would break when moved between environments.
* Scaling was slow and expensive (spinning up full VMs).

With Docker:

* **Build once, run anywhere**.
* **Fast, consistent deployments**.
* **Efficient scaling** with minimal overhead.

## Creating a Docker Container using the EC2 (VM)

* To install docker on the virtual machine = yum install docker -y
* To start the docker service = service docker start
* To create a docker (ubuntu docker container) = docker create ubuntu /bin/bash
* To view all the docker container irrespective of the status = docker ps -a
* View only the active docker containers = docker ps
* Allows us to enter the docker created = docker exec -it webserver-saketh-1 /bin/bash
* Creates the docker and enter = docker run -td --name webserver-saketh-1 ubuntu /bin/bash
* To exit from the docker and enter VM = exit
* To stop the docker container = docker stop <container-name>
* To remove the docker container = docker rm <container-name>
* To view all the docker images = docker images
* To remove the docker image = docker rmi -f <container-id>
* To stop all the docker containers = docker stop $(docker ps -a -q)
* To remove all the docker containers = docker rm $(docker ps -a -q)

To remove all the docker containers = docker rmi -f $(docker images -q)

docker build -t cart-app-node-task:v1.1.0 .

docker run -td --name cart-app-node-task -p 3000:3000 cart-app-node-task:v1.1.0

docker tag cart-app-node-task:v1.1.0 saketh1809/cart-app-node-task:v1.1.0

docker push saketh1809/cart-app-node-task:v1.1.0

docker run -td --name webserver-nginx-saketh -p 8081:80 nginx

docker run -td --name webserver-niginx -p 8080:8080 jenkins/jenkins

docker run -td --name webserver-sonarqube -p 9000:9000 sonarqube

## 📄 What is a Dockerfile?

A **Dockerfile** is a **text file** that contains a **set of instructions** to build a Docker image.

It tells Docker:

* What base image to use (e.g., Ubuntu, Node.js)
* What files to copy
* What commands to run
* What ports to expose
* What command to execute when the container starts

Dockerfile is a text document it contains set of instructions. It tells you that how to run the application in a container

### Dockerfile Instructions:

the **Dockerfile**, which is a **text document** containing a **set of instructions** that tells Docker **how to build a Docker image** and **run your application inside a container**.

#### **1. FROM**

* Defines the **base image** for your application.
* Every Dockerfile **must start** with a FROM instruction.
* In a single dockerfile we can have multiple FROM instruction
* It defines the parent image or base image of the application
* Examples:
  + FROM node:latest
  + FROM openjdk:latest
  + FROM python:latest
* You can use multiple FROM instructions for **multi-stage builds**.
* **Why**: It sets the environment (OS + language/runtime) your app will run in
* Alternate sources:
  + quay.io/node:latest
  + docker.io/library/node:latest

#### **2. COPY**

* Copies files from your **local machine** into the **Docker image**.
  + COPY <src> <dest>
  + . represents the current directory

A white background with black text

AI-generated content may be incorrect.

#### **3. RUN**

* Executes commands **inside the container** during the image build.
* To run a command in the container
* If we want to install dependencies of the applications or any specific package then we are using run instructions
* **Common Use**: Install packages, set up environment.

**A white background with black and white clouds

AI-generated content may be incorrect.**

A close-up of a text

AI-generated content may be incorrect.

#### **4.🌐 ADD**

* **Purpose**: Similar to COPY, but with **extra features**:
  + Can **download files from URLs**
  + **We can even copy the .tar walls, Zip files**
  + Can **extract compressed files** (like .tar.gz)

A close-up of a computer code

AI-generated content may be incorrect.

#### **5.👤 MAINTAINER (Deprecated in favor of LABEL)**

* **Purpose**: Specifies the **author** of the Docker image.
* **To assign name or email of the user**

A close-up of a website

AI-generated content may be incorrect.

#### **6.🚫 .dockerignore**

* **Purpose**: Works like .gitignore. It tells Docker which files/folders to **exclude** when building the image.
* **Example** (.dockerignore file):

A white background with black lines

AI-generated content may be incorrect.

#### **6. ONBUILD**

* **Purpose**: Defers execution of instructions until the image is used as a base in another Dockerfile.
* **Why**: Useful for **base images** that trigger actions in child images.

A white background with black and white clouds

AI-generated content may be incorrect.

#### **1. CMD**

* **Purpose**: Sets the **default command** to run when the container starts.
* **It is a default executor**. When ever we deploy a docker image in a container this command will execute automatically.
* **It will accept run time arguments** during the build.
* **Example**:

A white background with black and white clouds

AI-generated content may be incorrect.

#### **2. ENV**

* **Purpose**: Sets **environment variables** inside the container.
* **Environment variable are nothing but the backend url, secret keys,**
* **NOTE:** we are going to store all the environment variables in vaults
* **Example**:

A white background with black and white clouds

AI-generated content may be incorrect.

#### **3. EXPOSE**

* **Purpose**: Declares which **ports** the container listens on.
* **On which port number we want to access the application through the browser**
* **Example**:

A white background with black and white clouds

AI-generated content may be incorrect.**Note**: You still need to map ports using -p or -P when running the container.

#### **4. VOLUME**

* **Purpose**: Declares a **mount point** for persistent or shared data.
* **If we store the data in a container, once you lost the container we will also loose the data as well**. To take the backup of data, to store in a secure place we use VOLUME. It is a directory out side the container.
* **Example**:

A white background with black and white lines

AI-generated content may be incorrect.

#### **5. ENTRYPOINT**

* **Purpose**: Sets a command that **cannot be overridden** easily.
* **It is a Default executor, we are going to use to run a specific code in a container**
* **It will not accept any run time arguments**
* **Example**:
* A close-up of a computer screen

  AI-generated content may be incorrect.**Why**: Useful for setting a fixed executable and passing arguments via CMD.

#### **1. USER**

* **Purpose**: Sets the **user** to run commands.
* **To run an application as a non**-root user account we are using USER command
* **Example**:

A white background with black and white clouds

AI-generated content may be incorrect.

#### **2. WORKDIR**

* **Purpose**: Sets the **working directory** for subsequent instructions.
* **To store all source code, environment variables, everything inside the work directory**.
* **Every project will have separate work directory where exactly we can copy the files**.
* **Example**

A white background with black and white clouds

AI-generated content may be incorrect.

**🔄 Build vs Run Instructions**

**🧱 Build Instructions**

These are executed **when building the image** using docker build.

| **Instruction** | **Purpose** |
| --- | --- |
| FROM | Base image |
| COPY / ADD | Copy files into image |
| RUN | Run commands during build (e.g., install packages) |
| WORKDIR | Set working directory |
| .dockerignore | Ignore files during build |

**🚀 Run Instructions**

These are used **when the container starts** using docker run.

| **Instruction** | **Purpose** |
| --- | --- |
| CMD | Default command to run |
| ENTRYPOINT | Main command that can't be overridden easily |
| EXPOSE | Declare which ports the container listens on |
| ENV | Set environment variables |
| VOLUME | Declare persistent storage |

**🧠 Summary**

| **Concept** | **Description** |
| --- | --- |
| **Docker** | Platform to run apps in containers |
| **Dockerfile** | Script to build Docker images |
| **Build** | Happens during docker build, sets up the image |
| **Run** | Happens during docker run, starts the container |

### 7 steps while creating a dockerfile:

* + 1. Find the base image of the application
    2. Set up a work directory
    3. Copy dependencies into the docker image
    4. Install the dependencies to the application
    5. Copy whole code into docker image
    6. Define the expose instructions
    7. Define how to run the application in a container

apt-get update && apt-get install git -y

git clone -b master <https://github.com/Msocial123/Capgemini-Retail-Project.git>

cp -r Capgemini-Retail-Project/\* /usr/local/apache2/htdocs/

platform independent – when one application is run on a system will require to install all the dependencies and on another system we have to re build all the dependencies again

instead if we build a docker file the application can be containerised so that all the dependencies will also be installed automatically

when ever we have upgrade we rebuild the docker image, the name of the docker remains same but the tags will be updated.

Elastic container registry

Elastic Container Registry (ECR) is a fully managed Docker container registry provided by AWS. It allows developers to store, manage, and deploy Docker container images securely. In a CI/CD pipeline, ECR is typically used to store the Docker images built from your application code, which are then deployed to environments like Amazon ECS, EKS, or other Kubernetes clusters.

* create a docker image = docker build -t node-task-1:v1.1.0 .
* create a docker container = docker run -td --name node-task -p 3015:3015 node-task-1:v1.1.0
* Command to connect our credentials of hub.docker = docker login
* This tags your local Docker image = docker tag flask-task-4:v1.1.0 saketh1809/flask-task-4:v1.1.0
* pushes the tagged image to your Docker Hub = docker push saketh1809/flask-task-4:v1.1.0

A diagram of a process

AI-generated content may be incorrect.

create new instance -> select 30 GB && t3.medium && Amazon linux && create new security group -> open [Jenkins on AWS](https://www.jenkins.io/doc/tutorials/tutorial-for-installing-jenkins-on-AWS/) -> go to downloading and installing Jenkins -> follow the commands

We need to install the list of components to create a pipeline

* 1. docker
  2. docker compose
  3. docker should be run in non root user
  4. aws CLI

IAM -> users -> select your user id -> security credentials -> access key -> delete the current access key -> create access key -> select CLI -> create -> download the .csv file

Jenkins home -> profile settings -> credentials -> global -> add credentials -> enter aws credentials created in previous step -> add credentials -> add docker.hub credentials

Jenkins server -> git clone <https://github.com/Msocial123/Capgemini-Sep-DevOps-Training.git> ->

Jenkins can be classified into two ways of manual and scripting

chmod 777 /var/run/docker.sock

#### Docker Compose

docker compose is a tool within the docker to deploy the whole application stack.

**Docker Compose** is a tool used for defining and running multi-container Docker applications. It allows you to configure your application's services, networks, and volumes in a single YAML file (docker-compose.yml), making it easier to manage and deploy complex environments. It will create a custom network and with in the network it works.

Bi diualiance,

**🔧 Key Features of Docker Compose:**

1. **Multi-container orchestration**: Run multiple containers as a single service.
2. **Declarative configuration**: Define everything in a YAML file.
3. **Simplified commands**: Use docker-compose up and docker-compose down to start and stop your entire app.
4. **Environment isolation**: Easily manage different environments (dev, test, prod).
5. **Networking**: Automatically creates a network for your services to communicate.

**🚀 Common Commands:**

* docker-compose up – Starts all services.
* docker-compose down – Stops and removes containers, networks, volumes.
* docker-compose build – Builds images defined in the Compose file.
* docker-compose logs – View logs from all services.

Steps to implement Docker-Compose:

* sudo curl -L "<https://github.com/docker/compose/releases/download/v2.24.6/docker-compose-$(uname> -s)-$(uname -m)" -o /usr/local/bin/docker-compose
* sudo chmod +x /usr/local/bin/docker-compose
* docker-compose –version
* docker-compose up -d

**📜 2. Difference Between Docker Compose YAML and Ansible YAML**

Both use YAML syntax, but they serve **very different purposes**:

| **Feature** | **Docker Compose YAML** | **Ansible Playbook YAML** |
| --- | --- | --- |
| **Purpose** | Define and run multi-container apps | Automate infrastructure and configuration |
| **Execution Tool** | docker-compose | ansible-playbook |
| **Focus** | Container orchestration | Server provisioning, config management |
| **Structure** | Services, networks, volumes | Hosts, tasks, roles, handlers |
| **Example Use Case** | Start a web app with DB and cache | Install Nginx, configure firewall |

version: '3.8'

services:

frontend:

build: ./frontend

ports:

- "3000:3000"

depends\_on:

- backend

environment:

- REACT\_APP\_BACKEND\_URL=http://backend:5000

networks:

- app-network

backend:

build: ./backend

ports:

- "5000:5000"

networks:

- app-network

networks:

app-network:

driver: bridge

**🧱 What is a Container?**

A **container** is a lightweight, standalone, and executable package that includes everything needed to run a piece of software:

* Code
* Runtime
* System tools
* Libraries
* Settings

Containers are built using technologies like **Docker**, and they allow applications to run reliably across different computing environments.

**🔑 Key Features:**

* **Isolation**: Each container runs independently.
* **Portability**: Runs the same on any system with a container runtime.
* **Efficiency**: Uses fewer resources than virtual machines.
* **Fast startup**: Containers start quickly compared to VMs.

### ⚠️ Problems with Containers (Without Orchestration)

While containers are powerful, managing them at scale introduces challenges:

| **Problem** | **Description** |
| --- | --- |
| **Scaling** | Manually starting/stopping containers for load balancing is inefficient. |
| **Service Discovery** | Hard to find and connect services dynamically. |
| **Health Monitoring** | No built-in way to restart failed containers automatically. |
| **Networking** | Managing internal communication between containers can be complex. |
| **Storage & Volumes** | Persistent storage across container restarts is tricky. |
| **Security & Isolation** | Containers share the host kernel, which can be a risk. |
| **Configuration Management** | Managing environment variables, secrets, and configs across containers is hard. |
| **Rolling Updates** | Updating containers without downtime is manual and error-prone. |

**⚙️ What is Orchestration?**

**Container orchestration** is the automated management of containerized applications. It handles:

* Deployment
* Scaling
* Networking
* Monitoring
* Load balancing
* Rollbacks and updates

**🔧 Popular Orchestration Tools:**

* **Kubernetes** (most widely used)
* **Docker Swarm**
* **Apache Mesos**
* **Nomad**

**✅ Why Do We Choose Orchestration?**

| **Benefit** | **Description** |
| --- | --- |
| **Automated Scaling** | Automatically adds/removes containers based on load. |
| **Self-Healing** | Restarts failed containers, replaces unhealthy ones. |
| **Load Balancing** | Distributes traffic across containers. |
| **Rolling Updates** | Updates services with zero downtime. |
| **Service Discovery** | Automatically finds and connects services. |
| **Resource Management** | Efficient use of CPU, memory, and storage. |
| **Security** | Manages secrets, access control, and isolation. |
| **Declarative Configuration** | Define desired state using YAML/JSON files. |

**🧠 Summary**

* **Containers** are great for packaging and running apps.
* **Problems arise** when you have many containers to manage manually.
* **Orchestration tools** like Kubernetes solve these problems by automating deployment, scaling, and management.

1. Containers don’t support the autoscaling
2. Containers don’t support load balancing
3. Containers don’t support self-healing
4. Containers don’t support centralized network. communication between two containers is difficult in two different networks.
5. It doesn’t suit for the large scale applications.

To overcome these all challenges we use in built solution called docker swarm

Docker swarn is the orchestration tool.

Challenges of docker swarn.

* Complexity is high, configuration is high
* Doesn’t have saleability capabilities
* It doesn’t suit for large applications
* Networking:- Docker swan doesn’t have the capability to handle multiple networks
* Managing access permissions are very difficult.
* Very difficult to upgrade from one version to another version
* Managing resource to each and every container like allocation of memory CPU handling discussion is very difficult in docker swarn
* to overcome these challenges we have managing containers like Kubernetes and red hat open shift and noman

https://github.com/Msocial123/robot-shop-docker.git

**📊 Microservices Summary Table (**

| **Service** | **Purpose** | **Build Context** | **Image** | **Depends On** | **Ports** | **Env Vars** | **Healthcheck URL** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| mongodb | NoSQL DB for user/catalogue | mongo | ${REPO}/rs-mongodb:${TAG} | None | — | — | — |
| Redis | In-memory cache/store | — | redis:6.2-alpine | None | — | — | — |
| rabbitmq | Message broker | — | rabbitmq:3.8-management-alpine | None | — | — | — |
| catalogue | Product catalogue | catalogue | ${REPO}/rs-catalogue:${TAG} | mongodb | — | — | /health |
| User | User management/auth | user | ${REPO}/rs-user:${TAG} | mongodb, redis | — | — | /health |
| Cart | Shopping cart | cart | ${REPO}/rs-cart:${TAG} | redis | — | — | /health |
| mysql | Relational DB | mysql | ${REPO}/rs-mysql-db:${TAG} | None | — | — | — |
| shipping | Shipping logistics | shipping | ${REPO}/rs-shipping:${TAG} | mysql | — | — | /health |
| ratings | Product ratings | ratings | ${REPO}/rs-ratings:${TAG} | mysql | — | APP\_ENV=prod | /\_health |
| payment | Payment processing | payment | ${REPO}/rs-payment:${TAG} | rabbitmq | — | PAYMENT\_GATEWAY | /health |
| dispatch | Order dispatch | dispatch | ${REPO}/rs-dispatch:${TAG} | rabbitmq | — | — | — |
| Web | Frontend UI | web | ${REPO}/rs-web:${TAG} | catalogue, user, shipping, payment | 8080 | INSTANA\_AGENT\_KEY | / |

# **Kubernetes:**

## 🚀 What is Kubernetes?

Kubernetes helps you manage **containers** (like Docker containers) across a **cluster of machines**. It abstracts away the complexity of managing containers manually and provides a robust framework for running distributed systems reliably.

* It was introduced by google in 2014. It basically came from borg. Borg is a container orchestration tool.
* Google donated Kubernetes to CNCF ( cloud native cloud formation)
* Kubernetes has self-healing nature. If any of the container goes down it is going to become the container.
  + Kubernetes continuously monitors the health of containers.
  + If a container crashes or becomes unresponsive:
    - It automatically restarts it.
    - If a node fails, it reschedules the pod on another healthy node.
* Kubernetes support autoscaling.
  + This helps handle traffic spikes and optimize resource usage.
* Kubernetes support load balancing
  + Kubernetes has built-in **service load balancing**:
    - Distributes traffic evenly across healthy pods.
    - Uses internal DNS to route requests to the right service.
  + External traffic can be managed using **Ingress Controllers** (e.g., NGINX, Traefik).
* It is a platform independent. Write once run anywhere.
* To perform the automation in Kubernetes we have to write a Kubernetes manifestation file
  + Kubernetes uses **declarative configuration:**
    - You define the desired state of your app in YAML files (called manifests).
    - Kubernetes ensures the actual state matches the desired state.
* It is promising 0 downtime of the application
  + Kubernetes supports **rolling updates**:
    - Gradually replaces old pods with new ones.
    - Ensures traffic is only sent to healthy pods.
  + Also supports **rollbacks** if something goes wrong.
  + This enables **continuous delivery** without affecting users.
* Kubernetes is managing the application in cluster (group of nodes ).
* Node is a server where actually our containers run
* Kubernetes doesn’t understand container. It will understand pods. Pods is a smallest deployed unit where we run the application

 **Node**: A physical or virtual machine in the cluster.

 **Pod**: The smallest deployable unit in Kubernetes.

* Can contain one or more containers.
* Containers in a pod share network and storage.

 Kubernetes schedules **pods**, not individual containers.

### 🧠 Key Concepts in Kubernetes

| **Concept** | **Description** |
| --- | --- |
| **Pod** | The smallest unit in Kubernetes. A pod can contain one or more containers that share storage and network. |
| **Node** | A machine (VM or physical) where Kubernetes runs containers. |
| **Cluster** | A group of nodes managed by Kubernetes. |
| **Deployment** | Defines how to deploy and manage pods (e.g., replicas, updates). |
| **Service** | Exposes pods to the network and enables communication between components. |
| **Ingress** | Manages external access to services, typically HTTP/HTTPS. |
| **ConfigMap & Secret** | Used to manage configuration and sensitive data separately from code. |
| **Volume** | Persistent storage for containers. |
| **Namespace** | Logical partitioning of resources within a cluster. |

**🛠️ What Kubernetes Does**

* **Automated Deployment**: Launch containers across multiple nodes.
* **Scaling**: Automatically increase or decrease the number of containers based on demand.
* **Self-Healing**: Restarts failed containers, replaces them, and reschedules them.
* **Load Balancing**: Distributes traffic across containers.
* **Rolling Updates & Rollbacks**: Updates applications without downtime.
* **Service Discovery**: Automatically finds and connects services.

**📈 Why Use Kubernetes?**

| **Benefit** | **Description** |
| --- | --- |
| **High Availability** | Ensures your app is always running, even if some containers fail. |
| **Scalability** | Easily scale up/down based on traffic. |
| **Portability** | Works across cloud providers and on-premises. |
| **Efficiency** | Optimizes resource usage across nodes. |
| **Declarative Configuration** | Define desired state using YAML files. |
| **Security** | Manages secrets, RBAC, and network policies. |

### 🔄 Kubernetes vs Docker

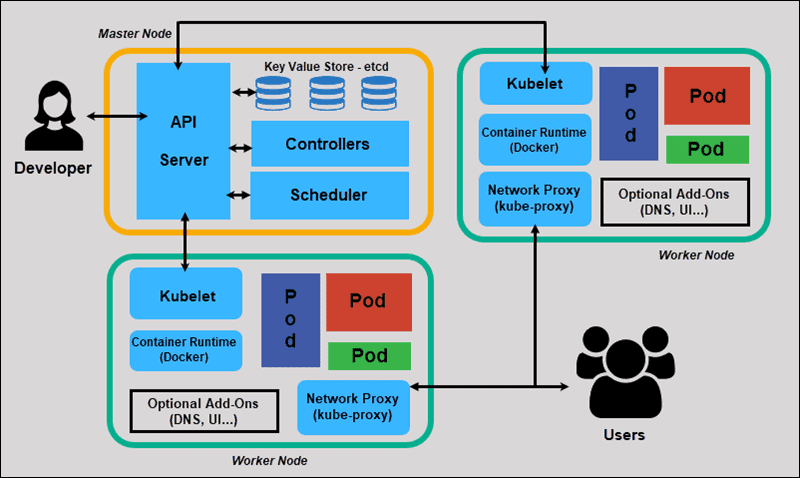
| **Feature** | **Docker (Standalone)** | **Kubernetes** |
| --- | --- | --- |
| **Container Runtime** | Yes | Yes (uses Docker or containerd) |
| **Orchestration** | No | Yes |
| **Scaling** | Manual | Automatic |
| **Self-Healing** | No | Yes |
| **Load Balancing** | Basic | Advanced |
| **Service Discovery** | Limited | Built-in |

## Kubernetes architecture:

In Kubernetes architecture, we have a master node and a worker node

Master node is a control plain which is going to manage whole cluster means scheduling jobs, health of the node, health of the pods, health of the cluster, autoscaling and load balancing.

Worker node is nothing but the servers where actually we are deploying our microservice application.



In master node we have 4 main components:

* API server
* Schedular
* Control manager
* ETCD
* Kubectl: it is a command line tool for Kubernetes, if you want to interact with the Kubernetes cluster we are going to use the kubectl. Without kubectl we can not communicate.
* API server: it is a hero of the cluster, when ever you want to perform autoscaling and load balancing to do the api calling. Plays crucial role in Kubernetes
* Kubernetes performs automatic deployment of the
* Scheduler has it’s own algorithm based on that it is going to schedule a pod on a particular node.
* Before it is scheduled a pod on the node, it is going to check if the resource are available like CPU, memory based on that it is going to schedule a pod on the node.
* ETCD: it is a distributed database. Whatever the information you want to store like if you want to deploy a application in a cluster or your application is saved in ETCD in key value pairs.

In a Kubernetes cluster we need at least one master node and one worker node.

Worker node components:

* Kubectl is an agent which is responsible for creating PODs and if something goes wrong, if pod goes down or node goes down it will communicate with the master node.
* Container runtime is responsible for pulling the docker runtime, creating the containers automatically and managing the lifecycle
  + By default in Kubernetes managing containerD as a container run time tool. From version 1.31
* Kube proxy- it is responsible for creating deployments, exposing applications to the internet. It is one of the most crucial component in cluster responsible for assigning the IP address, port numbers to establish communication from POD to POD, NODE to NODE and creating connection between services.
* Without those component we can not do anything to the server. It is managing overall networking in the cluster.
* POD is nothing but a small instance where exactly our containers run.
* Pod manages the container, container manages the docker image.
* Every microservice will run in a single service. A pod can have multiple containers unless there is no tight couple nature.
* Every POD has IP address
* Every container has port number
* Every POD can have CPU, memory units which is going to be assigned by the memory units.
* CONTROL MANAGER : It is responsible for the overall health of the cluster. If any pod goes down, it is going to recreate a new pod. If any node goes down it is going to recreate a new node. It has different components like node controller, job controller, end point slice controller and service account controller.

cloud controller manager:

## 🚀 What is Amazon EKS?

Amazon EKS lets you **run Kubernetes clusters** without having to install and operate your own Kubernetes control plane or nodes manually. It integrates deeply with other AWS services, making it easier to build secure, scalable, and highly available containerized applications.

**🧠 Key Features of Amazon EKS**

| **Feature** | **Description** |
| --- | --- |
| **Managed Control Plane** | AWS runs and scales the Kubernetes master nodes for you. |
| **Integration with AWS Services** | Works with IAM, VPC, CloudWatch, ALB, EBS, etc. |
| **Security** | Uses IAM for authentication and integrates with AWS Secrets Manager. |
| **Scalability** | Supports auto-scaling of worker nodes and pods. |
| **High Availability** | Control plane is spread across multiple Availability Zones. |
| **Platform Independence** | You can run EKS on AWS, on-premises (via EKS Anywhere), or hybrid environments. |
| **Support for Fargate** | Run containers without managing EC2 instances. |

**🛠️ How EKS Works**

1. **Control Plane**: Managed by AWS (you don’t have to worry about it).
2. **Worker Nodes**: You can use EC2 instances or AWS Fargate.
3. **Kubernetes API**: You interact with EKS using kubectl just like any other Kubernetes cluster.
4. **Networking**: Integrated with AWS VPC for secure communication.
5. **Storage**: Use EBS, EFS, or S3 for persistent storage.

To create a cluster with command line we have some requirements and need some set of tools like

* Kubectl
* Eksctl
* Awscli

If we are creating cluster using eksctl, it will use cloud formation template to create a cluster. It will create a unique VPC, subnets routing table elastic IP address and also it will create a master node along with that worker nodes but we have to define how many worker nodes are required and at configuration it will need and how many number of auto scaling

## Commands for executing the Kubernetes cluster on VM:

### 🖥️ Shell Script & Permissions

* vi cluster.sh  
  → Opens or creates a shell script file named cluster.sh for editing.
* https://github.com/Msocial123/EverNorth-DevOps-Training-Material.git
* chmod 777 cluster.sh  
  → Gives full read, write, and execute permissions to everyone for the script.
* sh cluster.sh  
  → Executes the shell script to run the commands inside it.

### ☁️ AWS & EKS Tools

* aws –version  
  → Displays the installed version of the AWS CLI tool.
* eksctl  
  → Runs the eksctl CLI tool used to create and manage EKS clusters.
* kubectl  
  → Runs the Kubernetes CLI tool to interact with your cluster.
* snap install kubectl –classic  
  → Installs kubectl via Snap package manager in classic mode (full system access).
* kubectl  
  → Verifies that kubectl is installed and ready to use.
* aws configure  
  → Sets up AWS CLI credentials and default region for your account.

### ☁️ Command for creating AWS EKS

eksctl create cluster --name saketh-test-cluster \

--version 1.32 \

--region ap-northeast-2 \

--nodegroup-name saketh-test-linux \

--node-type t3.medium \

--nodes 3 \

--nodes-min 3 \

--nodes-max 5 \

--managed

### 🔧 AWS & kubeconfig setup

* aws eks update-kubeconfig --name saketh-test-cluster-2 --region ap-northeast-2  
  → Updates your local kubeconfig file to connect to the specified EKS cluster.
* vi /root/.kube/config  
  → Opens the kubeconfig file in the vi editor to view or manually edit cluster connection settings.

### 🧱 Namespace Management

* kubectl create namespace test-ns  
  → Creates a new namespace called test-ns in the cluster.
* kubectl get ns  
  → Lists all namespaces currently available in the cluster.
* kubectl explain ns  
  → Shows documentation and structure for the Namespace resource.
* kubectl apply -f ns.yml  
  → Applies the configuration from ns.yml to create or update a namespace.

### 📦 Resource Inspection

* kubectl get all -n kube-system  
  → Lists all resources (pods, services, etc.) in the kube-system namespace.
* kubectl get pods -o wide  
  → Lists all pods with extended details like node name and IP.
* kubectl get pods --all-namespaces  
  → Lists all pods across all namespaces in the cluster.
* kubectl get pods -n your-namespace  
  → Lists all pods in a specific namespace (replace your-namespace with actual name).
* kubectl get pods -A  
  → Shortcut for listing all pods in all namespaces (-A = --all-namespaces).

### 🔍 Pod & Service Details

* kubectl describe pod <pod-name> -n <namespace>  
  → Shows detailed information about a specific pod in a given namespace.
* kubectl get service  
  → Lists all services in the current namespace.

eksctl delete cluster --name saketh-test-cluster --region ap-northeast-2

aws eks update-kubeconfig --name saketh-test-cluster-4 --region ap-northeast-2

kubectl get deployments

kubectl get rs

kubectl get pods

kubectl get svc

for applying the ingress controller : kubctl apply -f <https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.2.1/deploy/static/provider/cloud/deploy.yaml>

kubectl delete deployments --all

kubectl get pods

kubectl delete pods --all

kubectl delete ingress --all

kubectl delete svc --all

Metrics server that we are going to use to understand the utilization of resources.

VPC-cni : container network interface to establish pod to pod, pod to node, pod to container. There are different CNI software like Calico

Code dns – to maintain the naming clusters with in the clusters

## Kubernetes Namespace:

In Kubernetes, *namespaces* provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces.

a **Kubernetes namespace** is a way to allow **multiple users or teams to share the same Kubernetes cluster** without interfering with each other.

[Kubernetes API Reference Docs](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.32/#namespace-v1-core)

Added new context arn:aws:eks:ap-northeast-2:636768524979:cluster/murali-test-cluster to /root/.kube/config

🧭 1. **Imperative Approach**

You **manually issue commands** to make changes to the cluster.

**✅ Pros:**

* Quick and easy for small tasks
* Good for learning and experimentation

**❌ Cons:**

* Not repeatable or version-controlled
* Hard to track changes
* Not ideal for automation

📜 2. **Declarative Approach**

You **define the desired state** of your cluster in configuration files (usually YAML), and Kubernetes makes it happen.

**✅ Pros:**

* Version-controlled (can be stored in Git)
* Repeatable and consistent
* Ideal for automation and CI/CD
* Scales well for large teams and environments

**❌ Cons:**

* Requires learning YAML and structure
* Slightly more setup time
* To view namespace = kubectl create namespace saketh-test-ns
* kubectl get ns
* kubectl get all -n kube-system

open VScode -> enter the code in the file ns.yml -> make a copy in the git hub repository -> clone it in the Bastion-Host server

* kubectl apply -f ns.yml
* kubectl get nodes
* kubectl get pods -o wide
* kubectl get pod

**🧾 Kubernetes YAML File Structure**

A typical Kubernetes YAML file is made up of **key sections**:

**1. apiVersion**

* **What it means**: Specifies which version of the Kubernetes API you're using.
* **Example**: apiVersion: apps/v1
* **Why it's important**: Different resources (like Pods, Deployments, Services) use different API versions.

**2. kind**

* **What it means**: Defines the **type of Kubernetes object** you're creating.
* **Examples**:
  + Namespace → creates a namespace
  + Deployment → manages pods and replicas
  + Service → exposes your app to the network

**3. metadata**

* **What it means**: Contains **basic info** about the object.
* **Fields**:
  + name: Name of the object
  + namespace: (optional) Which namespace it belongs to
  + labels: (optional) Tags used for organizing and selecting resources

**4. spec**

* **What it means**: Describes the **desired state** of the object.
* This section varies depending on the kind.

POD limitations

1. Pod goes down means that particular service is not available for the customer
2. To recreate the pod automatically we need to attach controllers
3. If pod goes down ip address is also changed
4. If pod goes down, along with the pod we are going to lost the data as well. To over come this problem we must use volumes
5. If any pod goes down it is going to re create the same pod with the help of the template.
6. Template contains the pod definition
7. Replica set will maintain pod

Replicaset doesn’t support roll out and roll back

## Kubernetes Services:

1. clusterIP
2. node port
3. load balancer
4. environment variables

**🧠 What is a Kubernetes Service?**

In Kubernetes, a **Service** is an abstraction that defines **how to access a set of pods**. It provides a **stable network endpoint** (like a URL or IP) to reach your application, even if the underlying pods change or restart.

A **Service** in Kubernetes is like a **permanent, stable name and address** for accessing your application (pods), even if the pods themselves change or restart.

**🔐 Why Use a Service?**

* Pods have **dynamic IPs** — they change when restarted.
* A Service gives you a **fixed name (like a web URL)** and handles routing to the correct pod.
* It also helps you **control access** — whether internal (within the cluster) or external (from the internet).
* Pods are **ephemeral** — they can die and get recreated with new IPs.
* A Service gives you a **consistent way to access your app**, regardless of pod changes.

**🏢 Real-Life Analogy: Office Extension Numbers**

Imagine an office where employees (pods) keep changing desks (IP addresses). You don’t want to memorize their desk numbers every day. Instead, you call their **extension number** (Service), and the system routes your call to the right desk.

**🌐 So Yes — Service = Secure Web Name**

* Instead of using pod IPs (which change), you use a **Service name**.
* Kubernetes handles the routing and load balancing.
* You can **expose** your app securely using NodePort or LoadBalancer.

**🔧 Types of Kubernetes Services**

### 1. ClusterIP (Default)

* **What it does**: Makes the service accessible **only within the cluster**.
* **Use case**: Internal communication between microservices.
* **Example**: A frontend pod calling a backend pod.

it is one of the most secure way to protect way from the hackers. We are not exposing outside of cluster.

the services are accessible within the cluster if we have 10 microservices, if we deploy all 10 microservices. If we want to expose all these microservices to the internet we have to use ingress service.

### 2. NodePort

* **What it does**: Exposes the service on a **static port** on each node’s IP.
* **Use case**: Access the app from outside the cluster (e.g., browser).
* **Example**: http://<NodeIP>:<NodePort>

If you want to expose application on top of node using public ip address we have to use public IP address

This type we are not recommending to the customers because we are exposing the application through the public IP address and it may lead to hack attacks.

### 3. LoadBalancer

* **What it does**: Provisions an **external load balancer** (usually from a cloud provider).
* **Use case**: Production-grade external access.
* **Example**: http://<ExternalIP> (automatically assigned)

To expose the application through the internet one of the most common practice is that we use loadbalancer.

Kubernetes doesn’t provide any loadbalancer. The loadbalancer provided by the cloud provider like AWS, GCP, Azure, oracle, etc.

To distribute load we are creating application load balancers

When you create a load balancer it is creating outside the cluster. To distribute inside the cluster we are using node port, ClusterIP internally inside cluster.

### 4. Environment Variables

* Kubernetes automatically injects **service-related environment variables** into pods.
* Useful for apps that read connection info from environment variables.

**🌐 How the URL Changes**

| **Service Type** | **Access Method** | **Example** |
| --- | --- | --- |
| ClusterIP | Internal only | http://my-service:port (inside cluster) |
| NodePort | External via node IP | http://<NodeIP>:<NodePort> |
| LoadBalancer | External via cloud LB | http://<ExternalIP> |

**🧠 Summary**

| **Feature** | **Description** |
| --- | --- |
| **Service** | Stable way to access pods |
| **ClusterIP** | Internal access only |
| **NodePort** | External access via node IP and port |
| **LoadBalancer** | External access via cloud provider |
| **Env Variables** | Auto-injected for service discovery |

## Ingress Controllers

An **Ingress Controller** is a specialized software component in Kubernetes that manages incoming traffic to applications running in a Kubernetes environment. It acts as a bridge between external users and containerized services, allowing for HTTP and HTTPS routing based on user-defined rules.

* In order for an [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) to work in your cluster, there must be an *ingress controller* running. You need to select at least one ingress controller and make sure it is set up in your cluster.
* Unlike other types of controllers which run as part of the kube-controller-manager binary, Ingress controllers are not started automatically with a cluster.
* You may deploy any number of ingress controllers using [ingress class](https://kubernetes.io/docs/concepts/services-networking/ingress/#ingress-class) within a cluster. If you do not specify an IngressClass for an Ingress, and your cluster has exactly one IngressClass marked as default, then Kubernetes [applies](https://kubernetes.io/docs/concepts/services-networking/ingress/#default-ingress-class) the cluster's default IngressClass to the Ingress.

### Ingress

Make your HTTP (or HTTPS) network service available using a protocol-aware configuration mechanism, that understands web concepts like URIs, hostnames, paths, and more. The Ingress concept lets you map traffic to different backends based on rules you define via the Kubernetes API.

[Ingress](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.34/#ingress-v1-networking-k8s-io) exposes HTTP and HTTPS routes from outside the cluster to [services](https://kubernetes.io/docs/concepts/services-networking/service/) within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

A type of loadbalancer

Present outside the cluster

2 sections: ingress controller, resource

An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An [Ingress controller](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

A diagram of a diagram

AI-generated content may be incorrect.

Path based routing

Domain based routing

# **SonarQube:**

**SonarQube** is an open-source platform used for **continuous inspection of code quality**. It performs **automatic reviews** with static analysis of code to detect bugs, code smells, and security vulnerabilities in more than 25 programming languages.

**🔧 Key Components of SonarQube**

1. **SonarQube Server**
   * Central component that hosts the web interface and manages the database.
   * Provides dashboards, project views, and reports.
2. **Database**
   * Stores all the analysis results, configurations, and historical data.
   * Common choices: PostgreSQL, MySQL, Oracle.
3. **Scanner**
   * Tool that analyzes the source code and sends the results to the SonarQube server.
   * Examples: SonarScanner CLI, Maven plugin, Gradle plugin, Jenkins integration.
4. **Web Interface**
   * UI for developers and managers to view code quality metrics, issues, and trends.
   * Offers drill-down capabilities into specific issues.
5. **Plugins**
   * Extend functionality (e.g., support for new languages, integration with CI/CD tools, custom rules).

**🔄 SonarQube Workflow**

Here’s a typical workflow in a CI/CD pipeline:

1. **Code Commit**
   * Developer commits code to a version control system (e.g., Git).
2. **Build Trigger**
   * CI tool (e.g., Jenkins, GitHub Actions, Azure DevOps) triggers a build.
3. **Code Analysis**
   * SonarScanner runs during the build process.
   * It analyzes the code and sends results to the SonarQube server.
4. **Quality Gate Evaluation**
   * SonarQube applies a **Quality Gate** (a set of conditions like no new bugs, coverage thresholds).
   * If the gate fails, the build can be marked as failed.
5. **Results Visualization**
   * Developers view issues, metrics, and trends on the SonarQube dashboard.
   * They fix issues and push changes.
6. **Re-analysis**
   * The updated code is re-analyzed to ensure quality improvements.

**📊 Important Metrics Tracked**

* **Bugs**: Defects in the code.
* **Code Smells**: Maintainability issues.
* **Vulnerabilities**: Security risks.
* **Coverage**: Percentage of code covered by tests.
* **Duplications**: Repeated code blocks.
* **Technical Debt**: Estimated time to fix issues.

**🧪 Test Cases vs. SonarQube**

| **Aspect** | **Test Cases** | **SonarQube** |
| --- | --- | --- |
| **Purpose** | Validate functional correctness | Ensure code quality and maintainability |
| **Focus** | Runtime behavior | Static code analysis |
| **Detects** | Logical errors, incorrect outputs | Bugs, code smells, security vulnerabilities |
| **Execution** | Requires running the code | Analyzes code without execution |
| **Coverage** | Limited to what tests cover | Reviews all code paths, even untested ones |
| **Security** | May miss vulnerabilities | Identifies OWASP Top 10 issues |
| **Maintainability** | Not directly addressed | Highlights duplicated code, complexity, etc. |

**🔍 What SonarQube Adds Beyond Testing**

1. **Static Code Analysis**
   * Finds issues **before** code is run.
   * Detects things like unused variables, unreachable code, and bad practices.
2. **Security Vulnerabilities**
   * Identifies potential exploits (e.g., SQL injection, hardcoded secrets).
3. **Code Smells**
   * Highlights poor design choices that may not break functionality but hurt maintainability.
4. **Technical Debt Estimation**
   * Quantifies how much effort is needed to clean up the codebase.
5. **Enforces Coding Standards**
   * Ensures consistency across teams and projects.
6. **Quality Gates**
   * Automatically fail builds if code quality thresholds aren’t met.

**🧠 Example Scenario**

Imagine you write a function that passes all unit tests but:

* Has deeply nested logic (hard to read).
* Uses deprecated APIs.
* Contains duplicated code.
* Has poor naming conventions.

**Tests won’t catch these**, but SonarQube will.

# **Jenkins pipeline script for github->docker image->ECR->hub.docker->Kubernetes (groovy script)**

pipeline {

agent any

parameters {

string(name: 'IMAGE\_TAG', defaultValue: 'latest', description: 'Docker image tag')

string(name: 'CONTAINER\_NAME', defaultValue: 'cap-cart-container', description: 'Docker container name')

string(name: 'HOST\_PORT', defaultValue: '8080', description: 'Host port to map to container port 3000')

}

environment {

DOCKER\_IMAGE = "cap-cart-app:${params.IMAGE\_TAG}"

DOCKER\_HUB\_REPO = "saketh1809/cap-cart-app"

ECR\_REPO = "636768524979.dkr.ecr.ap-northeast-2.amazonaws.com/saketh-capgemini"

CONTAINER\_PORT = "3000"

}

stages {

stage('Clone Repository') {

steps {

git branch: 'master', url: 'https://github.com/saketh1809/Ever-Cart-App-Project.git'

}

}

stage('Build Docker Image') {

steps {

script {

sh "docker build -t ${DOCKER\_IMAGE} ."

}

}

}

stage('Login to Docker Hub & Push') {

steps {

withCredentials([usernamePassword(credentialsId: 'Docker\_Credentials', usernameVariable: 'DOCKER\_USER', passwordVariable: 'DOCKER\_PASS')]) {

script {

sh """

echo "$DOCKER\_PASS" | docker login -u "$DOCKER\_USER" --password-stdin

docker tag ${DOCKER\_IMAGE} ${DOCKER\_HUB\_REPO}:${params.IMAGE\_TAG}

docker push ${DOCKER\_HUB\_REPO}:${params.IMAGE\_TAG}

"""

}

}

}

}

stage('Login to AWS ECR & Push') {

steps {

withCredentials([[$class: 'AmazonWebServicesCredentialsBinding', credentialsId: 'AWS\_credentials']]) {

script {

sh """

aws ecr get-login-password --region ap-northeast-2 | docker login --username AWS --password-stdin ${ECR\_REPO}

docker tag ${DOCKER\_IMAGE} ${ECR\_REPO}:${params.IMAGE\_TAG}

docker push ${ECR\_REPO}:${params.IMAGE\_TAG}

"""

}

}

}

}

stage('Deploy Docker Container') {

steps {

script {

sh """

docker rm -f ${params.CONTAINER\_NAME} || true

docker run -d --name ${params.CONTAINER\_NAME} -p ${params.HOST\_PORT}:${CONTAINER\_PORT} ${DOCKER\_IMAGE}

"""

}

}

}

}

post {

success {

echo "Pipeline completed successfully!"

}

failure {

echo "Pipeline failed. Please check the logs."

}

}

}

# **Prometheus & Grafana :**

## Prometheus

an open-source systems monitoring and alerting toolkit originally built at [SoundCloud](https://soundcloud.com/)

Prometheus's main features are:

* a multi-dimensional [data model](https://prometheus.io/docs/concepts/data_model/) with time series data identified by metric name and key/value pairs
* PromQL, a [flexible query language](https://prometheus.io/docs/prometheus/latest/querying/basics/) to leverage this dimensionality
  + Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real time.
* no reliance on distributed storage; single server nodes are autonomous
* time series collection happens via a pull model over HTTP
* [pushing time series](https://prometheus.io/docs/instrumenting/pushing/) is supported via an intermediary gateway
* targets are discovered via service discovery or static configuration
* multiple modes of graphing and dashboarding support
* Prometheus is a **monitoring system** and **time-series database**.
* It **collects metrics** from various sources (like Kubernetes, Linux servers, applications).
* It stores these metrics with timestamps and labels.
* You can query these metrics using **PromQL** (Prometheus Query Language).

Helm is an open source package manager for Kubernetes. It provides the ability to provide, share, and use software built for Kubernetes.

Common actions for Helm:

* helm search: search for charts
* helm pull: download a chart to your local directory to view
* helm install: upload the chart to Kubernetes
* helm list: list releases of charts

help to collect the metrics

## Grafana

With Grafana Cloud, you can enable observability without the overhead of building and maintaining your own stack. This section covers the basics of observability concepts and terminology.

collect information about the server like traffic, requests, memory, etc

monitoring is all about the understanding resource utilization is happening like how much traffic is coming, memory utilization and also we are trying to setting up the alerts or thresholds.

Observability means it defines how your application is behaving. It is majorly focusing on logs, traces, metrics.

* Grafana is a **visualization and dashboard tool**.
* It connects to Prometheus (and other data sources) to **display metrics** in graphs, tables, and charts.
* It also supports **alerting** based on those metrics.

### Grafana dashboard

A Grafana dashboard is a set of one or more panels that provide an at-a-glance view of related information. These panels are the basic building block in Grafana dashboards, and they’re created using components that query and transform raw data from a data source into charts, graphs, and other visualizations.

A data source can be an SQL database, Grafana Loki, Grafana Mimir, or a JSON-based API. It can even be a basic CSV file. Queries allow you to reduce the entirety of your data to a specific dataset, providing a more manageable visualization.

### Grafana Alerting

Using Grafana Alerting, you create queries and expressions from multiple data sources, no matter where you store your data, giving you the flexibility to combine your data and alert on your metrics and logs in new and unique ways. You can then create, manage, and take action on your alerts from a single, consolidated view, and improve your team’s ability to identify and resolve issues quickly.

Grafana Alerting lets you define alert rules across multiple data sources and manage notifications with flexible routing.

Built on the Prometheus alerting model, it integrates with the Grafana stack to provide a scalable and effective alerting setup across a wide range of environments.

#### **Notification policies**

[Notification policies](https://grafana.com/docs/grafana-cloud/alerting-and-irm/alerting/fundamentals/notifications/notification-policies/) are an advanced option for handling alert notifications by distinct scopes, such as by team or service—ideal for managing large alerting systems.

| **State** | **Description** |
| --- | --- |
| **Normal** | The state of an alert when the condition (threshold) is not met. |
| **Pending** | The state of an alert that has breached the threshold but for less than the [pending period](https://grafana.com/docs/grafana/latest/alerting/fundamentals/alert-rule-evaluation/#pending-period). |
| **Alerting** | The state of an alert that has breached the threshold for longer than the [pending period](https://grafana.com/docs/grafana/latest/alerting/fundamentals/alert-rule-evaluation/#pending-period). |
| **Recovering** | The state of a firing alert when the threshold is no longer breached, but for less than the [keep firing for](https://grafana.com/docs/grafana/latest/alerting/fundamentals/alert-rule-evaluation/#keep-firing-for) period. |
| **Error\*** | The state of an alert when an error or timeout occurred evaluating the alert rule. You can customize the behavior of the [Error state](https://grafana.com/docs/grafana/latest/alerting/fundamentals/alert-rule-evaluation/nodata-and-error-states/#error-state), which by default triggers a different alert. |
| **No Data\*** | The state of an alert whose query returns no data or all values are null. You can customize the behavior of the [No Data state](https://grafana.com/docs/grafana/latest/alerting/fundamentals/alert-rule-evaluation/nodata-and-error-states/#no-data-state), which by default triggers a different alert. |

Notification policies routes alerts to contact points via label matching. They are defined in a tree structure, where the root of the notification policy tree is the **Default notification policy**, which ensures all alert instances are handled.

### Metrics

**Metrics are numerical values that represent the state or performance of a system over time. In the context of Kubernetes, Prometheus, and Grafana, metrics are used to monitor things like:**

* **CPU and memory usage**
* **Pod status**
* **Network traffic**
* **Disk I/O**
* **Application-specific data (e.g., HTTP request rates, error counts)**

**Structure of a Metric**

**Each metric typically includes:**

* **Name: e.g., kube\_pod\_status\_ready**
* **Labels (key-value pairs): e.g., namespace="default", pod="nginx-123"**
* **Timestamp: when the metric was collected**
* **Value: the actual measurement (e.g., 1 for ready, 0 for not ready)**

**Where Are These Metrics Stored?**

These metrics are collected by **Prometheus** via the **kube-state-metrics** component and stored in Prometheus's **time-series database**.

Prometheus scrapes these metrics at regular intervals from configured targets (like kube-state-metrics, node exporters, etc.).

**How Are They Used in Grafana Alerting?**

Grafana connects to Prometheus as a **data source** and uses its metrics to:

1. **Visualize** data in dashboards.
2. **Create alert rules** based on metric conditions.

**Alerting Flow:**

1. **Metric must exist in Prometheus.**
2. **Grafana queries Prometheus** for that metric.
3. **You define alert conditions** in Grafana (e.g., if a metric value > threshold).
4. **Grafana evaluates the rule** periodically.
5. If the condition is met, **Grafana triggers an alert** and sends notifications (email, Slack, webhook, etc.).

### Grafana SLO

Grafana SLO (Service Level Objective) provides a framework for measuring the quality of service you provide to users. Use SLOs to collect data on the reliability of your systems over time and as a result, help engineering teams reduce alert fatigue, focus on reliability, and provide better service to your customers.

By creating SLIs (Service Level Indicators) and SLOs, you define what an acceptable level of service is and how to react if you are not providing the expected level of service.

#### **Service Level Indicators (SLIs)**

A key performance metric, like availability. These are the metrics you measure over time that inform you about the health or performance of a service. It expresses actual results as a fraction. For example, 99.9% system availability, or 0.999. Grafana SLO can help create a high-quality SLI using our ratio query builder, and can also support any custom PromQL query for an SLI.

#### **Service Level Objectives (SLOs)**

* **Definition**: A **target** for service reliability or performance.
* **Used internally** by engineering teams to guide operations.
* **Helps define** what “good enough” means.

When defining your SLOs, it is important to remember that you are not aiming for 100%. The cost and complexity of availability gets higher the closer you get to 100%, so it’s important to factor in a margin of error to your target, known as the error budget.

Using the SLO framework, this company can be more specific about their availability goals. The SLO in this case would be that they want 99.97% of requests to validate a credit card to return without a 500 error in less than 100ms. Validation should be instantaneous, because e-commerce websites need to show customers if they mistyped a number before a customer hits the “buy” button. The SLI in this case would be the % of requests to validate a credit card return without errors in less than 100ms.

## Steps for implementing grafana and Prometheus

Snap install helm –classic

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo add grafana https://grafana.github.io/helm-charts

helm repo update

helm install prometheus prometheus-community/kube-prometheus-stack \

--namespace monitoring --create-namespace

kubectl get pods -n monitoring

kubectl patch svc prometheus-kube-prometheus-prometheus -n monitoring \

-p '{"spec": {"type": "LoadBalancer"}}'

kubectl get svc -n monitoring

helm install grafana grafana/grafana --namespace monitoring

kubectl get pods -n monitoring

kubectl patch svc grafana -n monitoring \

-p '{"spec": {"type": "LoadBalancer"}}'

**To Get the Password :**

kubectl get secret --namespace monitoring grafana -o jsonpath="{.data.admin-password}" | base64 --decode

kubectl get svc -n monitoring

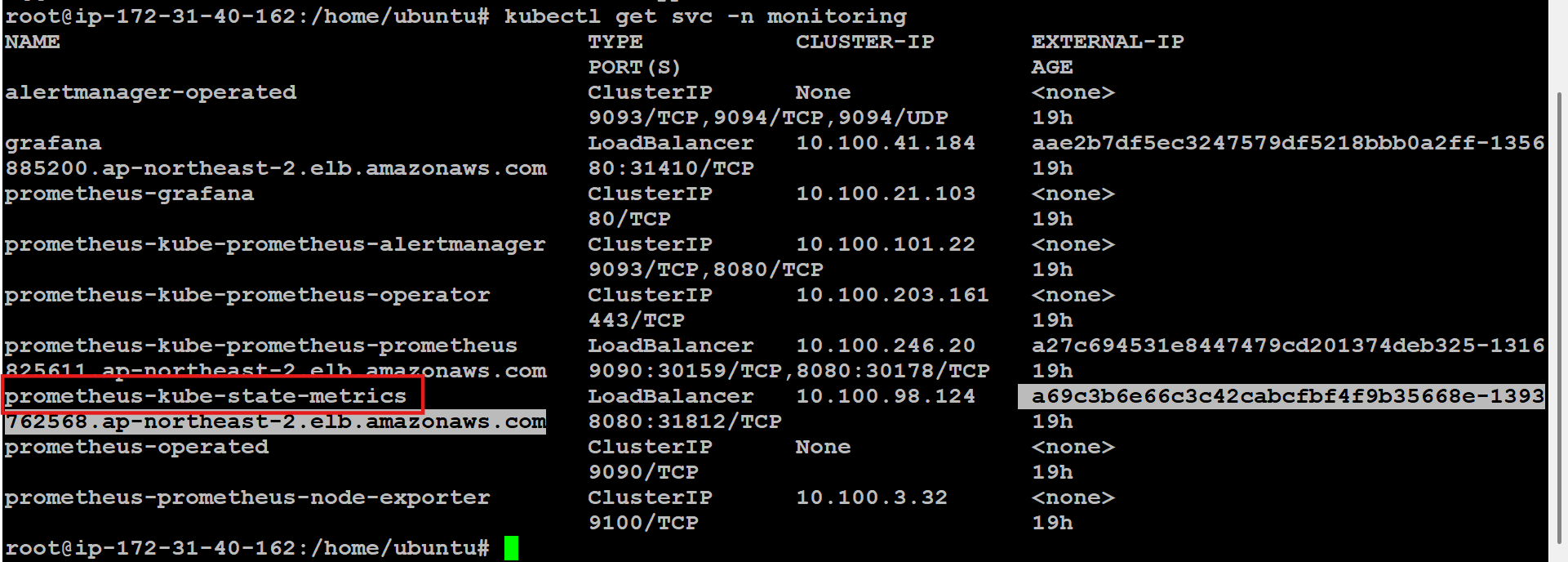
Grafana dashboard -> connections -> create new connections -> add new data source -> add the Prometheus dashboard link -> click save & test

Copy the Grafana dashboard ID from [Grafana dashboards | Grafana Labs](https://grafana.com/grafana/dashboards/)

Grafana dashboard -> Dashboard -> new -> import -> paste the DashboardID -> load

**Command to expose the metrics to the load balancer:**

kubectl patch service prometheus-kube-state-metrics -n monitoring -p '{"spec": {"type":"LoadBalancer"}}'



# **🛠️ AWS for DevOps**

AWS provides a suite of services that help automate software development, testing, deployment, and infrastructure management. These services support **CI/CD (Continuous Integration and Continuous Delivery)**, making it easier to build, test, and release software quickly and reliably.

The AWS Developer Tools help you securely store and version your application's source code and automatically build, test, and deploy your application to AWS or your on-premises environment.Start with AWS CodePipeline to build a continuous integration or continuous delivery workflow that uses AWS CodeBuild, AWS CodeDeploy, and other tools, or use each service separately.

**Automation**

AWS helps you use automation so you can build faster and more efficiently. Using AWS services, you can automate manual tasks or processes such as deployments, development & test workflows, container management, and configuration management.

**Secure**

Use AWS Identity and Access Management (IAM) to set user permissions and policies. This gives you granular control over who can access your resources and how they access those resources.

**Large Partner Ecosystem**

AWS supports a large ecosystem of partners which integrate with and extend AWS services. Use your preferred third-party and open source tools with AWS to build an end-to-end solution.

**Pay-As-You-Go**

With AWS purchase services as you need them and only for the period when you plan to use them. AWS pricing has no upfront fees, termination penalties, or long term contracts.

CodeCommit → CodePipeline → CodeBuild (build & push Docker image to ECR) → CodeDeploy (or ECS/EKS)

## 🔧 Key AWS DevOps Services

### 1. AWS CodeCommit

AWS CodeCommit is a fully-managed [source contro](https://aws.amazon.com/devops/source-control/)l service that makes it easy for companies to host secure and highly scalable private Git repositories. You can use CodeCommit to securely store anything from source code to binaries, and it works seamlessly with your existing Git tools.

* **What it is**: A fully managed source control service that hosts Git repositories.
* **Similar to**: GitHub, GitLab, Bitbucket.
* **Features**:
  + Supports Git commands like git clone, git pull, git push.
  + Secure and scalable with AWS IAM for access control.
  + Integrated with other AWS services for CI/CD.
  + No need to manage your own Git server.

### 2. AWS CodePipeline

* **What it is**: A CI/CD service that automates the build, test, and deploy phases of your release process.
* **Features**:
  + Visual workflow for defining stages (Source → Build → Test → Deploy).
  + Integrates with CodeCommit, GitHub, CodeBuild, CodeDeploy, etc.
  + Automatically triggers pipelines on code changes.
  + Supports manual approvals and custom actions.

### 3. AWS CodeBuild

AWS CodeBuild is a fully managed build service that compiles source code, runs tests, and produces software packages that are ready to deploy. With CodeBuild, you don’t need to provision, manage, and scale your own build servers. CodeBuild scales continuously and processes multiple builds concurrently, so your builds are not left waiting in a queue.

* **What it is**: A fully managed build service that compiles source code, runs tests, and produces deployable artifacts.
* We use buildspec.yaml file
* **Features**:
  + Scales automatically based on build volume.
  + Supports multiple languages (Java, Python, Node.js, etc.).
  + Pay-as-you-go pricing.
  + Integrated with CodePipeline for automated builds.

### 4. AWS CodeDeploy

AWS CodeDeploy automates code deployments to any instance, including Amazon EC2 instances and on-premises servers. AWS CodeDeploy makes it easier for you to rapidly release new features, helps you avoid downtime during application deployment, and handles the complexity of updating your applications.

* **What it is**: A deployment service that automates application deployments to various compute services like EC2, Lambda, or on-prem servers.
* **Features**:
  + Supports blue/green and rolling deployments.
  + Tracks deployment status and rollback on failure.
  + Works with EC2, ECS, Lambda, and even on-premises servers.
  + Integrated with CodePipeline for end-to-end automation.

**🔄 How They Work Together**

Here’s a typical flow:

1. **Developer pushes code** to CodeCommit.
2. **CodePipeline detects the change** and triggers the pipeline.
3. **CodeBuild compiles and tests** the code.
4. **CodeDeploy deploys** the application to the target environment.

**✅ Does AWS CodeBuild Build Docker Images?**

**Yes**, absolutely!

* **CodeBuild** can build Docker images as part of your build process.
* You can define this in your buildspec.yml file using Docker commands like:

YAML

phases:

build:

commands:

- docker build -t my-app .

- docker tag my-app:latest <your-aws-account-id>.dkr.ecr.<region>.amazonaws.com/my-app:latest

- docker push <your-ecr-uri>

Show more lines

* You need to enable **privileged mode** in CodeBuild to use Docker inside the build container.

**🐳 Do We Use AWS ECR in the Pipeline?**

**Yes**, typically in containerized workflows:

* **Amazon ECR (Elastic Container Registry)** is used to store Docker images.
* After CodeBuild builds and pushes the image to ECR, you can use it in:
  + ECS (Elastic Container Service)
  + EKS (Kubernetes)
  + Lambda (for container-based functions)
  + Or even EC2 instances

**🔄 Does CodePipeline Control CodeBuild and CodeDeploy?**

**Yes**, CodePipeline orchestrates the entire CI/CD flow.

Here’s how it works:

1. **Source Stage**:
   * CodePipeline pulls code from **CodeCommit**, GitHub, or Bitbucket.
2. **Build Stage**:
   * It triggers **CodeBuild** to compile, test, and optionally build Docker images.
3. **Deploy Stage**:
   * It hands over the artifacts to **CodeDeploy** (or ECS, Lambda, etc.) for deployment.

# **Google Cloud Platform**

**GCP** stands for **Google Cloud Platform**. It's a suite of cloud computing services offered by Google that runs on the same infrastructure that Google uses internally for its end-user products like Google Search, Gmail, YouTube, and more.

**Key Features of GCP:**

Here are some of the core services and capabilities it provides:

1. **Compute Services**:
   * **Compute Engine**: Virtual machines (VMs) for running workloads.
   * **App Engine**: Platform-as-a-Service (PaaS) for building scalable web apps.
   * **Cloud Functions**: Serverless functions triggered by events.
2. **Storage & Databases**:
   * **Cloud Storage**: Object storage for any amount of data.
   * **Cloud SQL**: Managed relational databases (MySQL, PostgreSQL).
   * **Bigtable**: NoSQL database for large-scale workloads.
   * **Firestore / Datastore**: NoSQL document databases.
3. **Networking**:
   * **VPC (Virtual Private Cloud)**: Customizable private network.
   * **Cloud Load Balancing**: Distributes traffic across resources.
   * **Cloud CDN**: Content delivery network for faster content delivery.
4. **Big Data & Machine Learning**:
   * **BigQuery**: Serverless data warehouse for analytics.
   * **Dataflow**: Stream and batch data processing.
   * **Vertex AI**: End-to-end ML platform for building and deploying models.
5. **Security & Identity**:
   * **IAM (Identity and Access Management)**: Controls who can do what.
   * **Cloud KMS**: Key management service for encryption keys.
   * **Security Command Center**: Centralized security management.
6. **Developer Tools**:
   * **Cloud SDK**: Command-line tools for interacting with GCP.
   * **Cloud Build**: CI/CD pipelines.
   * **Cloud Source Repositories**: Git repositories hosted on GCP.

**Use Cases:**

* Hosting websites and applications
* Data analytics and warehousing
* Machine learning model training and deployment
* IoT data processing
* Scalable backend infrastructure for mobile apps

## Cloud Source Repositories (CSR)

**Cloud Source Repositories** (CSR) is a **fully-featured, private Git repository service** provided by **Google Cloud Platform (GCP)**. It’s similar to GitHub or Bitbucket but tightly integrated with other GCP services.

Cloud Source Repositories provides fully featured, private [Git](https://git-scm.com/) repositories hosted on [Google Cloud](https://cloud.google.com/).

You can use Cloud Source Repositories for collaborative, version-controlled development of any app or service, including those that run on [App Engine](https://cloud.google.com/appengine/docs) and [Compute Engine](https://cloud.google.com/compute/docs).

If you're familiar with Git, you can get started quickly with Cloud Source Repositories. For example, you can add Cloud Source Repositories to a local Git repository as a remote, or you can connect it to a hosted repository on GitHub or Bitbucket. From a local repository, you can use the standard set of Git commands to interact with the repository in the cloud, including push, pull, clone, and log.

**🔍 What Cloud Source Repositories Does**

* Hosts **Git repositories** for your source code.
* Supports **version control**, **branching**, and **collaboration**.
* Integrates with **Cloud Build**, **Cloud Deploy**, and **Cloud Run** for CI/CD workflows.
* Can mirror repositories from **GitHub** or **Bitbucket**.

**🧠 Key Features**

| **Feature** | **Description** |
| --- | --- |
| **Git-based** | Fully compatible with Git commands and workflows |
| **Private Repos** | Secure, access-controlled repositories |
| **Integration with GCP** | Triggers Cloud Build on commits, links to Cloud Deploy |
| **Mirroring** | Can mirror external GitHub/Bitbucket repos for centralized control |
| **IAM Permissions** | Fine-grained access control using GCP IAM |
| **Code Search** | Powerful search across multiple repos (like grep for Git) |

Cloud Source Repositories automatically send logs on repository activity to Cloud Logging to help track and troubleshoot data access.

You can use these logs to review recent repository synchronization, repository access by other users, and administrative actions such as creations, deletions, and permission changes. Moreover, you can configure notification settings such that an alert is sent to you when an error is logged during a repository synchronization.

**🔁 Role in the GCP CI/CD Pipeline**

Here’s how Cloud Source Repositories fits into the pipeline:

1. **Developer pushes code** to CSR.
2. **Cloud Build Trigger** detects the change and starts a build.
3. **Cloud Build** builds the app and pushes the image to Artifact Registry.
4. **Cloud Deploy** picks up the image and deploys it to Cloud Run or GKE.
5. **Cloud Run** runs the deployed container.

**📄 Example Use Case**

You can set up a **Cloud Build trigger** like this:

* Trigger type: **Source Repository**
* Event: **Push to branch**
* Action: Run cloudbuild.yaml to build and deploy

**✅ Summary**

**Cloud Source Repositories** is GCP’s native Git hosting service, ideal for teams using GCP for CI/CD. It helps you manage code securely and integrate seamlessly with build and deployment tools.

## 🔧 Cloud Build – *Build & CI Tool*

**Cloud Build** is a **fully managed Continuous Integration (CI) service** in **Google Cloud Platform (GCP)** that automates the process of building, testing, and packaging your application.

You can configure Cloud Build to automatically build a new image any time a user pushes a change to files stored in Cloud Source Repositories. Events that initiate automatic builds are called [*build triggers*](https://cloud.google.com/build/docs/triggers). These triggers can help ensure that your container images are kept up to date. You can also use them to build and test feature branches.

Build triggers can perform a build based on either a Dockerfile or a build config file.

**🚀 What Cloud Build Does**

Cloud Build takes your **source code**, runs a series of **build steps**, and produces **artifacts** like Docker images, binaries, or packages. It’s often the **first stage** in a CI/CD pipeline.

**🧱 Key Capabilities**

1. **Build Docker Images**
   * Compile your app and package it into a Docker image.
2. **Run Tests**
   * Execute unit or integration tests as part of the build process.
3. **Push Artifacts**
   * Push Docker images to **Artifact Registry** or other repositories.
4. **Trigger Builds Automatically**
   * Use **Cloud Build Triggers** to start builds on Git commits, pull requests, or tag creation.
5. **Custom Build Steps**
   * Define steps using Docker containers in a cloudbuild.yaml file.

## Cloud Deploy

**Cloud Deploy** is a **fully managed Continuous Delivery (CD) service** in **Google Cloud Platform (GCP)**. It helps you **automate, manage, and track deployments** of your applications across environments like **development**, **staging**, and **production**.

**🚚 What Cloud Deploy Does**

Cloud Deploy focuses on the **delivery** part of CI/CD:

* **Defines deployment pipelines** using YAML.
* **Automates deployments** to services like:
  + **Cloud Run**
  + **Google Kubernetes Engine (GKE)**
  + **Anthos**
* **Supports approvals**, **rollbacks**, and **release tracking**.
* Integrates with **Cloud Build** for CI and **Artifact Registry** for image storage.

**🔁 How It Fits in the GCP Pipeline**

Here’s the full flow:

1. **Cloud Build**: Builds your app and pushes the Docker image to Artifact Registry.
2. **Artifact Registry**: Stores the image.
3. **Cloud Deploy**: Picks up the image and deploys it to Cloud Run or GKE.
4. **Cloud Run/GKE**: Runs the application.

**✅ Key Features of Cloud Deploy**

| **Feature** | **Description** |
| --- | --- |
| **Declarative YAML** | Define pipelines and targets in code |
| **Approvals** | Manual approval before production deployment |
| **Rollbacks** | Revert to previous versions easily |
| **Release Tracking** | View history of deployments |
| **Multi-Environment** | Deploy across dev, staging, prod |
| **Integration** | Works with Cloud Build, Artifact Registry, Cloud Run, GKE |

A screenshot of a computer

AI-generated content may be incorrect.

## 🚀 Cloud Run – Deployment & Runtime Platform

**Cloud Run** is a fully managed compute platform by **Google Cloud** that lets you run **containerized applications** in a **serverless** environment.

**🚀 Key Features of Cloud Run**

1. **Serverless**
   * No need to manage infrastructure. Google handles provisioning, scaling, and maintenance.
2. **Container-Based**
   * You deploy your app as a **Docker container**. Any language or framework is supported as long as it can run in a container.
3. **Automatic Scaling**
   * Scales up to handle traffic spikes and scales down to zero when idle—saving costs.
4. **Fast Deployments**
   * You can deploy directly from a container image stored in **Artifact Registry** or **Container Registry**.
5. **HTTP-Driven**
   * Cloud Run services are exposed via HTTPS endpoints. Ideal for web APIs, microservices, and event-driven apps.
6. **Stateless & Ephemeral**
   * Each request is handled independently. No persistent state between requests.
7. **Environment Variables**
   * You can configure runtime behavior using environment variables (e.g., CITY=London).
8. **Traffic Management**
   * Supports gradual rollouts, traffic splitting between revisions, and easy rollbacks.
9. **Security**
   * IAM-based access control, VPC connectors, and support for private services.

**🧾 Example Use Case**

You build a REST API in Python, package it in a Docker container, push it to Artifact Registry, and deploy it to Cloud Run. It scales automatically, is accessible via HTTPS, and you can manage its configuration using YAML or Terraform.

## Artifact Registry in GCP

**Artifact Registry** is GCP’s unified storage solution for managing:

* Docker container images
* Language-specific packages (e.g., Maven, npm, Python)
* OS packages (e.g., Debian, RPM)

It replaces **Container Registry** and provides better integration, security, and regional support.

### 🆚 Artifact Registry vs Amazon ECR

| **Feature** | **Google Cloud Artifact Registry** | **Amazon Elastic Container Registry (ECR)** |
| --- | --- | --- |
| **Purpose** | Stores container images and other build artifacts (e.g., Maven, npm, Python packages) | Stores container images for use with AWS services |
| **Supported Formats** | Docker, OCI, Maven, npm, Python, Debian, RPM | Docker, OCI |
| **Integration** | Tight integration with GCP services like Cloud Build, Cloud Run, GKE, Cloud Deploy | Integrated with AWS services like ECS, EKS, CodeBuild, CodePipeline |
| **Access Control** | IAM-based permissions with fine-grained access | IAM-based permissions with repository policies |
| **Security Features** | Vulnerability scanning, private repositories, VPC-SC support | Image scanning, encryption at rest, private repositories |
| **Regional Support** | Multi-region and regional repositories | Regional repositories |
| **Repository Types** | Standard and remote repositories (proxying external registries) | Private and public repositories |
| **Pricing** | Based on storage and network egress | Based on storage and data transfer |
| **CLI Tools** | gcloud, Docker CLI, Artifact Registry API | aws CLI, Docker CLI, ECR API |
| **Caching/Proxying** | Can proxy external registries (e.g., Docker Hub) | No native proxying support |

**🔍 Summary**

* **Artifact Registry** is more **versatile** in terms of supported artifact types (not just containers).
* **Amazon ECR** is more **container-focused**, tightly integrated with AWS container orchestration tools like ECS and EKS.
* Both offer **secure, scalable, and managed** solutions for storing and retrieving container images.

## Example shell scripts for Cloud Build and Cloud Run:

a **shell script** used to build and push a Docker image to **Google Cloud Artifact Registry** using **Cloud Build**.

A computer screen shot of a program

AI-generated content may be incorrect.

PROJECT\_ID=yaml-deploy-to-cloud-run

gcloud builds submit \

  --tag us-central1-docker.pkg.dev/$PROJECT\_ID/my-serverless-app/my-image \

  --project $PROJECT\_ID

🔍 **What It Does**

PROJECT\_ID=yaml-deploy-to-cloud-run = This sets the environment variable PROJECT\_ID to the name of your Google Cloud project.

gcloud builds submit \ = This command tells GCP to start a build using **Cloud Build**. It packages the current directory (by default) and sends it to Cloud Build.

The next specifies the **Artifact Registry** location where the built Docker image will be stored

--project $PROJECT\_ID = This tells gcloud which GCP project to use for the build and storage.

**After running this script:**

* Cloud Build will build the Docker image from your source code.
* The image will be pushed to **Artifact Registry** under the specified path.
* You can then deploy this image to **Cloud Run**, **GKE**, or other GCP services.

a **shell script** that deploys a containerized application to **Google Cloud Run**.

A screen shot of a computer program

AI-generated content may be incorrect.

PROJECT\_ID=yaml-deploy-to-cloud-run

* Sets the GCP project ID to yaml-deploy-to-cloud-run.

gcloud run deploy app \

* Deploys a service named app to Cloud Run.

--image us-central1-docker.pkg.dev/$PROJECT\_ID/my-serverless-app/my-image \

* Specifies the container image to deploy, stored in **Artifact Registry** under the path:
* us-central1-docker.pkg.dev/yaml-deploy-to-cloud-run/my-serverless-app/my-image

--set-env-vars=CITY=London \

* Sets an environment variable CITY with the value London for the deployed service.

--platform managed \

* Deploys to the **fully managed** Cloud Run platform (not Anthos or GKE).

--allow-unauthenticated \

* Allows public (unauthenticated) access to the service.

--region us-central1 \

* Specifies the deployment region as us-central1.

--project=$PROJECT\_ID

* Uses the previously defined project ID for the deployment.

**After running this script:**

This script deploys a Docker image to **Cloud Run**, sets an environment variable, makes the service publicly accessible, and targets a specific region and project.

### 🔁 What Is a Revision in Cloud Run?

A **revision** is an **immutable snapshot** of your Cloud Run service at a specific point in time. Every time you:

* Deploy a **new container image**
* Change **configuration settings** (like memory, CPU, environment variables, etc.)

Cloud Run creates a **new revision**.

**GCP revisions** are **versioned deployments** — each one is stored separately and can be referenced independently.

**📦 What Does a Revision Include?**

Each revision contains:

1. **Container Image**
   * The exact image you deployed (e.g., from Artifact Registry).
2. **Environment Variables**
   * Key-value pairs like CITY=London that are injected into the container at runtime.
3. **Configuration Settings**
   * CPU and memory allocation
   * Concurrency settings
   * Timeout settings
   * Authentication settings (e.g., allow unauthenticated access)
4. **Metadata**
   * Revision ID
   * Deployment timestamp
   * Region and platform info

**🧬 Why Are Revisions Important?**

* **Immutability**: Once created, a revision cannot be changed. This ensures consistency and traceability.
* **Rollbacks**: You can easily revert to a previous revision if something goes wrong.
* **Traffic Splitting**: You can split traffic between multiple revisions (e.g., 90% to v1, 10% to v2) for canary deployments.

**🧠 Summary**

A **revision** in Cloud Run is like a versioned snapshot of your service deployment. It includes the container image, environment variables, and all configuration settings. Every change creates a new revision, enabling safe rollbacks and traffic management.

# **Google Cloud Platform Deployment Pipeline:**

**Developer**

**↓**

**Cloud Source Repo**

**↓**

**Cloud Build (CI)**

**↓**

**Artifact Registry (Image Storage)**

**↓**

**Cloud Deploy (CD using YAML)**

**↓**

**Cloud Run (App Hosting) -> GKE**

## 🧑‍💻 1. Cloud Source Repositories (CSR) – *Version Control*

* Acts like GitHub.
* Developers push code here.
* Supports Git operations and integrates with GCP services.

✅ **Role:** Source of truth for your application code.

## 🛠️ 1. Cloud Build – Build & CI Stage

* **Purpose**: Automates the building, testing, and packaging of your application.
* **What happens**:
  + Source code is pulled from a repository (e.g., GitHub).
  + Cloud Build uses a cloudbuild.yaml file to define steps:
    - Build the Docker image.
    - Run tests (optional).
    - Tag and push the image to Artifact Registry.

A screenshot of a computer

AI-generated content may be incorrect.

## 📦 2. Artifact Registry – Image Storage

* **Purpose**: Stores Docker images and other build artifacts.
* **Similar to AWS**: Comparable to **Amazon ECR (Elastic Container Registry)**.
* **What happens**:
  + The Docker image built by Cloud Build is pushed to Artifact Registry.
  + This image is versioned and securely stored for deployment.

## 🚀 3. Cloud Run – Deployment & Runtime

* **Purpose**: Runs containerized applications in a serverless environment.
* **What happens**:
  + You deploy the image from Artifact Registry to Cloud Run.
  + Cloud Run automatically scales the app based on traffic.
  + You can configure environment variables, memory, concurrency, etc.
  + Similar to **Tomcat** in the sense that it **hosts and runs your app**, but it's modern, container-based, and cloud-native.

There are 2 ways to deploy the container images

1. Imperatively using CLI:

A screen shot of a computer code

AI-generated content may be incorrect.

1. Declaratively using a YAML file:

## 📄 4. YAML Configuration – Declarative Management

The YAML file is used in the deployment phase—specifically with Cloud Run—to define the configuration of the service in a declarative way.

**🔍 What the YAML File Does:**

* Specifies:
  + Which container image to deploy
  + Environment variables (e.g., CITY=London)
  + Concurrency, timeout, memory, etc.
  + Whether the service is public or private
* Can be version-controlled in Git
* Can be reused across environments (dev, staging, prod)

There are **two types of YAML files** in this pipeline:

**a. cloudbuild.yaml**

* Used by **Cloud Build**
* Defines how to build and push the image

**b. delivery-pipeline.yaml and target.yaml**

* Used by **Cloud Deploy**
* Define the **deployment pipeline**, **targets**, and **configuration**
* Specify:
  + Which image to deploy
  + Where to deploy (Cloud Run, region)
  + Environment variables
  + Deployment strategy

✅ **Role**: Makes the pipeline **declarative**, version-controlled, and repeatable.

### YAML scripting:

**YAML** stands for **"YAML Ain't Markup Language"**, and it's widely used in **DevOps** and **cloud deployments** because of its simplicity and readability. In **Google Cloud Platform (GCP)**, YAML is commonly used to define configurations for services like **Cloud Build**, **Cloud Run**, **Kubernetes (GKE)**, and **IAM policies**.

**🧩 YAML Syntax Basics**

Here’s a breakdown of YAML syntax and structure:

**✅ Key Rules:**

* **Indentation matters**: Use spaces (not tabs).
* **Key-value pairs**: key: value
* **Lists**: Use - for items.
* **Comments**: Start with #
* **No commands**: YAML is for data, not logic.

**📦 Basic YAML Structure for GCP Deployment**

Let’s look at a simple example: a **Cloud Run** service deployment YAML.

A screenshot of a computer program

AI-generated content may be incorrect.

**🔍 Breakdown:**

* apiVersion: Specifies the API version.
* kind: Type of resource (e.g., Service, Deployment).
* metadata: Info like name and namespace.
* spec: The actual configuration.
  + template: Template for the pod.
    - containers: List of containers.
      * image: Docker image to deploy.
      * ports: Port exposed by the container.

### 🛠️ YAML in Cloud Build (cloudbuild.yaml)

Here’s a basic CI pipeline using Cloud Build:

A screenshot of a computer

AI-generated content may be incorrect.

**🔍 Breakdown:**

* steps: Each step runs in a container.
* name: Docker image used for the step.
* args: Arguments passed to the command.
* of images to push after build.

**✅ Declarative Deployment in Cloud Run (without Terraform)**

Yes, you **can do declarative deployments in Cloud Run** without using Terraform. GCP supports this via:

* **Cloud Deploy**: A managed CD service that allows you to define deployment pipelines declaratively using YAML.
* **Cloud Build**: You can use a cloudbuild.yaml file to define build and deploy steps.
* **gcloud CLI** or **GitHub Actions**: You can script deployments declaratively using configuration files and CI triggers.

So while Terraform is a powerful Infrastructure-as-Code tool, it's **not required** for declarative deployments in Cloud Run.

Using **YAML configuration files** for Cloud Run deployments allows you to manage your services **declaratively**, which means:

* You **define the desired state** of your service (e.g., image, environment variables, memory, concurrency).
* GCP takes care of **applying that state** during deployment.
* You can **version control** the YAML file in Git or any source control system.
* It supports **repeatable, auditable, and collaborative** infrastructure management.

**✅ Benefits of Declarative Deployment with YAML**

1. **Source Control Friendly**
   * You can track changes, collaborate via pull requests, and roll back easily.
2. **Consistency Across Environments**
   * Use the same YAML file for dev, staging, and production with minor tweaks.
3. **Automation Ready**
   * Integrates well with CI/CD tools like Cloud Build, GitHub Actions, or Terraform.
4. **Human-Readable Format**
   * Easy to understand and modify without needing to write imperative scripts.

A screenshot of a computer

AI-generated content may be incorrect.

You can deploy this using = gcloud run services replace service.yaml

A screenshot of a computer program

AI-generated content may be incorrect.

## Cloud Deploy - fully managed Continuous Delivery (CD)

**🔁 Does Cloud Deploy Automate the Whole Pipeline?**

Not entirely. It automates the **delivery and deployment** part of the pipeline, but it **does not replace** other tools like Cloud Build or Artifact Registry. Instead, it **orchestrates** them.

### 📦 What Cloud Deploy Does (and Doesn’t Do)

**✅ Cloud Deploy Controls:**

| **Component** | **Role of Cloud Deploy** |
| --- | --- |
| **Artifact Registry** | **Reads** the image from here to deploy it |
| **Cloud Run** | **Deploys** the image to Cloud Run (or GKE/Anthos) |
| **Deployment Pipeline** | **Defines** stages like dev → staging → prod |
| **Approvals & Rollbacks** | **Manages** release lifecycle and history |

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

# **Presentation**

#### ❗ Problem Statement

In today’s fast-growing **e-commerce sector**, platforms like Amazon and Flipkart have made online shopping incredibly convenient.  
However, with this rapid growth, new challenges have emerged—especially in **shipment tracking** and **logistics management**.  
Customers often face uncertainty about delivery timelines, while logistics teams struggle with coordination and visibility.

#### 🔍 Problem Overview

To address these issues, we developed the **Tracking-System-Application**—a solution designed to:

* Provide **real-time shipment tracking** for customers and admins
* Help logistics teams **monitor and manage deliveries** efficiently
* Ensure **transparency, automation, and scalability** using modern DevOps practices
* Integrate with **monitoring tools** like Prometheus and Grafana for system health visibility
* Support a seamless user experience with features like login, catalog browsing, cart management, and community engagement

#### ❓ Why Do We Use DevOps?

In modern applications like your **Tracking-System-Application**, DevOps is essential because:

* Manual deployments are slow and error-prone
* Infrastructure needs to scale dynamically
* Continuous updates and monitoring are critical
* Teams need to collaborate across development, QA, and operations

DevOps helps solve these challenges by introducing **automation, monitoring, and collaboration** into every stage of the software lifecycle.

##### ✅ Benefits of DevOps

1. **Faster Time to Market**
   * Automates build, test, and deployment processes for quicker releases
2. **Improved Collaboration**
   * Breaks down silos between development, operations, and QA teams
3. **Higher Deployment Frequency**
   * Enables continuous integration and delivery (CI/CD)
4. **Better System Reliability**
   * Uses monitoring tools (like Prometheus, Grafana, Datadog) to detect and resolve issues early
5. **Scalable & Repeatable Infrastructure**
   * Tools like **Terraform** and **Kubernetes** ensure consistent and scalable deployments

#### Why Hyperscalar

**🚀 1. End-to-End Integrated Ecosystem**

Hyperscalers offer a **fully integrated platform**:

* CI/CD (Cloud Build, Azure DevOps, CodePipeline)
* Source control (Cloud Source Repositories or GitHub integration)
* Artifact storage (Artifact Registry, S3, Azure Artifacts)
* Monitoring (Cloud Monitoring, CloudWatch, Azure Monitor)
* Security (IAM, VPC Service Controls, etc.)

➡️ **Advantage**: You don’t need to stitch together multiple tools manually. Everything works seamlessly.

**⚖️ 2. Scalability and Elasticity**

Hyperscalers provide **auto-scaling infrastructure**:

* Run builds on demand with auto-scaling runners.
* Deploy to Kubernetes clusters that scale based on traffic.
* Use serverless platforms (Cloud Run, Lambda, Azure Functions).

➡️ **Advantage**: You don’t worry about provisioning or maintaining build servers.

**🔐 3. Security and Compliance**

Built-in enterprise-grade security:

* IAM roles and policies
* Audit logging
* Encryption at rest and in transit
* Compliance with standards (ISO, SOC, HIPAA, etc.)

➡️ **Advantage**: Easier to meet enterprise and regulatory requirements.

**📊 4. Observability and Monitoring**

Hyperscalers provide:

* Centralized logging (e.g., GCP’s Cloud Logging)
* Tracing and metrics
* Dashboards and alerts

➡️ **Advantage**: Full visibility into your pipeline and infrastructure.

**💸 5. Cost Optimization**

* Pay-as-you-go pricing
* Spot instances, committed use discounts
* Serverless options reduce idle costs

➡️ **Advantage**: More efficient resource usage compared to always-on Jenkins servers.

**🔄 6. Automation and Infrastructure as Code**

* Terraform, Deployment Manager, CloudFormation
* GitOps workflows
* Policy-as-code (OPA, GCP’s Policy Controller)

➡️ **Advantage**: Declarative, repeatable, and auditable infrastructure.

**🧠 7. AI/ML and Advanced Services**

Hyperscalers offer:

* ML model training and deployment
* AutoML, Vertex AI, SageMaker
* BigQuery, Athena for analytics

➡️ **Advantage**: You can integrate intelligent features into your pipelines and apps.

**🧰 When Jenkins/GitHub Alone Might Be Enough**

* Small teams or projects
* Limited budget
* Simple CI/CD needs
* No need for cloud-native scalability

Git clone the given repository

* git clone https://github.com/saketh1809/Tracking-System-Project.git

Write a Dockerfile

Manually test by running the dockerfile and create a image and check

* docker build -t tracking-system-project:v1.1.0 .
* docker run -td --name tracking-system-project -p 3000:3000 tracking-system-project:v1.1.0
* docker images
* docker ps

Upload the docker image to hub.docker

* docker tag tracking-system-project:v1.1.0 saketh1809/tracking-system-project:v1.1.0
* docker push saketh1809/tracking-system-project:v1.1.0

Write a Docker-compose.yaml file for testing the backend

* sudo curl -L "https://github.com/docker/compose/releases/download/v2.24.6/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
* sudo chmod +x /usr/local/bin/docker-compose
* docker-compose –-version
* docker-compose up -d

Write deployment files for deploying the application in Kubernetes with loadbalancer

* kubectl apply -f tracking-system-deployment.yaml
* kubectl apply -f mongodb-tracking-deployment.yaml
* kubectl get svc

# **Prompt Engineering**

R – assign a role

G – specify goal

O – output

L – level of details

D – Direction: who it’s for and what tone

## 🔍 LLM (Large Language Model)

An **LLM** is a type of AI model trained on massive amounts of text data to understand and generate human-like language. Examples include GPT-4, Claude, and Gemini.

**Key Concepts:**

* **Transformer architecture**: The backbone of most LLMs, enabling them to handle long-range dependencies in text.
* **Pretraining**: LLMs are trained on diverse corpora to learn grammar, facts, reasoning, and more.
* **Prompting**: You guide the model's behavior using natural language instructions. Prompt engineering is the art of crafting these inputs to get optimal outputs.

**Prompt Engineering Tips:**

* Use **clear instructions**: e.g., “Summarize this article in bullet points.”
* Provide **context**: e.g., “You are a helpful assistant for software engineers.”
* Use **few-shot examples**: Show the model how to behave by giving examples.
* Experiment with **temperature and max tokens** to control creativity and length.

## 📚 RAG (Retrieval-Augmented Generation)

**RAG** combines LLMs with external knowledge sources to improve accuracy and relevance, especially for domain-specific or up-to-date queries.

**How RAG Works:**

1. **Retrieval**: A search engine or vector database fetches relevant documents based on the query.
2. **Augmentation**: These documents are passed to the LLM as context.
3. **Generation**: The LLM uses both the query and retrieved context to generate a response.

**Benefits:**

* Overcomes LLM limitations like outdated knowledge or hallucinations.
* Enables **grounded answers** based on real data.
* Useful in **enterprise search**, **chatbots**, **Q&A systems**, and **knowledge management**.

**Prompt Engineering in RAG:**

* Design prompts that **integrate retrieved context** effectively.
* Use delimiters like <<context>> to separate retrieved data from the query.
* Ensure prompts encourage **citation or attribution** when needed.

## vector database

A **vector database** is a specialized type of database designed to store and search **vector embeddings**—numerical representations of data like text, images, audio, or video. These embeddings are typically generated by machine learning models (especially LLMs or other neural networks) and capture semantic meaning.

**🧠 Why Vector Databases Matter in AI & RAG**

In **Retrieval-Augmented Generation (RAG)**, vector databases are used to:

* **Store embeddings** of documents, FAQs, knowledge bases, etc.
* **Retrieve semantically similar content** to a user query using **nearest neighbor search** (not keyword matching).
* Provide **contextual grounding** for LLMs to generate accurate, relevant responses.

**🔧 How It Works**

1. **Embedding Generation**:
   * Text like “What is a vector database?” is converted into a high-dimensional vector (e.g., 768-dim).
   * Example: [0.12, -0.45, 0.88, ..., 0.03]
2. **Storage**:
   * These vectors are stored in a vector database like **Pinecone**, **Weaviate**, **FAISS**, or **Milvus**.
3. **Similarity Search**:
   * When a user asks a question, it’s embedded and compared to stored vectors using **cosine similarity**, **Euclidean distance**, etc.
   * The most similar vectors (documents) are retrieved.
4. **Augmentation**:
   * Retrieved documents are passed to the LLM to generate a grounded response.

**🧰 Popular Vector Databases**

| **Name** | **Type** | **Features** |
| --- | --- | --- |
| **Pinecone** | Managed cloud | Scalable, fast, easy API |
| **Weaviate** | Open-source | Hybrid search (vector + keyword) |
| **FAISS** | Library (by Meta) | High-performance, local use |
| **Milvus** | Open-source | GPU acceleration, clustering |

### 🧠 What Are Embeddings?

**Embeddings** are **numerical representations** of data—like words, sentences, or images—converted into **vectors** (lists of numbers) that capture their **semantic meaning**.

For example:

* The word “king” might be represented as a vector like [0.25, -0.13, 0.88, ..., 0.02]
* The word “queen” will have a similar vector, close in meaning.

These vectors live in a **high-dimensional space** (e.g., 768 or 1536 dimensions), where **similar meanings are close together**.

**📌 Why Are Embeddings Useful?**

Embeddings allow machines to:

* **Understand context and meaning** beyond exact words.
* **Compare similarity** between texts (e.g., “What is AI?” vs “Explain artificial intelligence”).
* **Search semantically** using vector databases.
* **Cluster or classify** data based on meaning.

**🔍 How Are Embeddings Used?**

1. **In LLMs**:
   * Embeddings are used internally to understand and generate language.
   * You can also extract embeddings from models like OpenAI’s text-embedding-3 or Hugging Face models.
2. **In RAG systems**:
   * You embed documents and store them in a vector database.
   * When a user asks a question, you embed the query and retrieve similar documents.
3. **In recommendation engines**:
   * Embeddings help match users with similar interests or products.