**J Anjali**

**What is Docker Engine?**

**Docker Engine is the core software that allows you to build, run, and manage containers on your system.**

**🧠 Think of Docker Engine as the brain of Docker — it handles everything behind the scenes to make containerization work.**

**🧩 Components of Docker Engine**

**Docker Engine has three main parts:**

| **Component** | **Description** |
| --- | --- |
| **1. Docker Daemon (dockerd)** | **A background service that runs on your machine. It manages images, containers, networks, volumes, etc.** |
| **2. Docker CLI (docker)** | **The command-line tool you use to interact with the Docker Daemon (e.g., docker run, docker build).** |
| **3. REST API** | **Used internally or by tools like Docker Desktop to talk to the Docker Daemon programmatically.** |

**🔧 How Docker Engine Works**

1. **You run a command like docker run nginx**
2. **The CLI sends this command to the Docker Daemon**
3. **The daemon pulls the image, creates a container, and runs it**
4. **Your app is now isolated and running inside a container**

**git remote -v – What It Does**

**The command git remote -v is used to display the URLs of remote repositories linked to your local Git project.**

**The command git remote -v is used to view the remote repositories connected to your local Git project.**

**🔍 The -v stands for verbose, so it shows the remote name along with the fetch and push URLs.**

**✅ Why Use git remote -v?**

| **Purpose** | **Description** |
| --- | --- |
| **📍 See connected remotes** | **Shows which remote repositories your project is linked to** |
| **🔁 Check fetch/push URLs** | **Confirms where your code is pulled from or pushed to** |
| **🛠️ Troubleshooting** | **Helps fix remote-related issues (e.g., wrong repo, access issues)** |
| **🔧 Before updates** | **Useful before changing or removing a remote** |

**⚙️ In Jenkins, how to switch to a slave (agent) server for build execution?**

**In Jenkins, a "slave" server is now more commonly called a "node" or "agent". You can configure Jenkins to run jobs on a specific slave instead of the master node.**

**🧭 Step-by-Step: Run Jenkins Job on a Slave Node**

**✅ Step 1: Add the Slave Node (Agent)**

1. **Go to Jenkins Dashboard → Manage Jenkins**
2. **Click “Manage Nodes and Clouds”**
3. **Click “New Node”**
4. **Enter a name (e.g., build-agent-1) → Select “Permanent Agent” → Click OK**
5. **Fill in:**
   * **# of executors: Number of parallel builds this agent can handle**
   * **Remote root directory: Directory on the agent server**
   * **Labels: Keywords to identify the node (e.g., linux, slave1)**
   * **Launch method: Choose how Jenkins will connect (e.g., SSH)**
6. **Click Save and make sure it connects successfully.**

**✅ Step 2: Configure Job to Use the Slave**

1. **Open your Jenkins job**
2. **Click “Configure”**
3. **Under “General”, check “Restrict where this project can be run”**
4. **In the Label Expression field, enter the label of the slave (e.g., slave1, linux)**

**Now this job will run on the selected slave agent.**

**🧠 What Actually Happens**

* **Jenkins master schedules the job.**
* **Instead of running it on itself, it sends it to the agent with the matching label.**
* **The slave executes the build and reports back to master.**

**☁️ What is AWS CloudFront?**

**Amazon CloudFront is a Content Delivery Network (CDN) service that helps you deliver your website, videos, APIs, and other content faster and more securely to users around the world.**

**🚀 CloudFront speeds up the delivery of content by caching it at edge locations (data centers) near your users.**

**🔧 How CloudFront Works**

1. **User requests content (like an image, video, or webpage)**
2. **CloudFront checks its nearest edge location:**
   * **If content is cached, it delivers it immediately**
   * **If not, CloudFront fetches it from the origin (e.g., S3 or EC2), caches it, and then delivers**
3. **Future users near the same edge get the cached copy – faster and cheaper**

**🐳 Different Ways to Create Docker Images**

**You can create Docker images in multiple ways depending on your use case and how much control you need. Here's a breakdown of the most common methods:**

**🔧 1. Using a Dockerfile (Most Common Way)**

**A Dockerfile is a script with instructions to build a Docker image.**

**✅ Steps:**

1. **Create a file named Dockerfile**
2. **Add instructions like:**

**Dockerfile**

**CopyEdit**

**FROM ubuntu:20.04**

**RUN apt update && apt install -y python3**

**COPY . /app**

**CMD ["python3", "/app/app.py"]**

1. **Build the image:**

**bash**

**CopyEdit**

**docker build -t myapp:latest .**

**🛠️ 2. Using docker commit (Manual Method)**

**This captures changes made in a running container as a new image.**

**✅ Steps:**

1. **Start a container:**

**bash**

**CopyEdit**

**docker run -it ubuntu /bin/bash**

1. **Make changes inside the container (install packages, modify files)**
2. **Open a new terminal and run:**

**bash**

**CopyEdit**

**docker commit <container\_id> my-custom-image**

**🔸 Not recommended for production since it's hard to reproduce or version.**

**🔁 3. Using docker import**

**Create an image from a tarball archive of a filesystem.**

**✅ Example:**

**bash**

**CopyEdit**

**cat rootfs.tar | docker import - my-imported-image**

**Useful for migrating or restoring file system-based images.**

**💻 4. Using Docker Plugins or Tools (like BuildKit, Buildx)**

**Advanced tools that extend Docker’s build capabilities.**

**✅ Example:**

**bash**

**CopyEdit**

**docker buildx build --platform linux/amd64 -t myapp:multi .**

**Useful for multi-architecture builds and performance improvements.**

**☁️ 5. Using Docker Compose (Indirectly)**

**While docker-compose doesn't directly build images, you can define a build context in the docker-compose.yml file:**

**yaml**

**CopyEdit**

**services:**

**web:**

**build: .**

**image: myapp:compose**

**Then run:**

**bash**

**CopyEdit**

**docker-compose build**

**🧾 Summary Table**

| **Method** | **Use Case** | **Reproducible** | **Recommended** |
| --- | --- | --- | --- |
| **Dockerfile** | **Standard builds** | **✅ Yes** | **✅ Yes** |
| **docker commit** | **Manual/image snapshots** | **❌ No** | **⚠️ Rarely** |
| **docker import** | **From existing filesystem tarball** | **⚠️ Limited** | **🚫 Rare** |
| **Buildx/BuildKit** | **Advanced/multi-arch builds** | **✅ Yes** | **✅ Yes** |
| **Docker Compose build** | **Multi-container apps** | **✅ Yes** | **✅ Yes** |

**✅ Simple Meaning of Target in Prometheus**

**A target in Prometheus is just a place from which Prometheus collects data.**

**👉 It is usually a URL like localhost:9100/metrics  
👉 Prometheus goes there regularly to get system or app health data**

**🔁 Example:**

**If you're monitoring your server with Node Exporter,  
then this will be your target:**

**bash**

**CopyEdit**

**http://localhost:9100/metrics**

**Prometheus will go to this link and collect metrics like CPU, memory, disk usage, etc.**

**📌 Summary in Simple Words:**

| **Term** | **Simple Meaning** |
| --- | --- |
| **Target** | **A system or app URL that Prometheus reads data from** |
| **Scrape** | **The action of Prometheus collecting data** |
| **Metrics** | **The data Prometheus collects from the target** |

**🧱 Components of Kubernetes (K8s) – Simple Explanation**

**Kubernetes has two main types of components:**

1. **Control Plane Components – The brain (manages the cluster)**
2. **Node Components – The workers (run the applications)**

**🧠 1. Control Plane Components (Master)**

**These manage the whole Kubernetes cluster.**

| **Component** | **Simple Meaning** |
| --- | --- |
| **API Server (kube-apiserver)** | **📞 Entry point for all commands – takes input from kubectl** |
| **Scheduler (kube-scheduler)** | **📦 Decides which pod runs on which node** |
| **Controller Manager (kube-controller-manager)** | **🤖 Watches and maintains desired state (e.g., restart crashed pods)** |
| **etcd** | **📚 Key-value database to store all cluster data (config, state)** |
| **Cloud Controller Manager *(optional)*** | **☁️ Integrates Kubernetes with cloud providers (like AWS, GCP)** |

**⚙️ 2. Node Components (Worker Nodes)**

**These run your actual applications in Pods.**

| **Component** | **Simple Meaning** |
| --- | --- |
| **kubelet** | **🧑‍🏭 Talks to the control plane, runs the containers** |
| **kube-proxy** | **🌐 Manages networking on the node** |
| **Container Runtime** | **🐳 Runs the actual containers (like Docker, containerd, etc** |

**🛠️ Maven Goals and Build Lifecycle – Simple Explanation**

**Maven is a build automation tool used mainly for Java projects. It uses a build lifecycle with goals to compile, test, and package your code.**

**🔁 What is a Maven Build Lifecycle?**

**A build lifecycle is a sequence of steps (called phases) that define the order in which Maven will build your project.**

**There are 3 built-in lifecycles, but the most used is:**

**🎯 1. Default Lifecycle → For building and deploying your app**

| **Phase** | **What It Does** |
| --- | --- |
| **validate** | **Checks if the project is correct** |
| **compile** | **Compiles the source code** |
| **test** | **Runs unit tests** |
| **package** | **Packages the code (e.g., into JAR/WAR)** |
| **verify** | **Runs checks on the package** |
| **install** | **Installs the package to local repo** |
| **deploy** | **Uploads the package to remote repo** |

**⚙️ What is a Maven Goal?**

**A goal is a single task that Maven performs.**

* **Phases consist of one or more goals**
* **You can also run a specific goal directly**

**✅ Common Maven Goals and What They Do**

| **Goal** | **Description** |
| --- | --- |
| **clean** | **Deletes target/ folder (old builds)** |
| **compile** | **Compiles Java source code** |
| **test** | **Runs unit tests** |
| **package** | **Creates JAR/WAR file** |
| **install** | **Adds the package to local .m2 repo** |
| **deploy** | **Sends the package to remote repo** |

**🧪 Example Maven Commands**

| **Command** | **What It Does** |
| --- | --- |
| **mvn clean** | **Deletes the previous build** |
| **mvn compile** | **Compiles the source code** |
| **mvn package** | **Builds the app (e.g., JAR or WAR file)** |
| **mvn clean install** | **Clean + compile + test + package + install** |
| **mvn test** | **Runs your tests** |

**🔄 Lifecycle vs Goal Summary**

| **Term** | **Meaning** |
| --- | --- |
| **Lifecycle** | **A series of build steps (phases)** |
| **Phase** | **A stage in the lifecycle (e.g., compile)** |
| **Goal** | **A task to perform (e.g., compile source code)** |

**🐍 Functions in Python – Simple Explanation**

**A function in Python is a block of code that performs a specific task. You can call (use) a function whenever you want, instead of writing the same code again.**

**✅ Why Use Functions?**

* **Reuse code (no repetition)**
* **Makes code cleaner and easier to read**
* **Helps divide the program into smaller pieces**

**🐍 Methods in Python – Simple Explanation**

**A method is like a function, but it belongs to an object (such as a string, list, or class instance). You call methods on objects to perform actions related to that object.**

**🔑 Key Point:**

* **Functions can stand alone**
* **Methods are functions tied to an object and usually work with the data inside that object**

**🐍 Popular Python Frameworks – Simple Overview**

**Python has many frameworks that help you build applications faster by providing ready-made tools and structure.**

**🖥️ 1. Web Development Frameworks**

| **Framework** | **Description** | **Use Case** |
| --- | --- | --- |
| **Django** | **Full-featured, batteries-included** | **Large, complex web apps** |
| **Flask** | **Lightweight and simple** | **Small to medium web apps** |
| **FastAPI** | **Modern, fast, async support** | **APIs and microservices** |
| **Pyramid** | **Flexible and scalable** | **Customizable web apps** |

**Selenium.**

**🧱 What is EBS?**

Think of **EBS (Elastic Block Store)** like a **hard drive** (or SSD) for your **EC2 virtual machine** in AWS.

* Just like your laptop needs a hard disk to store files, your EC2 instance needs storage too.
* EBS gives **block-level storage** (which means it stores data in blocks, like a regular disk).
* It is **persistent**, meaning your data **doesn’t disappear** if you stop or restart the EC2 instance.
* **How it works:**

1. You create an EC2 instance (a virtual server).
2. You attach an EBS volume to it (like plugging in a hard drive).
3. The EC2 instance can now **read/write data** to that volume.
4. You can **detach** the EBS volume and **attach it to another EC2** if needed.

**📦 Key Features:**

| **Feature** | **Explanation** |
| --- | --- |
| **Persistent** | Data remains even after the EC2 is stopped. |
| **Scalable** | You can increase the size anytime. |
| **Types** | EBS has different types like gp3 (general purpose), io2 (high performance), etc. |
| **Snapshots** | You can take backups (called **snapshots**) and restore them later. |
| **Encrypted** | You can encrypt your data for security. |

**Simple Analogy:**

EC2 = Computer  
EBS = Hard Drive  
Snapshot = Backup copy of the hard drive

**Common Use Cases:**

* Store application data (like a database).
* Host the root volume of an EC2 (i.e., the OS).
* Keep logs, uploads, or other persistent files.

**📸 What is a Snapshot in AWS EBS?**

Think of a **snapshot** like taking a **photo (backup)** of your hard drive (EBS volume) at a moment in time.

* It’s stored in **Amazon S3** (behind the scenes, you don’t see this directly).
* It is **incremental**, meaning:
  + The first snapshot = full backup.
  + Later snapshots = only changes since the last one (saves space and time).
* You can use a snapshot to **restore** a new EBS volume.

**✂️ What is Copy Snapshot?**

**Copy Snapshot** means making a **copy of an existing snapshot**.

**Why would you copy a snapshot?**

1. **Move it to another AWS Region** (for disaster recovery or performance).
2. **Share it** with another AWS account.
3. **Encrypt it** (you can copy and enable encryption during the copy).
4. **Create backups** in different places for safety.

**🧠 Quick Analogy:**

| **Term** | **Analogy** |
| --- | --- |
| EBS Volume | Your laptop’s hard drive |
| Snapshot | A backup photo of that hard drive |
| Copy Snapshot | Making a duplicate of that photo and keeping it in a new place or format |

**📦 What does Placement mean in AWS?**

**Placement in AWS means where and how your EC2 instances are physically placed in AWS data centers.**

**Imagine AWS is a big hotel with many rooms (servers). Placement is about which room your EC2 instance goes into and whether it's next to or far away from other instances.**

When you launch EC2 instances, you don’t always get to choose **exactly** where they run. But sometimes, you might **want control over the physical location** — for **performance, availability, or licensing reasons**.

So AWS gives you **placement options**.

**📍 1. Availability Zone (AZ) Placement**

* **Default placement** is across **Availability Zones**.
* An **AZ** is a **data center or group of them** in a region.
* Example: If you launch 3 EC2s in the same **Region** but different AZs → You get **high availability**.

**🧱 2. Placement Group**

This is where you **take control** of how instances are placed **inside** a region/AZ.

There are 3 types:

| **Type** | **What It Does** | **Use Case** |
| --- | --- | --- |
| **Cluster** | All instances are placed **very close** together in the same rack | **High performance, low latency**, e.g., HPC apps |
| **Spread** | Instances are placed **far apart** across hardware | For **high availability**, reduce risk of hardware failure |
| **Partition** | Instances are split across **separate partitions** (hardware groups) | For large **distributed systems**, like HDFS, Cassandra |

**📦 3. Dedicated Host / Dedicated Instance Placement**

* You can place your EC2 on **dedicated physical hardware** (not shared).
* Useful for:
  + Software licensing (e.g., Windows Server BYOL).
  + Compliance or isolation needs.

**🔁 Summary Table:**

| **Term** | **Simple Meaning** |
| --- | --- |
| **Availability Zone** | General placement across AWS data centers |
| **Placement Group** | Control over instance proximity |
| **Cluster** | Put instances **close together** |
| **Spread** | **Spread them apart** |
| **Partition** | Divide them into **hardware groups** |
| **Dedicated Host** | Your own **physical server** in AWS |

**📂 What is Amazon EFS?**

**Amazon EFS (Elastic File System)** is a **serverless**, **fully managed**, **scalable** **file storage** service that can be used with **Linux-based EC2 instances and other services**.

🧠 Think of it like a **shared folder** in your office that multiple computers can access at the same time — **no setup, no capacity planning, just use and grow**.

**🧱 Type of Storage: File Storage**

AWS offers 3 main types of storage:

| **Service** | **Type** | **Example Use** |
| --- | --- | --- |
| **EBS** | Block storage | One EC2 instance only (like a hard disk) |
| **S3** | Object storage | Store files, backups, images |
| **EFS** | File storage | Shared file system, like a network drive |

**🧩 Key Features of EFS**

| **Feature** | **Description** |
| --- | --- |
| 🔄 **Shared Access** | Multiple EC2 instances (even across AZs) can **mount the same EFS** |
| 📈 **Scalable** | Grows and shrinks **automatically** as you add/remove files |
| ⚙️ **Fully Managed** | AWS handles servers, maintenance, availability |
| 🧩 **POSIX-Compatible** | Works like a normal Linux file system (ls, mkdir, chmod, etc.) |
| 🔒 **Secure** | Supports encryption (at rest & in transit), IAM policies, and NFS access control |
| 🗺️ **Regional** | Accessible across all AZs in a region |
| 💸 **Pay-As-You-Use** | You only pay for the storage you use (in GB) |

**🏗️ How It Works (Architecture)**

1. You create an **EFS file system**.
2. You **mount it** on one or more **Linux EC2 instances** using the **NFS protocol**.
3. EC2s can now **read/write to EFS like a local folder** (/mnt/efs).
4. As files are added/removed, EFS scales **automatically**.

**🔄 EFS Performance Modes**

| **Mode** | **Description** | **Use Case** |
| --- | --- | --- |
| **General Purpose** (default) | Low latency, good for most apps | Web servers, content management |
| **Max I/O** | Higher throughput, slightly more latency | Big data, analytics, large-scale apps |

**💰 EFS Storage Classes (to Save Cost)**

| **Class** | **Description** | **Cost** |
| --- | --- | --- |
| **Standard** | Frequent access | Higher |
| **Infrequent Access (IA)** | For files not often accessed | Lower (good for backups/archive) |

You can **enable Lifecycle Management** to move old files automatically to IA.

**✅ Use Cases for EFS**

* Shared file storage for **web applications**
* Centralized **configuration files**
* Shared **logs or backups**
* **Machine learning** or **data analytics**
* **Content management systems** (CMS)

**🛠️ Steps to Use EFS (Basic Workflow)**

1. Go to AWS Console > EFS > Create File System.
2. Choose VPC, security group, mount targets (for each AZ).
3. Launch EC2 instances (Linux).
4. Use the EFS mount helper to mount it:

bash

CopyEdit

**sudo mount -t efs fs-xxxx:/ /mnt/efs**

1. Done! Now EC2 can read/write files to the shared EFS.

**🔐 Security in EFS**

* IAM policies (who can create or access EFS)
* **Security groups** (controls network access)
* **Encryption at rest** using KMS
* **Encryption in transit** using TLS (via mount helper)

**🪣 What is Amazon S3?**

**Amazon S3** is a **cloud-based object storage service** where you can **store and retrieve any amount of data** — like files, images, videos, backups, documents, and more.

🧠 Think of S3 like an **infinite USB drive in the cloud**, accessible from anywhere.

**🗂️ How is S3 organized?**

S3 uses a simple structure:

* **Bucket** = Like a **folder** that stores your data.
* **Object** = The **actual file/data** you upload (e.g., photo, PDF, ZIP file).
* **Key** = The **name** of the object (like the file path).

Example:

| **Component** | **Example** |
| --- | --- |
| Bucket | my-photo-storage |
| Object | vacation/beach.jpg |
| Key | vacation/beach.jpg |

**🧩 Key Features**

| **Feature** | **Description** |
| --- | --- |
| ☁️ **Unlimited Storage** | Store as much data as you want |
| 🌎 **Accessible Globally** | From anywhere via internet |
| 📂 **Object Storage** | Stores files, not block devices |
| 🛡️ **Secure** | Supports encryption, IAM policies, bucket policies |
| 🔄 **Versioning** | Keeps **multiple versions** of the same object |
| 🗑️ **Lifecycle Rules** | Automatically move or delete files after some time |
| 📁 **Folders (Pseudo)** | Uses object key names to simulate folders |
| 🔗 **Static Website Hosting** | Host websites directly from S3 |
| 🧾 **Logging** | Tracks access and actions for security/auditing |
| 🔐 **Access Control** | IAM, bucket policies, ACLs, signed URLs |

**🛠 Common Use Cases**

* 📸 Store media files (images, videos, audio)
* 🗄️ Backup and archive data
* 🌐 Host static websites
* 📤 Upload/download user files (e.g., resumes, invoices)
* 📦 Store logs or analytics data
* 🤖 Input/output for data processing (Lambda, ML jobs, etc.)

**🧪 Storage Classes (Cost Saving Options)**

S3 offers **different storage classes** based on how often you access the data:

| **Class** | **Use Case** | **Cost** |
| --- | --- | --- |
| **Standard** | Frequent access | 💲💲 |
| **Intelligent-Tiering** | Auto move between classes | 💲+ |
| **Standard-IA** | Infrequent access | 💲 |
| **One Zone-IA** | Infrequent, one AZ only | 💲 |
| **Glacier** | Archive, minutes to hours to access | 💰 (cheap) |
| **Glacier Deep Archive** | Long-term archive, hours to access | 💰💰 (very cheap) |

**🔐 S3 Security**

* **IAM** (who can do what)
* **Bucket Policies** (rules at bucket level)
* **ACLs** (access control lists per object)
* **Encryption**:
  + **SSE-S3** (managed by AWS)
  + **SSE-KMS** (you manage keys)
* **Public Access Settings** (block public access by default)

**🔄 Versioning**

* Keeps multiple versions of the same object.
* Helps you **recover from accidental deletion or overwrite**.

**📉 Lifecycle Rules**

* Move files to cheaper storage (e.g., Glacier) after 30 days.
* Delete old versions after 90 days.
* **Automatic cost savings**.

**📊 S3 vs EBS vs EFS**

| **Feature** | **S3** | **EBS** | **EFS** |
| --- | --- | --- | --- |
| Storage Type | Object | Block | File |
| Use Case | Files, backups, websites | One EC2's disk | Shared file system |
| Access | API (HTTPS) | OS-level | Mount |
| Multi-instance Access | Yes | No | Yes |
| Scalability | Auto | Manual | Auto |

**🧠 Analogy**

Imagine S3 as **Google Drive** for your applications.  
You can **store, share, secure**, and **access files** from anywhere — with **infinite space** and **fine-grained control**.

**📥 Example: Uploading to S3 (Console)**

1. Go to AWS Console → S3
2. Create a **bucket** (e.g., my-app-files)
3. Upload files (drag-and-drop or browse)
4. Set permissions (public/private)
5. Done! You can access the file via a URL if allowed.

**🪣 What Are S3 Storage Classes?**

S3 has **multiple storage classes** to help you **store data at the right cost** based on **how often** and **how quickly** you need to access it.

Think of it like choosing between a **fast locker**, a **cheap archive**, or a **smart automatic storage** — all in the cloud.

**📊 S3 Storage Classes Comparison**

| **Storage Class** | **Best For** | **Access Speed** | **Durability** | **Availability** | **Min Storage Time** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- |
| **S3 Standard** | Frequently accessed data | Milliseconds | 99.999999999% (11 9s) | 99.99% | None | Websites, apps, hot data |
| **S3 Intelligent-Tiering** | Data with unknown access patterns | Milliseconds | 11 9s | 99.9%–99.99% | 30/90 days (some tiers) | Logs, unpredictable data |
| **S3 Standard-IA** (Infrequent Access) | Infrequently accessed, but needed fast | Milliseconds | 11 9s | 99.9% | 30 days | Backups, older docs |
| **S3 One Zone-IA** | Infrequent access, low-cost | Milliseconds | 11 9s | 99.5% | 30 days | Re-creatable data (e.g. cache) |
| **S3 Glacier** | Archival storage (cheap) | Minutes to hours | 11 9s | Varies | 90 days | Compliance, long-term backup |
| **S3 Glacier Deep Archive** | Long-term cold archive | Hours | 11 9s | Varies | 180 days | Tape replacement, archive only |

**💡 Quick Summary Table**

| **Class** | **Access** | **Cost** | **Use** |
| --- | --- | --- | --- |
| 🔥 **Standard** | Frequent | $$$ | Main apps, daily use |
| 🤖 **Intelligent-Tiering** | Auto-manages | $$ | When you don’t know access pattern |
| ❄️ **Standard-IA** | Infrequent | $ | Quick access to old data |
| 🧊 **One Zone-IA** | Infrequent, 1 AZ | $ (cheaper) | Recoverable data |
| 🏔️ **Glacier** | Archive | $$ (low) | Long-term storage |
| 🕳️ **Glacier Deep Archive** | Very cold | $ (very low) | Compliance, rarely accessed |

**🔄 Intelligent-Tiering: How it Works**

* Starts in **frequent access tier**
* Moves data automatically to:
  + **Infrequent Access Tier**
  + **Archive/Deep Archive Tier** (optional)
* No performance loss
* Small monthly monitoring fee (~$0.0025 per object)

**🌐 What is a VPC?**

**Amazon VPC (Virtual Private Cloud)** is like your own **private network inside AWS** — where you can **launch and control** resources (like EC2, RDS) securely.

🧠 Think of it like building your own data center inside AWS — with full control over networking.

**🧩 Key VPC Components**

Here are the **main building blocks** of a VPC:

**1. 🧱 VPC**

* A **virtual network** in AWS.
* You define its **IP range** using a **CIDR block** (e.g., 10.0.0.0/16).
* You launch AWS resources (like EC2s) inside it.

**2. 📦 Subnets**

* Smaller networks **inside a VPC**.
* Two types:
  + **Public Subnet** – connected to the internet.
  + **Private Subnet** – isolated from the internet.
* Subnets are **AZ-specific**.

🧠 Think of subnets as rooms in a building (your VPC).

**3. 🌍 Internet Gateway (IGW)**

* Lets instances in **public subnets** access the **internet**.
* Must be **attached to your VPC**.
* Public EC2s need:
  + Public IP
  + Route to IGW
  + IGW attached

**4. 🚪 NAT Gateway / NAT Instance**

* Lets **private instances** access the **internet** (e.g., for updates) **without exposing them** to inbound internet traffic.
* Placed in **public subnet**.
* NAT = **Network Address Translation**.

**5. 📜 Route Tables**

* Control **where traffic goes** inside the VPC.
* Subnets are associated with route tables.
* Used to send traffic:
  + Within VPC
  + To IGW (internet)
  + To NAT
  + To VPN/peering

**6. 🔐 Security Groups**

* Like a **firewall** for EC2 instances.
* Controls **inbound and outbound traffic**.
* Instance-level (not subnet-level).
* **Stateful** – response traffic is allowed automatically.

**7. 🧱 Network ACLs (NACLs)**

* Optional **subnet-level firewall**.
* Controls **traffic in and out of subnets**.
* **Stateless** – must allow both inbound and outbound explicitly.
* Good for extra security.

**8. 🧭 DHCP Options Set**

* Controls **DNS and domain name settings** inside the VPC.
* By default, AWS provides its own DNS.

**9. 🛣️ VPC Peering**

* Connects two VPCs (same or different AWS accounts) **privately**.
* Allows **resources in each VPC to talk** to each other.

**10. 🔐 VPN Gateway & Customer Gateway**

* Used to create a **secure VPN connection** between AWS VPC and your **on-premises network**.
* **VPN Gateway** = on AWS side
* **Customer Gateway** = your company’s router/firewall

**11. 🛰️ Transit Gateway**

* Connects **multiple VPCs and on-prem networks** through a **central hub**.
* Scalable alternative to many VPC peering connections.

**✅ What is AWS Trusted Advisor?**

**AWS Trusted Advisor** is like a **smart assistant** for your AWS account.  
It helps you **optimize performance, security, reliability, and cost** by **analyzing your AWS environment** and giving **recommendations**.

Linux  
Linux plays a **foundational role in DevOps** due to its flexibility, open-source nature, and powerful command-line interface. Here's how Linux fits into DevOps workflows and practices:

**🔧 Why Linux is Crucial in DevOps**

**1. Infrastructure and Server Management**

* Most **cloud servers** (AWS, Azure, GCP) and **containers (like Docker)** run on Linux.
* DevOps tools like **Ansible**, **Terraform**, and **Kubernetes** are designed with Linux in mind.
* Linux skills help with configuring services, networking, and firewalls (e.g., using iptables, netstat, systemctl).

**2. Automation and Scripting**

* Shell scripting (e.g., Bash) automates repetitive tasks: deployments, backups, log rotation, monitoring, etc.
* Cron jobs are used for task scheduling.
* Easy integration with CI/CD tools like Jenkins.

**3. Containerization**

* Docker is **natively built for Linux** using container technologies like **cgroups** and **namespaces**.
* Managing Docker images, containers, and Dockerfiles on Linux is more efficient.

**4. CI/CD Pipelines**

* Jenkins agents often run on Linux.
* Build tools (e.g., Maven, Gradle), test runners, and deployment scripts are run on Linux-based runners or VMs.

**5. Monitoring and Logging**

* Tools like **Prometheus**, **Grafana**, **ELK Stack (Elasticsearch, Logstash, Kibana)** are primarily Linux-native.
* System logs are stored in /var/log and analyzed for system health and performance.

**6. Security and Access Control**

* Use of ssh, file permissions (chmod, chown), and user/group management (useradd, usermod) is essential.
* Secrets and credentials are managed securely in environments like Linux with vault tools.

Shell scripting is writing a sequence of commands for the shell to automate tasks. In most DevOps environments, **Bash (Bourne Again Shell)** is used on Linux systems.

**🔐 What is SSH?**

**SSH (Secure Shell)** is a way to **safely connect to another computer** over the internet or a network.

**🧠 Think of it like:**

A **secure remote control** for another computer.

**💻 What can you do with SSH?**

* **Login** to a remote server (like your cloud server).
* **Run commands** on that server.
* **Transfer files** between your computer and the server.
* **Automate tasks** in DevOps (using tools like Ansible or Jenkins).

**🛠️ Example**

If you have a server at 192.168.1.10 and your username is john, you can log in like this:

bash

CopyEdit

Git

**🔄 git reset – Simple Explanation**

git reset is a Git command used to **undo changes** in your repository.

**📦 Basic Types of git reset**

There are **three main types**, based on how much you want to undo:

| **Command** | **What It Does** | **Affects** |
| --- | --- | --- |
| --soft | Undo commits, keep changes in **staging** | Commit history only |
| --mixed (default) | Undo commits and **unstage** changes | Commit + Staging |
| --hard | Undo everything, discard all **local changes** | Commit + Staging + Working directory |

**🧪 Examples**

**1. 🔙 Undo Last Commit, Keep Changes**

bash

CopyEdit

git reset --soft HEAD~1

Removes the last commit, but keeps your code changes staged.

**2. 🔄 Undo Commit + Unstage Changes**

bash

CopyEdit

git reset --mixed HEAD~1

Removes the last commit and unstages the files (they remain on disk, not deleted).

**3. ⚠️ Completely Erase Last Commit & Changes**

bash

CopyEdit

git reset --hard HEAD~1

**DANGER:** Deletes the commit **and** your local file changes.

**📌 Undo Changes to a Specific File**

bash

CopyEdit

git reset HEAD file.txt

Unstages file.txt, but keeps the changes in the working directory.

**🚨 Be Careful**

* --hard can permanently delete work.
* Avoid using it if you've already pushed the commit to others.

**🧠 Tip**

If you've pushed a commit and want to "undo" it safely, use:

bash

CopyEdit

git revert <commit-hash>

It creates a new commit that undoes the old one—safe for shared repos.

**⚔️ What Are Git Conflicts?**

A **Git conflict** happens when **two people (or branches) change the same part of a file**, and Git doesn’t know which version to keep.

**🧠 Simple Example**

You and your teammate both edit the same line in app.js.

* You: console.log("Hello world");
* Teammate: console.log("Hi world");

When you try to **merge**, Git gets confused and says:

"I don't know which version to use. You decide."

**🧪 When Do Conflicts Happen?**

| **Situation** | **Example** |
| --- | --- |
| 🔀 Merge | git merge branch-name |
| ⬅️ Rebase | git rebase branch-name |
| ↩️ Cherry-pick | git cherry-pick commit |

**🛠️ How to Fix a Conflict**

1. Run a merge and get a conflict:

git merge branch-name

Git will show:

CONFLICT (content): Merge conflict in file.txt

1. Open the file and look for conflict markers:

txt

CopyEdit

<<<<<<< HEAD

Your version

=======

Other branch's version

>>>>>>> branch-name

1. Edit the file and keep the correct version.
2. Stage the resolved file:

bash

CopyEdit

git add file.txt

1. Commit the merge:

git commit

Rebase

It is a powerful tool used to integrate the changes from one branch to another by moving or combining sequence of commits into a new base commit.

**📦 What is Maven in DevOps?**

**Maven** is a **build automation and project management tool** used mainly for **Java-based projects**. It simplifies compiling code, running tests, managing dependencies, packaging applications, and deploying them—all crucial for DevOps pipelines.

**🧠 Simple Explanation**

Maven is like a **recipe manager** for Java projects. It knows what ingredients (dependencies) you need, how to prepare them (build), and how to serve (deploy) the final dish (your application).

**🔧 Key Features of Maven**

| **Feature** | **Description** |
| --- | --- |
| 📁 Standard Structure | Follows a clear directory format (like src/main/java) |
| ⚙️ Dependency Management | Handles all JARs and libraries via pom.xml |
| 🔨 Build Lifecycle | Compile, test, package, install, and deploy |
| 🧪 Plugin System | Extend Maven with plugins (e.g., Surefire for testing) |
| 🤝 CI/CD Friendly | Easy to integrate with Jenkins, Git, Docker, etc. |

**📄 pom.xml – The Heart of Maven**

xml

CopyEdit

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>my-app</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>3.1.0</version>

</dependency>

</dependencies>

</project>

**🔁 Maven Lifecycle Phases**

| **Phase** | **What it Does** |
| --- | --- |
| validate | Check if project is correct |
| compile | Compile the source code |
| test | Run unit tests |
| package | Create a JAR/WAR |
| install | Install to local repo |
| deploy | Upload to remote repo (like Nexus or Artifactory) |

**🛠️ Common Maven Commands**

bash

CopyEdit

mvn clean # Remove old compiled files

mvn compile # Compile the source code

mvn test # Run tests

mvn package # Create a JAR/WAR file

mvn install # Install to local Maven repository

mvn deploy # Deploy to remote repo

**🧪 Maven in DevOps Pipelines**

| **DevOps Stage** | **Maven Use** |
| --- | --- |
| 🛠️ Build | Compile and package code |
| 🧪 Test | Run unit and integration tests |
| 🚀 Deploy | Push artifacts to artifact repositories |
| 🔁 CI/CD | Automate with Jenkins, GitLab CI, or GitHub Actions |

**⚙️ Example in Jenkins Pipeline**

groovy

CopyEdit

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'mvn clean install'

}

}

stage('Test') {

steps {

sh 'mvn test'

}

}

stage('Package') {

steps {

sh 'mvn package'

}

}

}

}

**🧰 Tools Maven Works Well With**

* **Jenkins** – Build automation
* **SonarQube** – Code quality scanning
* **Docker** – Containerize Maven-built apps
* **Nexus / Artifactory** – Artifact repositories
* **Selenium** – Automated testing via Maven plugin

**🧰 Types of Jenkins Jobs & Their Differences**

In Jenkins, a **Job** (also called a "project") is a task that Jenkins runs—like building code, running tests, or deploying an application. Jenkins supports multiple job types depending on how complex or flexible your CI/CD needs are.

**📋 1. Freestyle Project**

**✅ Description:**

A simple, GUI-based job with basic configuration steps.

**🛠️ Use Case:**

* Compile and build code
* Run shell scripts
* Basic automation without code

**📌 Example:**

text

CopyEdit

- Pull code from Git

- Run build command (e.g., mvn install)

- Archive artifacts

**🧾 Configuration:**

Done via Jenkins GUI.

**📜 2. Pipeline Job**

**✅ Description:**

A job defined **as code** in a Jenkinsfile. You can version it in Git.

**🛠️ Use Case:**

* Complex CI/CD pipelines
* Multi-stage deployments
* Integrates easily with DevOps tools

**🧾 Syntax Types:**

* **Declarative** (easier, structured)
* **Scripted** (more flexible, harder)

**📌 Example:**

groovy

CopyEdit

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'mvn clean install'

}

}

}

}

**🌿 3. Multibranch Pipeline**

**✅ Description:**

Automatically creates and runs pipeline jobs for **each branch** in your Git repository.

**🛠️ Use Case:**

* Teams working on feature branches
* Automating builds per branch
* PR (pull request) testing

**📌 Example:**

GitHub repo with dev, test, main → Jenkins auto-creates jobs for each branch.

**🧪 4. Multi-Configuration (Matrix) Job**

**✅ Description:**

Runs the same job with **different parameters** or **environments** (e.g., Java versions, OS types).

**🛠️ Use Case:**

* Cross-platform testing
* Compatibility testing
* Multiple JDK or DB versions

**📌 Example:**

Build and test with:

* JDK 8 and JDK 11
* Ubuntu and Windows

**🗂️ 5. Folder**

**✅ Description:**

Organize related jobs into **folders** for better structure.

**🛠️ Use Case:**

* Large projects or teams
* Separate environments or microservices

**📎 6. External Job *(rarely used)***

**✅ Description:**

Used to monitor jobs **run outside Jenkins**.

**🛠️ Use Case:**

* Legacy external jobs
* Scripts running on other systems

| **Job Type** | **GUI/Code** | **Use Case** | **Auto Branch Detection** | **Parallel Runs** |
| --- | --- | --- | --- | --- |
| **Freestyle Project** | GUI | Basic builds, scripts | ❌ | ❌ |
| **Pipeline** | Code | Complex CI/CD workflows | ❌ | ✅ |
| **Multibranch Pipeline** | Code | Multi-branch automation | ✅ | ✅ |
| **Multi-Config (Matrix)** | GUI | Multiple env/config testing | ❌ | ✅ |
| **Folder** | GUI | Job organization | ❌ | ❌ |
| **External Job** | Manual | Track outside Jenkins jobs |  |  |

**Triggers in Jenkins – Easy Explanation**

**Triggers** in Jenkins are ways to **automatically start a job** (build) when something happens — like a code push, time schedule, or external event.

**🔔 Common Jenkins Triggers (Simple & Easy)**

| **Trigger Type** | **What It Does** | **Example Use** |
| --- | --- | --- |
| ⏰ **Timer (Cron)** | Run jobs on a schedule | Nightly builds |
| 🌐 **SCM Polling** | Check Git repo regularly for changes | Every 5 mins |
| 🧑‍💻 **Webhook (GitHub/GitLab)** | Run job **immediately** after a push | CI pipelines |
| 🔗 **Upstream Job** | Run after another job completes | Chained jobs |
| 📦 **Manual Trigger** | Start job by clicking "Build Now" | On-demand |

**🔧 1. Timer Trigger (Cron Job)**

⏱️ Run the job automatically at certain times.

**In GUI**:

* Check “Build periodically”
* Use Cron format:

text

CopyEdit

H 2 \* \* \* → Every day at 2 AM

H/5 \* \* \* \* → Every 5 minutes

**🌐 2. SCM Polling**

🔁 Jenkins checks your Git repo every few minutes.

**In GUI**:

* Enable “Poll SCM”
* Enter Cron:

text

CopyEdit

H/2 \* \* \* \* → Poll every 2 minutes

📝 **Does NOT trigger automatically on push** — it only checks.

**⚡ 3. Webhook Trigger (GitHub/GitLab)**

✅ Best for **real-time builds** when code is pushed.

**Steps**:

1. In Jenkins:
   * Enable “GitHub hook trigger for GITScm polling”
2. In GitHub:
   * Go to **Settings > Webhooks**
   * Add URL:

perl

CopyEdit

http://<your-jenkins-url>/github-webhook/

🎯 Jenkins gets notified **instantly** when code is pushed.

**🔗 4. Trigger from Another Job (Upstream Job)**

📦 Useful when job B should run after job A.

**In Job B**:

* Enable: “Build after other projects are built”
* Enter job A’s name

**🧑‍💻 5. Manual Trigger**

🖱️ Just click **“Build Now”** in the Jenkins GUI.

**🧪 In a Jenkinsfile (Declarative Pipeline)**

groovy

CopyEdit

pipeline {

agent any

triggers {

cron('H 1 \* \* \*') // Daily at 1 AM

pollSCM('H/15 \* \* \* \*') // Poll Git every 15 mins

}

stages {

stage('Build') {

steps {

sh 'echo Building...'

}

}

}

}

In Maven, the **pom.xml** file (short for **Project Object Model**) is the **core configuration file** used to manage a Java project. It defines the project structure, dependencies, build settings, and more. Here's a breakdown of its key roles:

**🔧 What Does pom.xml Do?**

1. **Project Metadata**  
   It contains basic information about the project such as:
   * groupId: The group or organization the project belongs to.
   * artifactId: The name of the project.
   * version: The current version of the project.
2. **Dependencies**  
   You declare external libraries (like JUnit, Spring, etc.) that your project needs. Maven will automatically download them from a central repository.
3. **Build Configuration**  
   You can define how the project should be built, including plugins, goals, and phases.
4. **Repositories**  
   Specifies where Maven should look for dependencies (e.g., Maven Central or custom repositories).

5 **Profiles**  
Allows you to define different configurations for different environments (e.g., dev, test, production).

3. What is the maven build lifecycle?

The **Maven Build Lifecycle** is the sequence of phases that define the order in which the goals (tasks) are executed during a build process.

Maven has **three built-in lifecycles**, but the most commonly used one is the **"default"** lifecycle.

**🔄 1. Maven Default Build Lifecycle (Most Common)**

This lifecycle handles **project deployment** and consists of the following **main phases**:

| **Phase** | **Description** |
| --- | --- |
| validate | Checks if the project structure is correct and all necessary info is available. |
| compile | Compiles the source code of the project. |
| test | Runs unit tests using a suitable testing framework like JUnit or TestNG. |
| package | Packages the compiled code into a JAR, WAR, or other formats. |
| verify | Runs any checks to verify the package is valid and meets quality criteria. |
| install | Installs the package into the local Maven repository for use in other projects. |
| deploy | Copies the final package to a remote repository for sharing with others. |

**🧪 2. Maven Clean Lifecycle**

Used to clean the project and remove files generated in previous builds.

| **Phase** | **Description** |
| --- | --- |
| pre-clean | Do work before cleaning. |
| clean | Deletes the target/ directory. |
| post-clean | Do work after cleaning. |

**📦 3. Maven Site Lifecycle**

Used to generate project documentation.

| **Phase** | **Description** |
| --- | --- |
| pre-site | Tasks before generating the site. |
| site | Generates the site documentation. |
| post-site | Tasks after site generation. |
| site-deploy | Deploys the site to a web server. |

**✅ What Are Dependencies in Maven?**

**Dependencies** are external **libraries or components** your project needs to compile, run, or test successfully.

In a **Maven** project, dependencies are declared in the pom.xml file, and Maven automatically downloads them from **remote repositories** like Maven Central.

**🚀 Which AWS Service Is Used for Application Building?**

There isn’t just one—**multiple AWS services** support **application building**, depending on the stage (code, build, test, deploy). Here are the key services used in **application building**, especially in DevOps workflows like yours:

| **Service** | **Purpose** |
| --- | --- |
| **AWS CodeCommit** | Git-based **source control** (like GitHub, Bitbucket). |
| **AWS CodePipeline** | Automates **CI/CD pipeline** – connects CodeCommit → CodeBuild → CodeDeploy. |
| **AWS CodeDeploy** | Automates **deployment** to EC2, Lambda, or on-prem servers. |
| **AWS CodeArtifact** | Stores and shares **build artifacts** (like Maven repos). |
| **Amazon S3** | Stores build outputs (artifacts, logs, etc.). |

Python

**✅ Advantages of Python**

Python is one of the most popular and versatile programming languages today. Here are the **key advantages** that make Python a go-to language for beginners, professionals, and companies:

**🧠 1. Simple and Easy to Learn**

* Python uses **clear, human-readable syntax** (like English).
* Ideal for beginners in programming and quick prototyping.

**⚡ 2. Versatile and Multi-Purpose**

* Used in **web development**, **data science**, **AI/ML**, **automation**, **DevOps**, **scripting**, etc.
* Works on almost any platform (Windows, Linux, Mac).

**📚 3. Massive Libraries and Frameworks**

* Rich ecosystem of **libraries**:
  + **Web**: Django, Flask
  + **Data Science**: Pandas, NumPy, Scikit-learn
  + **ML/AI**: TensorFlow, PyTorch
  + **DevOps**: Boto3 (AWS SDK), Ansible modules

**🤝 4. Strong Community Support**

* Huge global community and tons of **tutorials, forums, and open-source projects**.
* Easy to find solutions or ask questions (StackOverflow, GitHub, etc.).

**🚀 5. Productivity and Speed**

* Less code, fewer bugs.
* Faster development and deployment cycles compared to Java or C++.

**🧪 6. Great for Testing and Automation**

* Commonly used for **test automation**, **CI/CD scripting**, and **DevOps** tasks.
* Integrates easily with Jenkins, Selenium, Docker, and AWS SDKs.

**🌐 7. Cross-Platform Compatibility**

* Write once, run anywhere (with Python installed).
* Portable across all major operating systems.

**🧬 8. Dynamic Typing and Flexibility**

* No need to declare variable types explicitly.
* Supports multiple programming paradigms: **OOP**, **procedural**, and **functional**.

2. Frameworks in python for web developmen

| **Framework** | **Type** | **Best For** | **Built-in Features** |
| --- | --- | --- | --- |
| Django | Full-stack | Complex, secure apps | Yes (ORM, Auth, Admin) |
| Flask | Micro | Lightweight apps & APIs | Minimal |
| FastAPI | Micro | Fast async APIs | Docs, validation |
| Pyramid | Flexible | Scalable apps (flexible stack) | Partial |
| Tornado | Async | Real-time web services | Networking support |

3. Write a program to print the value of a variable in pythonmessage = "Hello, World!"

# Print the value of the variable

print(message)

1. Various Data types in python
2. **🔢 1. Numeric Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| int | Integer numbers | x = 10 |
| float | Floating-point (decimal) numbers | y = 3.14 |
| complex | Complex numbers | z = 2 + 3j |

1. **📝 2. Text Type**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| str | String (text data) | name = "Alice" |

1. **✅ 3. Boolean Type**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| bool | Logical values (True / False) | is\_valid = True |

1. **📦 4. Sequence Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| list | Ordered, mutable collection | fruits = ['apple', 'banana'] |
| tuple | Ordered, immutable collection | point = (1, 2) |
| range | Sequence of numbers | numbers = range(5) |

Purpose of Terraform state file

The **Terraform state file (terraform.tfstate)** is a critical component in Terraform's infrastructure as code workflow. Its primary purpose is to **track the current state of your infrastructure** so that Terraform can manage and update resources efficiently and accurately.

**🔍 Key Purposes of the terraform.tfstate File:**

1. **Tracks Resource Metadata**
   * Maintains a mapping between real-world infrastructure and your Terraform configuration.
   * Stores resource IDs, attributes, and dependencies.
2. **Facilitates Plan and Apply**
   * Terraform compares the state file with your configuration files to determine **what changes** need to be made during terraform plan and terraform apply.
3. **Supports Incremental Changes**
   * Enables Terraform to **update only what has changed** instead of recreating everything.
   * Ensures resources are not unnecessarily destroyed or duplicated.
4. **Ensures Idempotency**
   * Prevents the same configuration from producing different results when applied multiple times.
5. **Required for Collaboration (with Remote State)**
   * In team environments, storing state in a **remote backend (e.g., S3, Terraform Cloud)** ensures everyone works from a **single source of truth**, avoiding conflicts.

**⚙️ Auto Scaling Policies in AWS**

**Auto Scaling Policies** in AWS automatically adjust the number of EC2 instances (or other scalable resources) in an **Auto Scaling Group (ASG)** based on predefined rules or metrics — helping you maintain performance and optimize cost.

**🔄 Types of Auto Scaling Policies**

**1. Target Tracking Scaling Policy**

* **Most commonly used**
* Automatically adjusts capacity to keep a metric (like CPU utilization) at a **target value**
* **Example:** Keep average CPU usage at 50%

✅ **Best for:** Simple, automatic scaling without needing custom thresholds

json

CopyEdit

"TargetValue": 50.0

**2. Step Scaling Policy**

* Scales based on **specific metric thresholds** and **step adjustments**
* You define how much to scale based on how far the metric is from the threshold

**Example:**

* CPU > 70% → add 2 instances
* CPU > 85% → add 4 instances

✅ **Best for:** More **control and precision** over scaling actions

**3. Simple Scaling Policy (Legacy)**

* Triggers a scaling action when a CloudWatch alarm is triggered
* Adds or removes a fixed number of instances

**Example:** If CPU > 70%, add 1 instance

✅ **Note:** Replaced in most cases by **target tracking** or **step scaling**

**4. Scheduled Scaling**

* Launch or terminate instances based on **date/time**
* Good for predictable usage patterns

**Example:** Scale up to 5 instances at 8:00 AM, scale down to 2 at 8:00 PM

✅ **Best for:** Predictable workloads like business hours

**5. Predictive Scaling (Optional/Advanced)**

* Uses **machine learning** to forecast traffic and automatically plan scaling
* Analyzes historical data and trends

✅ **Best for:** Applications with **regular, repeating traffic patterns**

**🌩️ AWS CloudTrail – Explained Simply**

**AWS CloudTrail** is a **logging and monitoring service** that records **every action taken in your AWS account** — whether by a user, an app, or a service.

Think of it as your **AWS activity CCTV camera** 📹 — it captures **who did what, when, and from where**.

**✅ Key Features of CloudTrail**

| **Feature** | **Description** |
| --- | --- |
| 📝 **Event Logging** | Records all API calls (management & data events) made via AWS Console, CLI, SDKs, and services |
| 🔍 **Audit & Compliance** | Helps meet security, audit, and compliance needs (e.g., who deleted a resource?) |
| 🔒 **Security Analysis** | Detects unauthorized activity, investigations during security incidents |
| 📂 **Stores in S3** | Trail logs are stored in your S3 bucket |
| 📈 **Integration with CloudWatch** | You can create alarms when specific actions are detected |

**🧩 What Does CloudTrail Log?**

* **Who** made the request (IAM user, role, service)
* **What** action was taken (e.g., ec2:TerminateInstances)
* **Which resource** was affected (like an EC2 instance ID)
* **When** the action occurred
* **Where** it came from (IP address, region)

**🔄 Types of Events Logged**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Management Events** | Control plane operations (create, delete, modify resources) | RunInstances, CreateBucket |
| **Data Events** | Data-level access (read/write to objects) | GetObject, PutObject |
| **Insights Events** | Detect unusual activity or API spikes | Sudden spikes in StopInstances calls |

**🛠️ How to Set Up CloudTrail**

1. Go to **CloudTrail** in AWS Console
2. Click **“Create trail”**
3. Choose:
   * **Trail name**
   * **Apply to all regions** ✅ (recommended)
   * **S3 bucket** for storing logs
   * **Log type** (management/data/insights)
4. Enable **CloudWatch Logs** integration (optional but useful)

**💡 Common Use Cases**

* Who **terminated an EC2 instance**?
* Which IP address made **unauthorized API calls**?
* Audit: Track **user activity over the last 90 days**
* Compliance: Proof of activity for **ISO, PCI-DSS, HIPAA**

**🧾 Example Log Snippet (JSON)**

json

CopyEdit

{

"eventName": "StartInstances",

"userIdentity": {

"userName": "admin-user"

},

"sourceIPAddress": "203.0.113.0",

"eventTime": "2025-06-05T08:23:45Z",

"requestParameters": {

"instancesSet": {

"items": ["i-0abcd1234efgh5678"]

}

}

}

**🧠 Quick Facts**

* CloudTrail is **enabled by default** (basic logging for 90 days)
* To store long-term or across regions, you need to create a **trail**
* One trail can be used to log **all regions**
* Supports integration with **AWS Athena**, **CloudWatch**, **Lambda**, **SIEM tools**

**📊 AWS CloudWatch – Explained Simply**

**Amazon CloudWatch** is a **monitoring and observability service** for AWS.  
It lets you **collect, view, and analyze** data (metrics, logs, events, and alarms) from your AWS resources and applications.

**🛡️ Read Replica in AWS (Amazon RDS) – Explained Simply**

A **Read Replica** in AWS is a **copy of your database** that is **read-only**, used to:

* **Improve performance** (by offloading read traffic)
* **Scale out read-heavy workloads**
* **Provide backup for disaster recovery**

**🌀 AWS Lambda – Explained Simply**

**AWS Lambda** is a **serverless compute service** that lets you **run code without managing servers**.  
You just write your function, and AWS takes care of **running, scaling, and billing** based on usage.

💡 Think of it as: "Upload your code, set the trigger, and you're done!"

**✅ Key Features of AWS Lambda**

| **Feature** | **Description** |
| --- | --- |
| 🧠 **Serverless** | No need to provision or manage servers |
| ⏱️ **Event-driven** | Runs in response to triggers (like S3 upload, API Gateway call, or a schedule) |
| 💸 **Pay-as-you-go** | You’re charged **only when your code runs** |
| ⚙️ **Auto-scaling** | Automatically scales based on number of requests |
| 🧩 **Flexible language support** | Supports Python, Node.js, Java, Go, .NET, Ruby, and custom runtimes |

**🧩 How AWS Lambda Works**

1. You write a **function** (the code to be executed)
2. Set a **trigger** (event source), such as:
   * S3 (e.g., run code when a file is uploaded)
   * API Gateway (HTTP endpoint)
   * DynamoDB (on data changes)
   * CloudWatch Events (scheduled job)
3. Lambda automatically **executes** the function when the event occurs
4. AWS manages the compute and resources behind the scenes

**🧱 AWS CloudFormation – Explained Simply**

**AWS CloudFormation** is a **service that helps you create and manage AWS resources using code** — instead of clicking around in the AWS Console.

🛠️ Think of it like “Infrastructure as Code (IaC)” for your entire AWS environment.

**✅ Why Use CloudFormation?**

| **Benefit** | **Description** |
| --- | --- |
| 🧾 **Infrastructure as Code (IaC)** | Define your entire AWS setup in a YAML or JSON file |
| 🔁 **Repeatable Deployments** | Launch identical environments consistently across dev, staging, and prod |
| ⚙️ **Automation** | Automatically creates, updates, and deletes resources |
| 🧩 **Dependency Management** | Understands resource order (e.g., VPC before EC2) |
| 🛡️ **Safe Rollbacks** | If something fails, it can automatically rollback to previous state |

**🧩 How It Works**

1. You write a **CloudFormation Template** (YAML or JSON)
2. Upload to CloudFormation and launch a **Stack**
3. CloudFormation reads the template and **provisions resources**
4. You can **update or delete** the stack any time — CloudFormation handles all changes

**🚀 What You Can Deploy with CloudFormation**

* **EC2**, **S3**, **RDS**, **Lambda**, **VPC**, **IAM**, **Auto Scaling**, etc.
* You can **version control** templates using Git
* Works with **AWS CodePipeline**, **Terraform**, **Ansible**, etc.

**🛫 AWS Migration – Explained Simply**

**AWS Migration** means **moving your existing data, applications, or entire infrastructure** from on-premises (or another cloud) **to AWS**.

Think of it like **shifting your IT home into the cloud** — more scalable, secure, and cost-effective.