**<https://github.com/techworldwithmurali/aws-zero-to-hero>**

**Linux commands.**

**Services Names:**

* **S3: k**
* **EC2: k**
* **VPC: k**
* **Load Balancer: k**
* **Auto scaling.**
* **Cloud watch**
* **VAP**
* **cloud factory.**
* **Cloud trail**
* **SNS.**
* User**: svsrikanth07**
* Pwd**: 7780@Aws07**
* Page | 1SKIP: **11,12,13,14,15,20,21,22,24**

**Step 1 – Foundation (1–2 Weeks)**

**Goal: Understand AWS core services and cloud basics.**

* **Learn:**
  + **EC2 (compute)**
  + **S3 (storage)**
  + **RDS (databases)**
  + **IAM (security)**
  + **VPC basics (networking)**
* **Resources:**
  + ***AWS Cloud Practitioner Essentials* (free AWS training)**
  + **FreeCodeCamp’s AWS beginner course on YouTube (8 hours)**
* **Hands-On: Create a free AWS account, deploy a simple website using EC2 + S3.**

**Step 2 – Core AWS Skills (3–4 Weeks)**

**Goal: Build skills for real backend/cloud projects.**

* **Learn Services:**
  + **Lambda (serverless)**
  + **API Gateway**
  + **DynamoDB & RDS**
  + **CloudFormation (Infrastructure as Code)**
  + **CloudWatch & CloudTrail (monitoring)**
* **Hands-On Projects:**
  + **Deploy a Spring Boot app to AWS EC2.**
  + **Use S3 + CloudFront for static file hosting.**
  + **Create a serverless API with Lambda + API Gateway + DynamoDB.**

# ✅ What Is Cloud?

* Accessing storage, servers, and software over the internet to store files, run apps, and use powerful servers remotely is called cloud.

**Key Concepts:**

* **Remote Servers:** Files and apps are stored on servers by providers like AWS or Google, not on your device.
* **On-Demand Access:** Use cloud services anytime, anywhere via the internet.
* **Pay-as-You-Go:** Only pay for the resources you use, without buying expensive hardware.
* **Scalability:** Easily increase or decrease storage and computing power as needed.

# Types of Cloud Computing:

* Public Cloud – Services are delivered over the public internet and shared multiple organizations (e.g., AWS, Azure).
* Private Cloud – Services are maintained on a private network for a single organization.
* Hybrid Cloud – A mix of public and private cloud, allowing data and applications to move between them.

# Common Cloud Services:

* **IaaS (Infrastructure as a Service):** Provides virtual machines, storage, and networking (e.g., AWS EC2).
* **PaaS (Platform as a Service):** Platforms to build and deploy apps without managing infrastructure (e.g., Google App Engine).
* **SaaS (Software as a Service):** Fully developed applications accessible online (e.g., Gmail, Microsoft 365).

**Why cloud?**

Cloud computing means using the internet to get data, apps, and services — instead of keeping them on your own computer.

* **Flexible** (use it from anywhere)
* **Cost Savings** (pay only for what you use)
* **Scalable** (easily grow or shrink resources)

**Difference between public cloud and private cloud**

* **Public Cloud**: Shared by many users, provided by companies like AWS or Google, accessible over the internet.
* **Private Cloud**: Used by a single company, more secure and controlled, hosted on-premises or by a dedicated provider.

# ✅ Why AWS?

* AWS is special because of its early start, unmatched service range, global infrastructure, and strong enterprise trust.
* Let me know if you want a side-by-side comparison with Azure or GCP.

**Why Some Companies Move Back to Private Cloud**

* Some businesses switch back to private cloud to get:
  + More control
  + Better security
  + More predictable costs
* Public cloud can be costly or less flexible for some needs.
* Many now use a hybrid cloud — a mix of public and private — to get the best of both.

# What is IAM ?

* **IAM** stands for **Identity and Access Management**. It is a web service that helps you **securely control access** to services and resources.

# Why IAM?

* AWS IAM allows you to securely manage access to AWS services and resources. It helps control **who** can access **what**, and **what actions** they can perform.

# Root User vs IAM User

* **Root User**:
  + Created when the AWS account is made Logs in with **email**
  + Has **full access** to everything
  + Used for **important tasks** only
  + **Can’t be deleted**
* **IAM User**:
  + Created by root or admin, Log in with **username**
  + Has **limited access** based on rules
  + Used for **daily work**
  + **Can be added or removed**

# ✅ Key Concepts of IAM:

# Users:

* + Individuals or applications that interact with AWS services.
  + Each user gets credentials (like passwords or access keys).

# Groups:

* + A collection of IAM users.
  + Permissions assigned to a group are inherited by all users in that group.

# Roles:

* + Similar to users, but **not associated with a specific person**.
  + Used to grant **temporary access** to AWS services for users, applications, or services. (roles are creating communicate to One AWS to Other AWS).
  + Commonly used in cross-account access, EC2 instance access, or Lambda functions.

# Policies:

* Rules written in JSON.
* They say **who can do what** in AWS.
* Can be attached to users, groups, or roles.

# Permissions:

* Define **what actions** someone can do (like read from S3 or start an EC2).

# Principals:

* Anyone who can make a request — like a user, a role, or even an app.

# ✅ What is EC2 and Why Is It Important?

Amazon EC2 is like renting virtual computers in the cloud.

* Easily scale up or down based on demand
* 99.99% availability and strong built-in security (via AWS Nitro System)
* Run applications without managing physical hardware.
* Flexible, cost-saving options like Graviton, Spot Instances, and Savings Plans

**EC2 Use Cases**

1. **Run Business Applications**: Run secure, reliable with high performance.
2. **High-Performance Computing (HPC)**: Handle heavy tasks like data analysis (HPC)
3. **Quick Setup & Auto Scaling**: Quickly launch and scale servers as needed.
4. **Machine Learning Workloads**: Power machine learning with high-speed compute and storage

**EC2 Instance Types**

Amazon EC2 offers different instance types to fit various workloads. Each type is optimized for specific use cases:

1. **General Purpose**  
   Balanced for compute, memory, and networking (e.g., **t3**, **m6g**)  
   *Use case: web servers, small databases*
2. **Compute Optimized**  
   High-performance processors for compute-heavy tasks (e.g., **c7g**)  
   *Use case: batch processing, high-performance web apps*
3. **Memory Optimized**  
   More RAM for memory-intensive applications (e.g., **r6g**, **x2idn**)  
   *Use case: in-memory databases, real-time big data*
4. **Storage Optimized**  
   High disk throughput for fast local storage (e.g., **i4i**, **im4gn**)  
   *Use case: data warehousing, large transactional systems*
5. **Accelerated Computing**  
   Uses GPUs or FPGAs for specialized workloads (e.g., **p4**, **inf2**)  
   *Use case: machine learning, video processing, scientific computing*

**EC2 Instance Basics:**

* **What is an EC2 Instance?**  
  A virtual server in AWS used to run applications.
* **Customizable Resources:**  
  Choose CPU, memory, storage, and networking based on your needs.
* **Instance Types:**  
  Pick from general purpose, compute, memory, storage, or GPU-accelerated types.
* **Pay-as-You-Go:**  
  Pay only for the compute time you use.
* **Scalable:**  
  Easily increase or decrease instance capacity as your workload changes.

**Launching an EC2 Instance:**

1. **Choose AMI (Amazon Machine Image):**  
   Select the OS and pre-installed software (e.g., Ubuntu, Amazon Linux).
2. **Select Instance Type:**  
   Pick based on CPU, memory, and performance needs (e.g., t3.micro).
3. **Configure Instance:**  
   Set network, IAM roles, shutdown behavior, etc.
4. **Add Storage:**  
   Choose the type and size of disk (e.g., SSD or HDD).
5. **Add Tags (Optional):**  
   Label your instance for easy identification.
6. **Configure Security Group:**  
   Set firewall rules (e.g., allow SSH, HTTP).
7. **Launch and Connect:**  
   Review settings, choose a key pair, and launch.  
   Then, connect via SSH or EC2 Connect.

**Managing EC2 Instances:**

1. **Start / Stop / Reboot:**  
   Control the instance lifecycle as needed without data loss (for EBS-backed instances).
2. **Connect to Instance:**  
   Use SSH (Linux) or RDP (Windows) to access and manage your instance.
3. **Monitor Performance:**  
   Use **Amazon CloudWatch** to track CPU, memory, disk, and network usage.
4. **Resize (Change Instance Type):**  
   Upgrade or downgrade instance type based on performance needs.
5. **Attach / Detach Volumes:**  
   Manage storage by adding or removing EBS volumes.
6. **Security Management:**  
   Update security groups, key pairs, and IAM roles as needed.
7. **Terminate:**  
   Delete the instance when it’s no longer needed to stop billing.

Pending topics:

 **Elastic Block Store (EBS) – storage for EC2**

 **Networking Basics (Elastic IP, Security Groups, VPC)**

 **Security for EC2 (IAM roles, key pairs, access management)**

 **Auto Scaling with EC2**

 **Placement Groups and High Availability**

 **Spot Instances, Reserved Instances, and Pricing Models**

 **Monitoring & Logging (CloudWatch, CloudTrail)**

 **Advanced Features (Dedicated Hosts, Burstable Instances, Nitro System)**

 **Automation & Infrastructure as Code (CloudFormation, Terraform, EC2 Image Builder)**

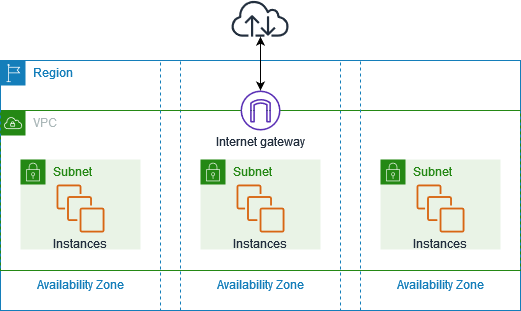
# ✅ What is VPC? (Simple Explanation)

VPC is like your own private area in the cloud where you can safely run apps and keep your data, separated from everyone else.

**🛠️ Key Points (Easy to Remember):**

* + You choose the **IP range** and create **subnets** to organize your resources.
  + You can connect to the **internet** using **gateways or routers**.
  + You control **who can access** what, using **security groups** and **firewalls**.

**In short:**A VPC is like your own secure neighbourhood in the cloud — you build it, control it, and protect it.



By default, when you create an AWS account, AWS will create a default VPC for you but this default VPC is just to get started with AWS. You should create VPCs for applications or projects.

The following features help you configure a VPC to provide the connectivity that your applications need:

**Virtual private clouds (VPC)**

A VPC is a virtual network that closely resembles a traditional network that you'd operate in your own data centre. After you create a VPC, you can add subnets.

**Subnets:**

* A subnet is a range of IP addresses in your VPC. A subnet must reside in a single Availability Zone. After you add subnets, you can deploy AWS resources in your VPC.

**IP addressing:**

* You can assign IP addresses, both IPv4 and IPv6, to your VPCs and subnets. You can also bring your public IPv4 and IPv6 GUA addresses to AWS and allocate them to resources in your VPC, such as EC2 instances, NAT gateways, and Network Load Balancers.

**Network Access Control List (NACL)**

* A Network Access Control List is a stateless firewall that controls inbound and outbound traffic at the subnet level. It operates at the IP address level and can allow or deny traffic based on rules that you define. NACLs provide an additional layer of network security for your VPC.

**Security Group**

* A security group acts as a virtual firewall for instances (EC2 instances or other resources) within a VPC. It controls inbound and outbound traffic at the instance level. Security groups allow you to define rules that permit or restrict traffic based on protocols, ports, and IP addresses.

**Routing**

* Use route tables to determine where network traffic from your subnet or gateway is directed.

**Gateways and endpoints**

* A gateway connects your VPC to another network. For example, use an internet gateway to connect your VPC to the internet. Use a VPC endpoint to connect to AWS services privately, without the use of an internet gateway or NAT device.

**Peering connections**

* Use a VPC peering connection to route traffic between the resources in two VPCs.

**Traffic Mirroring**

* Makes a **copy of network traffic** for monitoring and security analysis. for deep packet inspection.

**Transit gateways**

* Use a transit gateway, which acts as a central hub, to route traffic between your VPCs, VPN connections, and AWS Direct Connect connections.

**VPC Flow Logs**

* A flow log captures information about the IP traffic going to and from network interfaces in your VPC.

**VPN connections**

* Connect your VPCs to your on-premises networks using AWS Virtual Private Network (AWS VPN).

**Key Features of a VPC:**

1. **Subnetting**: You can divide your VPC into subnets (public or private).
2. **IP Addressing**: Choose your own IP address range using **CIDR blocks**.
3. **Routing**: Customize route tables to control traffic between subnets.
4. **Security**:
   * **Security Groups**: Act like virtual firewalls for EC2 instances.
   * **Network ACLs**: Control inbound and outbound traffic at the subnet level.
5. **Internet Gateway (IGW)**: Allows communication between resources in your VPC and the internet.
6. **NAT Gateway/Instance**: Lets instances in a private subnet access the internet without exposing them to inbound internet traffic.
7. **VPC Peering**: Connects two VPCs for private communication.
8. **VPN and AWS Direct Connect**: Enables secure communication between your on-premises data centre and AWS.

# Why Use a VPC?

* Complete control over your network.
* Secure and scalable setup.
* Isolated from other cloud users.
* Needed to run services like EC2, RDS, and ECS.

AWS (Amazon Web Services) provides multiple layers of security to protect resources and cloud infrastructure. These are two types

**Key Difference:**

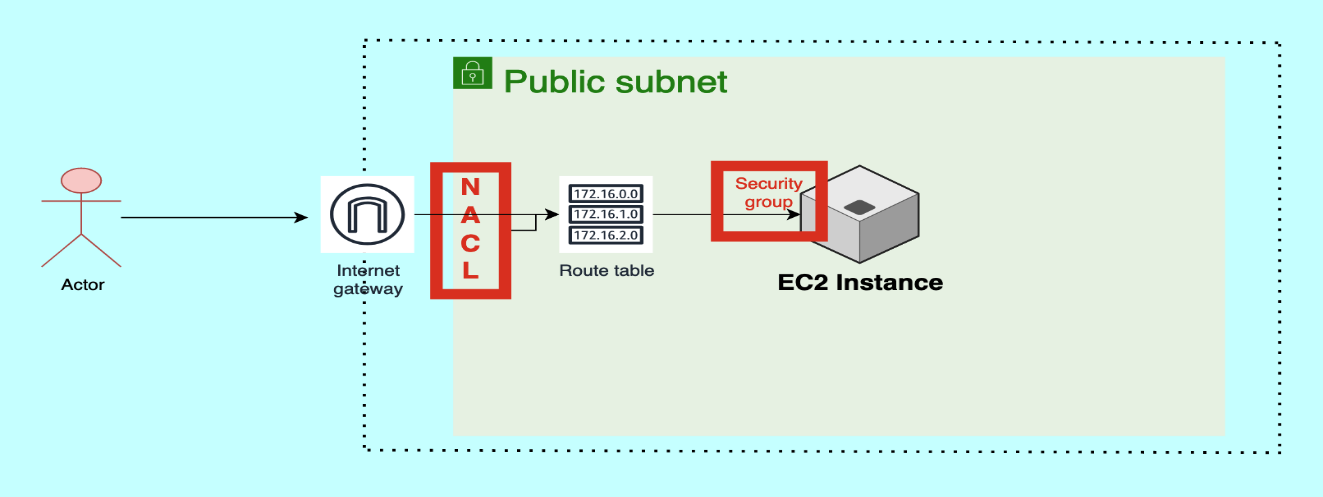
* **Security Group** = Works like a firewall for each instance (only "allow" rules). protecting **EC2 instances.**
* **NACL** = Works like a firewall for the subnet (has both "allow" and "deny" rules). securing traffic at the **subnet level.**

# ✅ Security Group:

1. **Security Groups** are like safety gates for EC2 instances.
2. They decide **who can come in** (inbound) and **who can go out** (outbound).
3. You can add one or more security groups to each EC2 instance.
4. Rules allow traffic based on things like **IP, port, or protocol**.
5. They work **directly on the instance**, checking rules before letting traffic in.
6. They are **stateful**, so if you allow incoming traffic, the reply goes out automatically.
7. **Changes apply instantly** — no need to restart anything.

# ✅ NACL: (Network Access Control Lists)

1. NACLs work like a **security filter** for an entire **subnet**, not individual instances.
2. They **control traffic in and out** of the subnet.
3. NACLs are **stateless** – you must create separate rules for **inbound and outbound** traffic.
4. Each **subnet can have only one NACL**, but one NACL can be used by **many subnets**.
5. NACLs use a **numbered list of rules**, and AWS checks them **in order** (from smallest number to largest).
6. Each rule defines things like **allow or deny**, IP range, port, and protocol.
7. You can **allow or block** traffic based on those rules.
8. Because they are stateless, you must **explicitly allow both directions** (e.g., in and out).
9. **Changes to NACLs** may take a little time to apply across your resources.



**1. The request enters via the Internet Gateway**

* The **Internet Gateway (IGW)** connects your **VPC to the internet**.
* It allows traffic from the outside world (like users or applications) to **enter your AWS environment**.

**✅ 2. It is checked by the NACL**

* Before entering the subnet, the request hits the **Network Access Control List (NACL)**.
* NACL checks whether the traffic is **allowed or denied** based on IP, port, and protocol.
* Since NACL is **stateless**, both inbound and outbound rules must be configured.

**✅ 3. Routed through the Route Table**

* If allowed by the NACL, the request is passed to the **Route Table**.
* The Route Table decides **where to send the traffic** inside the VPC (i.e., to which subnet or instance).

**✅ 4. Further filtered by the Security Group**

* Now, the request reaches the **Security Group** attached to the **EC2 instance**.
* The Security Group checks if the request is **allowed to reach the instance**, based on rules like allowed ports (e.g., 22 for SSH or 80 for HTTP).

**✅ 5. Finally reaches the EC2 instance if allowed**

* If the request passes all layers — **Internet Gateway → NACL → Route Table → Security Group** — it is finally delivered to the **EC2 instance**.
* If any layer blocks the request, it won’t reach the instance.

# What is NAT Gateway?

* Your EC2 in a **private subnet** sends traffic to the NAT Gateway.
* NAT Gateway changes the *source private IP* to its *own public IP*.
* NAT Gateway sends it to the **Internet Gateway**.
* Response comes back to the NAT Gateway → translated back to your EC2’s private IP → sent to the instance

**Example:** Think of the NAT Gateway as the office receptionist — employees (private subnet EC2s) can ask the receptionist to send out letters (outbound requests), but strangers can’t directly walk into the office from outside.

# ✅ S3(Simple Storage Service)

**What is S3?**

* S3 (**Simple Storage Service**) in AWS that lets you **store, manage, and retrieve unlimited data** from anywhere over the internet.
* It’s commonly used **for storing files, backups, images, videos, logs, static websites, and big data.**

**What are S3 buckets?**

* S3 buckets are **AWS containers** for **storing and organizing files**.
* Each bucket has a **unique name** in AWS

**Uses:** S3 buckets are used to **securely store**, and **access data** in the cloud.

# ✅ Key Features of Amazon S3

**a) Scalability:** S3 buckets automatically scale to store unlimited data. With out manual configuration.

**b) High Availability & Durability:** S3 stores data across multiple zones with 99.999999999% durability.

**c) Security:** S3 provides access control via IAM, policies, and ACLs, with encryption for data at rest and in transit.

**d) Cost-Effective: Pay-as-you-go** pricing model — you only pay for what you store.

**e) Data Management**

* **Versioning** → Keeps multiple versions of an object.
* **Lifecycle policies** → Move objects between storage classes automatically.
* **Replication** → Sync objects between different AWS regions.

# How S3 Works (Concepts)

* **Bucket** → A container for storing objects (similar to a folder).
* **Object** → A single file stored in S3.
* **Key** → The unique identifier of an object inside a bucket.
* **Region** → You choose where your Bucket’s physical location.
* **URL Access** → Files can be made **public** or **private** and accessed via URL.

**Example S3 Object URL:**

<https://mybucket.s3.ap-south-1.amazonaws.com/photo.jpg>

**Common Use Cases**

* Store static website files
* Backup and disaster recovery
* Big data storage
* Media hosting (images, videos, audio)
* Application log storage
* Machine learning datasets

# Steps to Create an S3 Bucket

1. **Go to S3 in AWS Console** → Click **Create bucket**.
2. **Bucket Name** → Must be **unique globally**. Example: my-app-data-2025.
3. **Region** → Choose where data will be stored (e.g., Asia Pacific (Mumbai) - ap-south-1).
4. **Options** (optional):
   * **Versioning** → Keep multiple versions of files.
   * **Encryption** → Secure files automatically.
   * **Tags** → Add labels like Project=Ecommerce.
5. **Access Settings** → Buckets are private by default (you can allow public access if hosting a website).
6. Click **Create bucket** → Bucket is ready to use.

# S3 Integration with Other AWS Services

* **EC2** → Store backups, logs, images.
* **CloudFront** → Deliver content via CDN.
* **Lambda** → Trigger functions when new objects are uploaded.
* **Athena** → Query S3 data directly using SQL.
* **Glue** → ETL jobs on S3 data.

# ✅ AWS CLI?

# What is AWS CLI?

* The **AWS CLI** is a **command-line tool** that is **manage and automate AWS services** directly from the terminal using commands, instead of using the AWS Management Console.

# Key Points about AWS CLI

* **Definition** → Open-source tool to manage AWS services via commands.
* **Purpose** → Automates tasks, manages resources, and simplifies operations.
* **Cross-Platform** → Supports Windows, macOS, and Linux.
* **Versions** → **v1** (older) | **v2** (latest, recommended).

# Advantages of AWS CLI

* **Faster** than AWS Console for repetitive tasks
* **Automates deployments** using scripts
* **Integrates with CI/CD pipelines** (Jenkins, GitHub Actions, etc.)
* **Easier bulk operations** (e.g., managing 100+ EC2 instances)

# Where AWS CLI Is Used

* **DevOps & Cloud Engineers** → Automate deployments
* **Developers** → Manage app infrastructure
* **QA Teams** → Manage test environments
* **System Administrators** → Manage servers, backups, monitoring

# Why Use AWS CLI 🛠️

* **Faster Operations** → Execute tasks quicker than using the AWS Management Console.
* **Automation** → Easily automate repetitive tasks using **shell** or **batch scripts**.
* **Bulk Management** → Manage multiple AWS resources (e.g., EC2, S3, IAM) at once.
* **Integration** → Works seamlessly with **CI/CD pipelines** like Jenkins, GitHub Actions, etc.
* **Cross-Platform** → Available on Windows, macOS, and Linux.
* **Cost & Time Efficiency** → Reduces manual effort and saves time.

# Basic AWS CLI Workflow

1. **Install AWS CLI** → Download & verify: aws --version
2. **Configure AWS CLI** → Set credentials: aws configure

Requires **Access Key**, **Secret Key**, **Region**, and **Output Format**.

1. **Use Commands** → Examples:
   * List S3 buckets → aws s3 ls
   * Upload file → aws s3 cp myfile.txt s3://mybucket/
   * List EC2 instances → aws ec2 describe-instances
   * Start EC2 → aws ec2 start-instances --instance-ids <id>

**AWS installation guide:** <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html>

**NOTE:**

* **AWS CLI** → Best for **manual commands** and **quick resource management**.
* **Terraform** → Best for **multi-cloud IaC** and **complex infrastructure** automation.
* **CloudFormation** → Best for **AWS-only IaC** with **native integration**.

# ✅ Cloud Watch

# What is cloud watch?

* **CloudWatch** is AWS’s monitoring and logging service. It is used to tracks performance, collects logs, sets alarms, automates responses.
* **Integrate** with services like EC2, Lambda, RDS, S3, and Auto Scaling.

**Common Use Cases**

* **Monitor AWS services** → EC2, RDS, S3, Lambda, DynamoDB, etc.
* **Track application health** → Error rates, response times.
* **Set up alerts** → Get notifications via **SNS** or **email**.
* **Automate scaling** → Integrated with **Auto Scaling Groups**.
* **Debug issues** → Analyze logs from apps and services. **Basic CloudWatch CLI Commands**

**1. List Metrics**

aws cloudwatch list-metrics

**2. Get Metric Statistics**

aws cloudwatch get-metric-statistics \

--metric-name CPUUtilization \

--start-time 2025-08-28T00:00:00Z \

--end-time 2025-08-28T23:59:59Z \

--period 300 \

--namespace AWS/EC2 \

--statistics Average

**3. Create an Alarm**

aws cloudwatch put-metric-alarm \

--alarm-name HighCPUUsage \

--metric-name CPUUtilization \

--namespace AWS/EC2 \

--statistic Average \

--period 300 \

--threshold 80 \

--comparison-operator GreaterThanThreshold \

--dimensions Name=InstanceId,Value=i-1234567890abcdef0 \

--evaluation-periods 2 \

--alarm-actions arn:aws:sns:ap-south-1:123456789012:MyTopic

**06/09/2025**

Core AWS Services

* Compute:
  + EC2 → Virtual servers
  + Lambda → Serverless computing
  + ECS / EKS → Container orchestration
* Storage:
  + S3 → Object storage
  + EBS → Block storage
  + Glacier → Archival storage
* Databases:
  + RDS → Managed relational database
  + DynamoDB → NoSQL database
  + Redshift → Data warehouse
* Networking:
  + VPC → Private network
  + Route 53 → DNS and domain routing
  + CloudFront → Content delivery

# ✅ What is a Load Balancer in AWS?

**Load Balancer** distributes **incoming traffic** to multiple servers (EC2 instances).

* No single server gets **overloaded** ✅
* If one server **fails**, traffic goes to other healthy servers ✅
* Your app stays **fast, reliable, and always available** ✅

AWS provides this service through **Elastic Load Balancing (ELB)**.

**Simple Example**

Imagine you have a **Spring Boot application** deployed on **3 EC2 instances**:

* **Server 1** → 10.0.0.1
* **Server 2** → 10.0.0.2
* **Server 3** → 10.0.0.3

**Without a Load Balancer ❌**

* Users have to connect **directly** to each server using its IP.
* If **Server 1** fails → Users connected to it face **downtime**.
* Some servers get **overloaded** while others stay idle.

**With a Load Balancer ✅**

* Users connect to **one URL**: https://myapp.example.com
* The Load Balancer **automatically distributes traffic**:
  + 1st request → Server 1
  + 2nd request → Server 2
  + 3rd request → Server 3
* If Server 2 goes down → LB **stops sending traffic** to it.
* Users never know any server failed → **Zero downtime**.

**1. Vertical Scaling (Scale Up/Down) ⬆️⬇️**

Adding **more power** (CPU, RAM, storage) to a **single server**.

**How it works**

* You keep the **same server** but make it **stronger**.
* Increase CPU, RAM, or storage to handle more load.
* There’s **a limit** to how much you can scale up.

**Example**

You have one EC2 instance with **2 CPUs and 4GB RAM**:

* If traffic increases → Upgrade to **8 CPUs and 16GB RAM**.
* Same server, but **more powerful**.

**2. Horizontal Scaling (Scale Out/In) ↔️**

Adding **more servers** to share the load.

**How it works**

* Instead of upgrading one server, you **add multiple servers**.
* Traffic is **distributed** using a **Load Balancer**.
* Can scale **almost infinitely**.

**Example**

You start with **1 EC2 instance**:

* When traffic increases → Add **2 more EC2 instances**.
* A **Load Balancer** distributes requests among all servers.

# Types of AWS Load Balancers

**1. Application Load Balancer (ALB) ✅ (Most Common)**

* Best for **web apps** and **APIs**.
* Works with **HTTP** and **HTTPS**.
* Can **route traffic smartly** based on:
  + URL path → /login → Auth Service
  + Domain name → api.example.com → API Service
* Perfect for **Spring Boot microservices**.

**Example:**  
https://api.example.com/auth → Auth microservice  
https://api.example.com/payment → Payment microservice

# Steps to Create a Load Balancer in AWS

**Step 1: Login to AWS Console**

* Go to https://aws.amazon.com/console/
* Sign in with your **AWS account**.

**Step 2: Open EC2 Dashboard**

* Go to **Services** → **EC2**.
* On the **left menu**, click **Load Balancers**.
* Click **Create Load Balancer**.

**Step 3: Choose Load Balancer Type**

AWS offers 3 types:

* **Application Load Balancer (ALB)** → Best for HTTP/HTTPS traffic ✅ *(Recommended)*
* **Network Load Balancer (NLB)** → Best for TCP/UDP high-performance traffic.
* **Gateway Load Balancer (GWLB)** → Used for security appliances.

For a **Spring Boot app** or **web app**, **choose Application Load Balancer** → **Create**.

**Step 4: Configure Basic Settings**

* **Name** → e.g. my-app-alb
* **Scheme** → Choose:
  + **Internet-facing** → If your app is public (recommended)
  + **Internal** → For private apps
* **IP Address Type** → Keep default (**IPv4**)

**Step 5: Configure Network & Subnets**

* Select the **VPC** where your EC2 instances are running.
* Choose **at least two subnets** in **different Availability Zones** for **high availability**.

**Step 6: Configure Security Settings**

* If you want **HTTPS**, select an **SSL certificate** from **AWS Certificate Manager (ACM)**.
* For **HTTP only**, you can skip this.

**Step 7: Configure Security Groups**

* Create or select a **Security Group** that allows:
  + **HTTP (Port 80)**
  + **HTTPS (Port 443)** *(optional)*
  + Your **Spring Boot port** if needed, e.g., **8080**

**Step 8: Configure Target Groups**

* A **Target Group** is a group of servers (EC2 instances, containers, etc.) that will receive traffic.
* Click **Create a new target group**:
  + **Target Type** → Choose **Instance**
  + **Protocol** → **HTTP**
  + **Port** → The port your Spring Boot app runs on (e.g., **8080**)
* Click **Next** and select your EC2 instances.
* Click **Register** → Add selected instances.

**Step 9: Configure Health Checks**

* By default, health checks use / path.
* If your Spring Boot app has a custom **health endpoint** (e.g., /actuator/health), update it here.
* Healthy instances will get traffic; unhealthy ones are skipped.

**Step 10: Review and Create**

* Review all settings carefully.
* Click **Create Load Balancer**.
* Wait for the status to become **Active** (usually 1–2 minutes).

**Step 11: Access Your Application**

* Once created, go to the **Description** tab of the load balancer.
* Copy the **DNS name** → Something like:
* my-app-alb-123456789.ap-south-1.elb.amazonaws.com
* Open it in your browser → It will distribute traffic across all registered EC2 instances.

**How It Works**

User → AWS ALB → Target Group → EC2 Instances → Spring Boot App

**Best Practices**

* Always create **2+ EC2 instances** in **different Availability Zones**.
* Enable **Cross-Zone Load Balancing** for even traffic.
* Use **HTTPS** with **AWS Certificate Manager** for security.
* Enable **Access Logs** for monitoring.
* Integrate with **Auto Scaling Groups** for automatic scaling.

**2. Network Load Balancer (NLB) ⚡**

* Best for **high-performance** apps.
* Works with **TCP, UDP, TLS**.
* Can handle **millions of requests per second**.
* Used for **real-time apps**, like:
  + Gaming 🎮
  + Video streaming 📺
  + Financial trading 💰

# Steps to Create NLB in AWS (Simple) ⚡

**Step 1: Go to EC2 → Load Balancers**

* Login to AWS Console → **EC2 Dashboard** → **Load Balancers** → **Create Load Balancer**.
* Select **Network Load Balancer (NLB)** → **Create**.

**Step 2: Configure Basic Settings**

* **Name** → my-nlb
* **Scheme** → **Internet-facing** *(for public apps)*
* **IP Type** → **IPv4** *(default)*
* Choose your **VPC** and **at least 2 subnets** for high availability.

**Step 3: Add Listener**

* **Protocol** → **TCP**
* **Port** → Your app port (e.g., **8080** for Spring Boot).

**Step 4: Create Target Group**

* **Target Type** → **Instance**
* **Protocol** → **TCP**
* **Port** → **8080**
* **Health Check** → /actuator/health *(if Spring Boot Actuator enabled)*.

**Step 5: Register Targets**

* Select your **EC2 instances** → **Add to registered** → **Next**.

**Step 6: Review & Create**

* Review settings → **Create Load Balancer**.
* Wait until status = **Active** ✅.

**Step 7: Access Application**

* Go to the **NLB Description tab**.
* Copy the **DNS name** → Example:
* my-nlb-123456.ap-south-1.elb.amazonaws.com
* Use this DNS instead of EC2 IP.

**How It Works**

User → Network Load Balancer → Target Group → EC2 Instances → Application

**Best Practices**

* Deploy EC2 instances in **multiple Availability Zones**.
* Use **Cross-Zone Load Balancing** for better traffic distribution.
* Enable **Health Checks** to avoid sending traffic to unhealthy servers.
* Use **TLS termination** for secure communication.
* Enable **CloudWatch Monitoring** for performance tracking.

# ✅ Auto Scaling

# What is Auto Scaling in AWS?

Auto Scaling automatically adjusts the number of resources (like EC2 instances) based on demand.

**In short:**

* High traffic → adds resources
* Low traffic → removes resources

# Why Do We Need Auto Scaling?

If you have a **Spring Boot application** running on **3 EC2 instances**:

* **Normal hours** → 3 servers are enough.
* **Peak hours** → Traffic increases, 3 servers are **not enough**.
* **Night hours** → Traffic decreases, **3 servers aren’t needed**.

**NOTE:** Manually Adjusting the servers is slow and time taken process so to avoid this will use Auto Scaling. Auto Scaling automatically adds or removes servers based on demand.

# How Auto Scaling Works

Auto Scaling monitors your application and adjusts the number of servers based on defined rules.  
It adds servers when demand increases and removes them when demand drops.

Auto Scaling works by using these components:

**1. Launch Template / Launch Configuration**

* A **blueprint** for creating new EC2 instances.
* Defines:
  + AMI (Operating System & Application)
  + Instance type (e.g., t3.medium)
  + Security groups
  + Key pairs

# Scaling Policies

These decide **when to add or remove instances**:

**a) Target Tracking (✅ Recommended):** Keeps a target metric (e.g., CPU = 50%); scales automatically.

**b) Step Scaling:** Scales in steps based on thresholds (e.g., CPU > 70% → add 1, > 85% → add 2).

**c) Scheduled Scaling:** Scales at set times (e.g., 9 AM → 10 instances, 8 PM → 3 instances).

# CloudWatch Alarms

* AWS **CloudWatch** monitors metrics like **CPU**, **memory**, or **network traffic**.
* When a threshold is reached, it **triggers Auto Scaling**.

# What is Time-Based Scaling

**Time-based scaling** allows you to set up a schedule to **scale up** (add instances) or **scale down** (remove instances) **at specific times and dates**, regardless of traffic or load.

**Example Scenario**

Imagine you have a **Spring Boot application** running on **3 EC2 instances**:

* **Business Hours (9 AM – 8 PM)** → High traffic → Need **10 instances**.
* **Night Hours (8 PM – 9 AM)** → Low traffic → Only **3 instances** needed.

# ✅ Steps to Create an Auto Scaling Group in AWS

**Step 1 — Go to Auto Scaling**

* Open **AWS Management Console** → **EC2 Service**.
* In the left menu, click **Auto Scaling Groups**.
* Click **Create Auto Scaling Group**.

**Step 2 — Choose Launch Template**

* Select **Launch Template** or **Launch Configuration**.
* Pick the existing one or **create a new launch template**.
* Click **Next**.

**Step 3 — Configure Auto Scaling Group**

* Enter **Auto Scaling Group Name**.
* Select your **VPC**.
* Choose **Subnets** (preferably multiple for high availability).
* Click **Next**.

**Step 4 — Attach Load Balancer (Optional)**

* Choose:
  + **Attach to an existing load balancer** → Select ALB or Target Group.  
    *(Skip if not using a load balancer.)*
* Click **Next**.

**Step 5 — Configure Group Size**

* Set:
  + **Desired Capacity** → e.g., 3
  + **Minimum Capacity** → e.g., 2
  + **Maximum Capacity** → e.g., 6
* Click **Next**.

**Step 6 — Configure Scaling Policies**

* Select scaling type:
  + **Target Tracking Scaling** (recommended).
  + Or leave as **No scaling policies** for now.
* If target tracking → Choose **CPU Utilization** → Set **50%**.
* Click **Next**.

**Step 7 — Configure Notifications (Optional)**

* Choose an **SNS topic** if you want email alerts.
* Click **Next**.

**Step 8 — Add Tags (Optional)**

* Add **Name** and **Environment** tags if needed.
* Click **Next**.

**Step 9 — Review & Create**

* Review all settings.
* Click **Create Auto Scaling Group**.

**Step 10 — Verify**

* Go to **Auto Scaling Groups**.
* Check the **Instances** tab → Your EC2 instances should be launching automatically.
* If scaling policy is enabled, test by increasing load.

# ✅ Steps to Create Time-Based Auto Scaling (Scheduled Scaling)

**Step 1 — Open AWS Console**

* Go to **AWS Management Console** → **EC2**.
* From the left menu, select **Auto Scaling Groups**.

**Step 2 — Select Your Auto Scaling Group**

* Click on the **Auto Scaling Group** you want to configure.
* Go to the **Automatic Scaling** tab.

**Step 3 — Create a Scheduled Action**

* Click **Create Scheduled Action**.
* Enter:
  + **Name** → e.g., Morning-Scale-Up
  + **Start Time** → Date and time to trigger scaling.
  + **End Time** *(optional)* → If you want scaling to stop after a certain period.

**Step 4 — Set Scaling Details**

* Choose one of the options:
  1. **Set Desired Capacity** → Fixed number of instances.
  2. **Set Min and Max Capacity** → Change limits temporarily.
* Example:
  1. Start Time: **09:00 AM** → Set **Desired Capacity = 10** (scale up).
  2. Start Time: **08:00 PM** → Set **Desired Capacity = 3** (scale down).

**Step 5 — Save the Schedule**

* Click **Create**.
* The scheduled action is added to your **Auto Scaling Group**.

**Step 6 — Verify**

* Go to **Auto Scaling Groups → Scheduled Actions**.
* Check your configured schedules.
* Instances will automatically scale **at the specified times**.

# ✅ Amazon RDS (Relational Database Service)

# What is Amazon RDS?

* **Amazon RDS (Relational Database Service)** is a **managed AWS service** that helps you **easily set up, operate, and scale** relational databases in the cloud.
* AWS handles **installation, backups, patching, scaling, and high availability**, letting you focus on **application development**.

**Key Points**

* **RDS** = Relational Database Service
* Run relational databases easily on AWS
* Automates setup, backups, scaling, and maintenance
* **Highly scalable, secure, and cost-effective**
* Supports multiple database engines (MySQL, PostgreSQL, MariaDB, Oracle, SQL Server, Amazon Aurora).

# Difference between Amazon RDS and Amazon Aurora?

|  |  |  |
| --- | --- | --- |
| **Feature** | **Amazon RDS** | **Amazon Aurora** |
| **Type** | Managed relational DB | AWS-built high-performance DB |
| **Performance** | Standard | Up to **5x faster** than MySQL |
| **Cost** | Cheaper | Slightly expensive |
| **Scalability** | Manual scaling | Auto-scaling supported |
| **Best For** | General-purpose apps | High-performance apps |

# What is IAM database authentication in RDS?

* Instead of using **username & password**, you connect to RDS using **AWS IAM roles**.
* More secure and easier to manage.

# How do you scale an RDS instance? There are two ways:

1. **Vertical Scaling** → Increase instance size (e.g., db.t3.micro → db.m5.large).
2. **Horizontal Scaling** → Add **read replicas** to distribute traffic.

# How do backups work in RDS? RDS provides two types of backups:

* **Automated Backups** → Daily snapshots + transaction logs.
* **Manual Snapshots** → You create them; they don’t expire.

# How do you monitor RDS?

* **Amazon CloudWatch** → Monitors CPU, memory, storage, connections.
* **Enhanced Monitoring** → Deeper insights into performance.
* **Performance Insights** → SQL query-level monitoring.

# Can RDS be integrated with Lambda and Spring Boot? Yes ✅

* Create an **RDS instance**.
* Use **AWS Lambda** or **Spring Boot app** to connect via **JDBC URL**.
* Use RDS **endpoint, username, and password**.

# How do you migrate a database to RDS?

* Use **AWS Database Migration Service (DMS)**.
* Import via **S3 snapshots**.
* Use **mysqldump** or **pg\_dump** for MySQL/PostgreSQL.

# What are Read Replicas in RDS?

* **Read Replicas** are **read-only copies** of your database.
* They improve **performance** by handling **read-heavy workloads**.
* They are **asynchronous** copies of the primary DB.
* Commonly used in reporting, analytics, and scaling applications.

# Difference between Multi-AZ and Read Replicas?

|  |  |  |
| --- | --- | --- |
| **Feature** | **Multi-AZ** | **Read Replicas** |
| **Purpose** | High availability | Performance scaling |
| **Replication** | **Synchronous** | **Asynchronous** |
| **Failover** | Automatic | No automatic failover |
| **Use Case** | Disaster recovery | Handle heavy read traffic |

# Steps to create an RDS DB instance (AWS Console)

1. **Sign in** to the AWS Management Console.
2. **Open RDS**: Services → **RDS** → **Databases**.
3. Click **Create database**.
4. **Choose creation method**: **Standard create**.
5. **Choose engine**: pick one (MySQL / PostgreSQL / MariaDB / Oracle / SQL Server / Amazon Aurora).
6. **Template**: choose **Free tier** (for learning) or **Production** (for real workloads).
7. **DB instance details**:

* **DB instance identifier**: e.g., my-rds-db
* **Master username** and **password** (save them).
* **Initial DB name** (optional — creates the first DB/schema).

1. **DB instance class & storage**:

* Select **DB instance class** (e.g., db.t3.micro for dev).
* Select **Storage type** (gp3/gp2 or Provisioned IOPS) and **Allocated storage** (GB).
* Optionally enable **storage autoscaling**.

1. **Connectivity**:

* Choose **VPC** and **subnets** (use multiple AZs for HA).
* **Public access**: Yes (if you need direct internet access) or No (recommended for private).
* **VPC security group**: choose existing or create new (ensure it allows DB port inbound).
* **Availability & durability**: enable **Multi-AZ** for production high availability.

1. **Additional configuration**:

* **Backup retention** days (e.g., 7).
* **Backup window** (or leave Auto).
* Enable **Encryption** (KMS) if required.
* Enable **Performance Insights** / **Enhanced Monitoring** if needed.
* **Deletion protection** (recommended for production).

1. **Tags** (optional): add Name / Environment tags; enable **propagate** if needed.
2. Click **Create database**. RDS will provision the instance (few minutes).

**After creation — quick checks & connect**

1. Go to **RDS → Databases**, select your instance.
2. Copy the **Endpoint** (hostname) and **Port**.
3. Edit the **Security Group** inbound rule: allow your application IP or EC2 security group on the DB port (MySQL 3306 / Postgres 5432).
4. Test connection:
   * MySQL: mysql -h <endpoint> -P 3306 -u <user> -p
   * Postgres: psql -h <endpoint> -p 5432 -U <user> -d <dbname>
5. Use the endpoint in your application (JDBC URL).

# ✅ Multi-AZ in RDS

# What is Multi-AZ in RDS?

When you enable **multi-AZ deployment** for an RDS instance:

* AWS creates a **primary database** in one AZ.
* It automatically makes a **standby replica** in another AZ.
* **AWS manages both databases** — no manual setup needed.
* If the **primary DB fails**, **AWS automatically switches** to the standby.

# How It Works

**Scenario 1: Normal Operations**

* Your application connects to the **primary DB**.
* RDS performs **synchronous replication** to the **standby DB**.
* The standby DB is **not readable**; it's only used for failover.

**Scenario 2: Primary DB Failure**

* If the **primary DB** becomes unavailable due to:
  + Hardware failure
  + AZ outage
  + Network issues
  + Manual reboot or maintenance
* **Automatic Failover Happens**:
  + The standby DB becomes the new primary.
  + AWS automatically updates the **DNS endpoint** to point to the new primary.
  + Your application **doesn’t need code changes**—just reconnects using the same endpoint.

**Key Features of Multi-AZ**

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **High Availability** | Provides automatic failover to minimize downtime. |
| **Synchronous Replication** | Standby DB always has up-to-date data. |
| **Automatic Backups** | Backups are taken from the standby instance to avoid load on the primary. |
| **Automatic Patching** | Patches are applied first to the standby, then the primary, minimizing downtime. |
| **Single Endpoint** | Application connects to one DNS endpoint; AWS manages switching. |
| **Data Durability** | Protects data from AZ-level failures. |

# Multi-AZ vs multi-Region vs Read Replicas

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Multi-AZ** | **Multi-Region** | **Read Replicas** |
| **Purpose** | High availability & disaster recovery | Disaster recovery across regions | Scaling read performance |
| **Replication Type** | Synchronous | Asynchronous | Asynchronous |
| **Standby Readable?** | ❌ No | ❌ No | ✅ Yes |
| **Use Case** | Automatic failover within a region | Backup for region-wide failure | Offload read traffic & an |

**Example Scenario**

Imagine you’re running an **insurance project** (like your Vehicle Insurance Project) on AWS RDS with MySQL.

* **Primary DB:** rds-primary in **ap-south-1a** (Mumbai AZ1)
* **Standby DB:** rds-standby in **ap-south-1b** (Mumbai AZ2)

**Application connects using:**

jdbc: mysql://vehicle-insurance-db.abc123.ap-south-1.rds.amazonaws.com:3306/insurance

* If AZ1 fails:
  + AWS promotes **rds-standby** in AZ2.
  + DNS automatically points to the new primary.
  + Application reconnects automatically — **no config changes needed**.

# When to Use Multi-AZ

* For **high availability** and **automatic failover**.
* To ensure **minimal downtime** and **data durability**.

**When Not Needed ❌**

* For **development/testing** environments.
* If **manual failover** is acceptable.
* For **read scaling** → use **Read Replicas** instead.

**Pricing Impact 💰**

* **Multi-AZ costs more** since you pay for **primary**, **standby**, and **storage**.
* Example: **Single-AZ: $0.20/hr** → **Multi-AZ: ~$0.40/hr**.

**RDS Multi-AZ Overview 🚀**

* AWS creates a **primary database** in one AZ and a **standby replica** in another AZ.
* Data is **synchronously replicated** to the standby.
* **AWS manages failover automatically** — no manual setup or intervention needed.
* If the **primary DB fails**, AWS **promotes the standby** and **updates the DNS** automatically.
* Applications reconnect using the **same endpoint** without code changes.

# Steps to Create RDS Multi-AZ 🛠️

1. **Login** → Go to AWS Console → Open **RDS**.
2. **Create DB** → Click **Create database** → Choose **Standard create**.
3. **Select Engine** → MySQL, PostgreSQL, etc.
4. **Deployment** → Choose **Multi-AZ DB instance** ✅.
5. **DB Settings** → Set **DB name**, **username**, **password**.
6. **Instance Class** → Pick size (e.g., db.t3.medium).
7. **Storage** → Set allocated storage & enable autoscaling if needed.
8. **VPC & Subnets** → Select **VPC**, **Subnets**, and enable **Multi-AZ**.
9. **Connectivity** → Choose **security group**, decide **public access**.
10. **Backups & Monitoring** → Enable automatic backups & monitoring.
11. **Review & Create** → Click **Create database**.
12. **Connect** → Use the **RDS endpoint** in your Spring Boot app:

spring.datasource.url=jdbc:mysql://<rds-endpoint>:3306/insurance

# ✅ Read Replica

**What is a Read Replica in RDS? 🗂️**

* The **main database** (called **primary**) handles **both read and write** operations.
* A **Read Replica** is a **copy of your RDS database** created only for **read operations**.
* It automatically receives data from the **primary DB.**
* helps **reduce load** and handling **read queries** separately.

**Why We Use Read Replicas** ✅

* To **speed up performance** by handling **read-heavy queries**.
* To **share read traffic** and run **reports or analytics** without affecting the main database.

**Key Points** ✅

* **Replication** → Happens **asynchronously** (small delay possible).
* **Read-only** → Replicas **cannot** handle INSERT, UPDATE, or DELETE.
* **Multiple Replicas** → You can create up to **15 replicas**.
* **Cross-region support** → Replicas can be in **different regions**.

**How It Works 🔄**

1. You create a **primary RDS database**.
2. You create **one or more read replicas** from it.
3. The primary database **sends updates** to replicas **automatically**.
4. Applications **read** from replicas but **write** only to the primary database.

**Simple Example**

* **Primary DB** → Handles **write queries** (e.g., add a new insurance policy).
* **Read Replica** → Handles **read queries** (e.g., show insurance history).
* If you have **10,000 users**, replicas **reduce pressure** on the main DB, keeping your app **fast**.

**Read Replica vs Multi-AZ 🔀**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Read Replica 🟢** | **Multi-AZ 🔵** |
| **Purpose** | Improve **read performance** | Ensure **high availability** |
| **Replication** | **Asynchronous** | **Synchronous** |
| **Readable?** | ✅ Yes | ❌ No |
| **Automatic Failover** | ❌ No | ✅ Yes |
| **Use Case** | Handle heavy **read queries** | Keep DB **always online** |

**Steps to Create a Read Replica in AWS Console 🛠️**

1. **Login to AWS Console** → Open **RDS**.
2. **Select Primary DB** → Go to **Databases** → Click on your **primary database**.
3. **Create Read Replica** → Click **Actions → Create read replica**.
4. **Configure Replica** →
   * Set **DB instance identifier**.
   * Choose **instance class** (size).
   * Select **VPC, subnets**, and **availability zone**.
   * Enable **multi-AZ** if needed.
5. **Additional Settings** → Enable **backups**, **monitoring**, and **encryption** if required.
6. **Create** → Click **Create read replica**.
7. **Connect** → Use the **replica endpoint** in your application for **read queries**.

# ✅ RDS Backup

**What is RDS Backup**

* **Automated Backups** → AWS takes **daily snapshots** and keeps **transaction logs**, letting you **restore to any point in time** within the retention period.
* **Manual Snapshots** → You can **create backups anytime**, which are kept until you delete them.

**Purpose** → Protects your data, allows **point-in-time recovery**, and helps with **disaster recovery** or safe maintenance.

**Why Use RDS Backup ✅**

* **Data protection** → Recover from accidental deletion or corruption.
* **Point-in-time recovery** → Restore DB to any second within retention period.
* **Disaster recovery** → Protect against AZ or hardware failures.
* **Safe maintenance** → Create snapshot before upgrades or migrations.

**How Backups Work** ✅

* Automated backups include **full snapshot + transaction logs**.
* Stored **across multiple AZs** for durability.
* Allows **restoring the database** to a specific point in time.

**Why Use RDS Backup ✅**

* **Data protection** → Recover from accidental deletion or corruption.
* **Point-in-time recovery** → Restore DB to any second within retention period.
* **Disaster recovery** → Protect against AZ or hardware failures.
* **Safe maintenance** → Create snapshot before upgrades or migrations.

**Example in Spring Boot**

* You don’t need to change code for backups.
* Backups are managed **automatically by AWS**, and you can **restore via console or CLI**.

# Steps to Create/Manage RDS Backup 🛠️

**1. Automated Backups**

1. Go to **AWS Console → RDS → Databases**.
2. Select your **DB instance**.
3. Under **Backup**, enable **Automated Backups**.
4. Set **retention period** (1–35 days).
5. AWS automatically takes **daily snapshots** and logs for point-in-time recovery.

**2. Manual Snapshots**

1. Go to **AWS Console → RDS → Databases**.
2. Select your **DB instance**.
3. Click **Actions → Take snapshot**.
4. Enter a **name** for the snapshot.
5. Click **Create snapshot** → Stored until you **delete it manually**.

**3. Restore from Backup**

1. Go to **RDS → Snapshots**.
2. Select the **snapshot** or **point-in-time backup**.
3. Click **Restore** → Configure **DB instance settings**.
4. AWS creates a **new database** from the backup.

# ✅ AWS Lambda

**1. What is AWS Lambda?**

1. **Lambda** is a **serverless compute service** provided by **AWS** that **runs your code** without needing to manage servers.
2. You **No Need to set up, configure, monitor, or maintain servers** — AWS takes care of the rest.  
   You **just focus on writing code**.

**Why Use Lambda?** 🚀

* **No servers** → AWS manages everything
* **Automatic execution** → Runs only when triggered
* **Cost-effective** → Pay only for execution time
* **Scales automatically** → Can handle 1 request or 1M requests
* **Easy integration** → Works with almost all AWS services

**Example Scenario**

**Traditional Way (Before Lambda)**

You need to **buy or rent servers** and **set up OS, software, and monitoring tools**.  
Servers must **run 24/7**, leading to **high costs even when idle**.

**Example:**  
For a Java file-processing app, you rent an EC2 instance, install Java, configure everything, and keep the server running even if files come **once a day**.

**With AWS Lambda (Serverless Way)**

No server management needed — just **write and upload your code**.  
AWS runs it **only when required** and you **pay only for execution time**.

**Example:**  
Upload the same Java app to **Lambda**.  
When a file is uploaded to **S3**, Lambda **automatically runs** your code and stops after finishing — no server setup, updates, or scaling needed.

**Steps to Create AWS Lambda Function ✅**

1. **Open AWS Lambda** → Go to AWS Console → Search **Lambda** → Click **Create Function**.
2. **Author from Scratch** → Give a **name**, choose **runtime** (Python, Java, Node.js), and select **permissions** (create a new role).
3. **Write or Upload Code** → Use the inline editor or upload a **ZIP file**.
4. **Add Trigger (Optional)** → Example: S3, API Gateway, or EventBridge.
5. **Test the Function** → Create a test event → Click **Test** → Check results.
6. **Monitor Logs** → Go to **CloudWatch** for execution details.

**NOTE:**

AWS Lambda lets you run code without managing servers. To create it, go to **AWS Console → Lambda → Create Function**, write or upload your code, configure triggers if needed, and test the function.