Sr. No.	Name of Experiments / Case Studies (Any Eight)	COs
1	Study and implementation of Infrastructure as a Service-Amazon Elastic Compute Cloud (EC2) using AWS Management Console .	
2	Create AWS VPC For Add or remove an IPv6 CIDR block from your subnet.	
3	Study & Implement Windows VM using the Terraform code.	
4	Understand & implement how hypervisors such as QEMU utilize the Linux KVM API to provide efficient hardware-assisted virtualization.	
5	Configuring Athena to access data in Amazon S3 for Running Queries in AWS Athena.	
6	Study the Migration tools and services to facilitate the migration of workloads to its cloud platform (use any cloud platform).	
7	Study and Implement CloudFormation templates to provide the requested resources in your AWS account.	
8	Implement container management with Kubernetes.	
9	Implement common scenarios by using the AWS SDK for Kotlin with IAM.	
10	Demonstrate the use of Cloud Computing Security Tools (Notable Open Source).	
Content Beyond Syllabus		
11	DevOps Tooling by AWS for to Adopt a DevOps Model.	

EXPERIMENT NO. 1

AIM: Study and implementation of infrastructure as a Service- Amazon Elastic Compute Cloud (EC2) using AWS Management Console.

Prerequisites:

- An active AWS account with console access.
- Basic understanding of cloud computing concepts.
- Access to a computer with internet connectivity and SSH client (for connecting to Linux instances).
- Familiarity with using a terminal/command prompt is helpful but not mandatory.

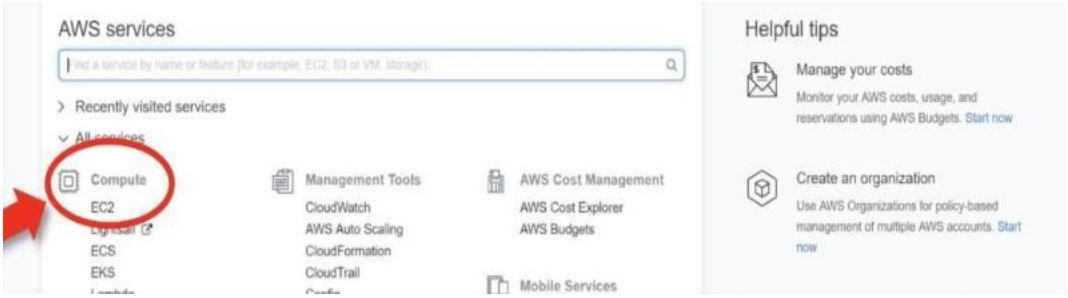
Theory:

Amazon EC2 (Elastic Compute Cloud) is an on-demand cloud computing service from AWS that lets you rent virtual servers to run applications. Instead of maintaining physical hardware, you can launch virtual machines with customizable memory, CPU, and storage.

Follow the below steps to create an EC2 instance in AWS (Amazon):

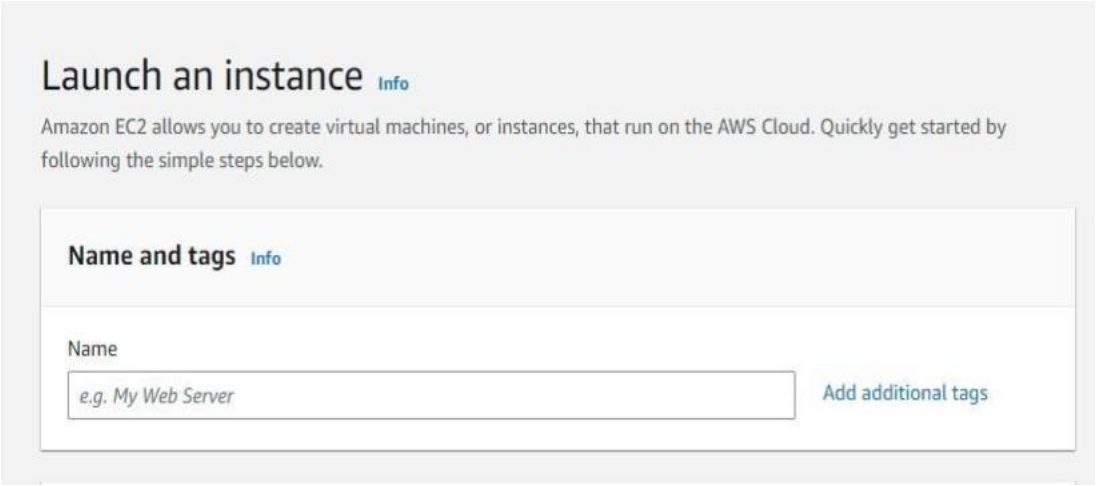
Step 1: Log in and Open EC2 Dashboard

- Sign in to your AWS account.
- Click Services > EC2.
- Check if any instances are running under Instances running.



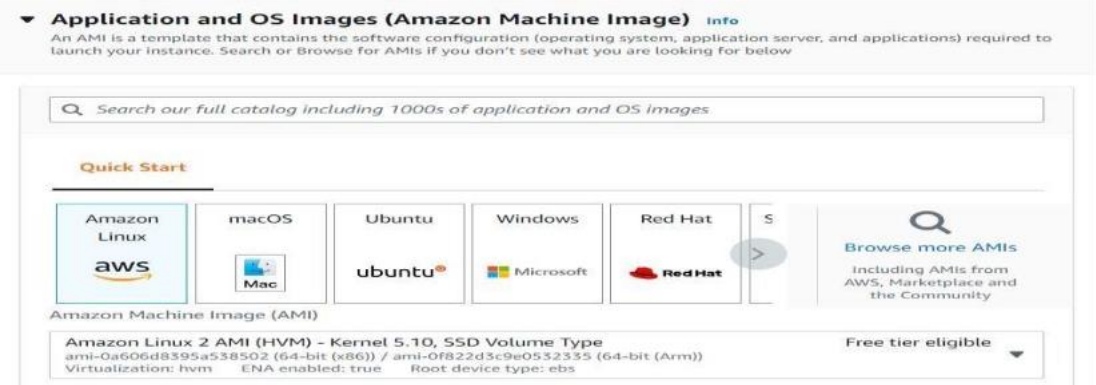
Step 2: Launch a New EC2 Instance

- Click Launch Instance.
- Enter a name for your instance.



Step 3: Select an Amazon Machine Image (AMI)

- Select AMI - Required operating system from the available. There are different types of OS available select the OS as per your requirement.

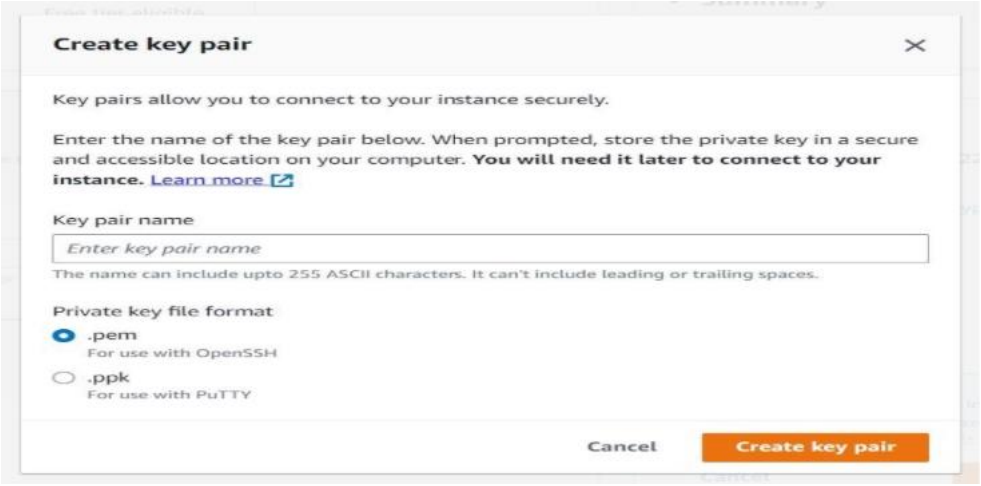


Step 4: Choose Instance Type

- By default, it selects a free tier of storage. (If you are eligible for the free tier). From the available storage specifications, select a free tier-eligible storage service.

Step 5: Configure Key Pair

- Create a new key pair (select .pem format) and download it.
- Keep this file safe—you'll need it to connect.

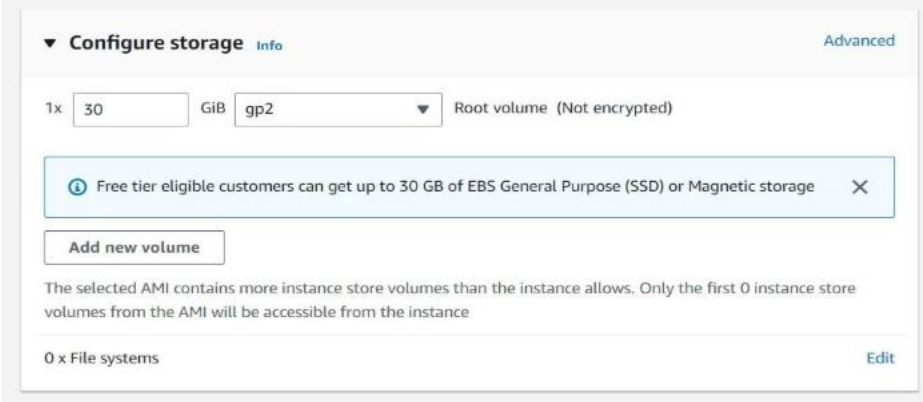


Step 6: Configure Network and Storage

- Use default network settings

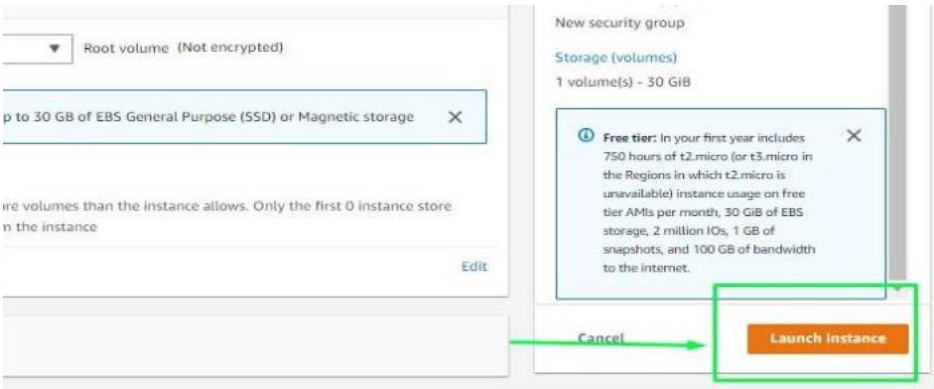
(VPC,subnet).

- Storage defaults to 8 GB (free tier eligible); keep it as is or adjust if needed.



Step 7: Review and Launch

 Verify all selections (ensure free tier eligibility if applicable). Click Launch to start your instance.



Conclusion:

This exercise demonstrated the fundamental concepts of Infrastructure as a Service (IaaS) using AWS EC2.By using EC2, we can deploy scalable and secure virtual machines without the need for physical hardware management. This flexibility, along with the ability to customize the environment and pay-as-you-go pricing, makes EC2 a powerful tool for deploying applications in the cloud.

EXPERIMENT NO. 2

AIM: Create AWS VPC For Add or remove an IPv6 CIDR block from your subnet.

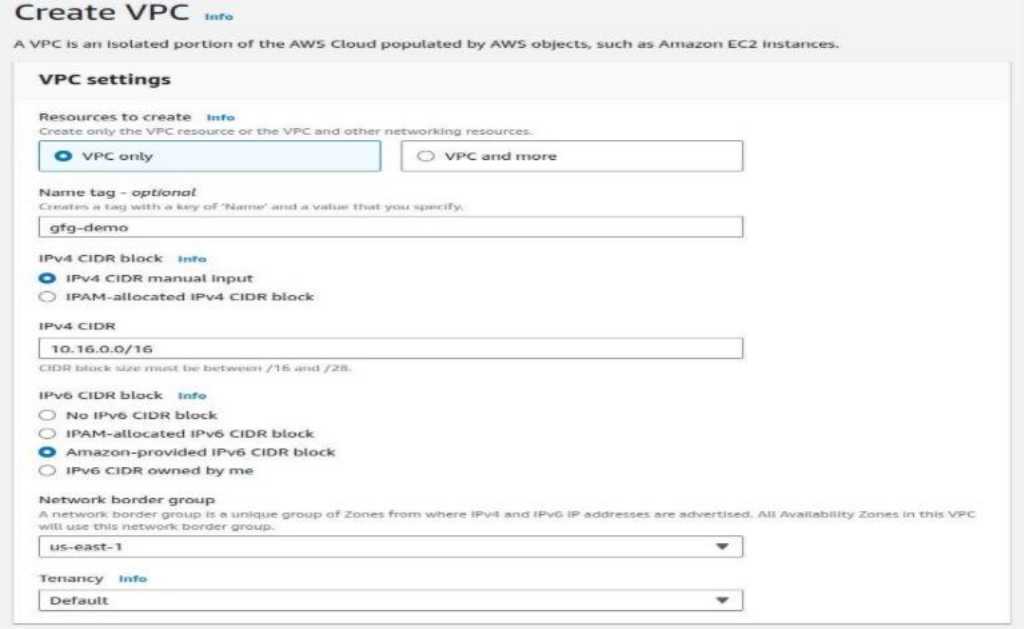
Prerequisites:

- An active AWS account with console access.
- Basic understanding of cloud computing concepts.
- Access to a computer with internet connectivity and SSH client (for connecting to Linux instances).
- Existing VPC with an IPv4 CIDR block.
- VPC must have an associated IPv6 CIDR block (Amazon-provided or BYO).
- At least one subnet in the VPC.

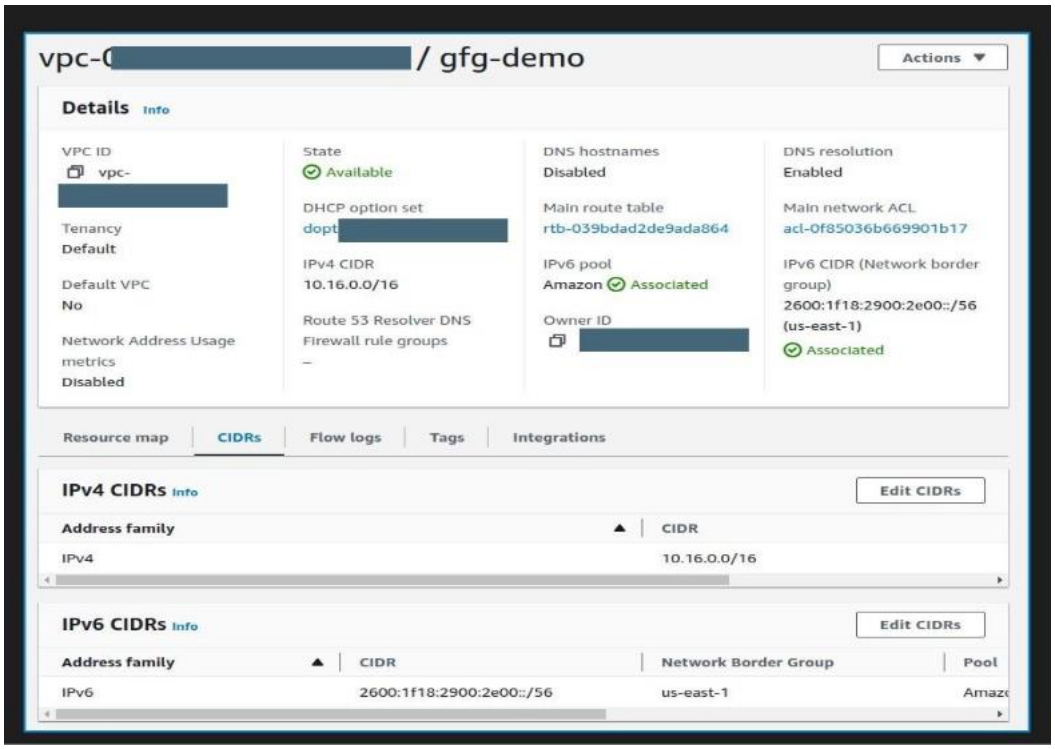
Theory: Amazon Web Service allows users to securely host applications and scale to millions of users without worrying about infrastructure provision and maintenance.

A Virtual Private Cloud (VPC) is a customizable, isolated network environment for deploying cloud resources. A default VPC is configured and ready for you to use when you create your account. You can also create your Custom VPC.

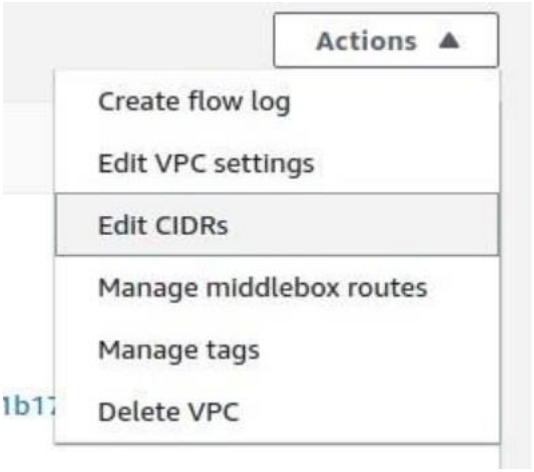
1. **Create or select your VPC**
   - Go to the VPC Dashboard in the AWS Console.
   - If your VPC doesn't have an IPv6 CIDR block yet, you can assign one there by choosing Actions > Edit CIDRs > Add IPv6 CIDR.



2. **Create or select your Subnet**
   - In the VPC console, click on Subnets.
   - When creating a new subnet, you'll see an option to assign an IPv6 CIDR block. It will let you pick a /64 subnet from your VPC's IPv6 CIDR block**.**
3. **Add an IPv6 CIDR block to an existing subnet** Select the subnet in the console.
   - Choose Actions > Edit IPv6 CIDRs or similar options (depends on UI updates).
   - You can then associate (add) a new IPv6 CIDR block to the subnet.



4. **Remove an IPv6 CIDR block from a subnet**
   - In the subnet details, look for the IPv6 CIDR associations.
   - Select the IPv6 CIDR block you want to remove and disassociate it.



Conclusion:

We can **add or remove an IPv6 CIDR block from your subnet** once your VPC has an associated IPv6 CIDR block and you have the necessary permissions. This can be done easily through the AWS Management Console without any coding or automated using AWS CLI/SDK. Proper subnet, routing, and security configurations ensure your IPv6 network works smoothly.

EXPERIMENT NO. 3

AIM:

Study and implementation of Windows Virtual Machine provisioning using Terraform Infrastructure-as-Code (IaC) on AWS.

Prerequisites:

- An active AWS account with access credentials.
- Terraform installed and configured on the system.
- Basic understanding of cloud computing and virtualization.
- A computer with internet access.
- AWS key pair and a valid AMI ID for Windows OS.
- Familiarity with terminal/command line is helpful.

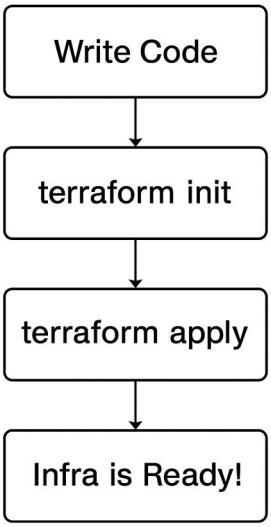
Theory:

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

With Terraform, users can automate the provisioning of cloud resources such as virtual machines, networks, and storage across multiple providers like AWS, Azure, and GCP.

In this experiment, we use Terraform to provision a Windows EC2 instance on AWS Cloud, eliminating the need to manually configure infrastructure through the AWS Console.

Terraform Workflow:



Steps to Launch Windows VM on AWS using Terraform:

1. Step 1: Install and Initialize Terraform

- Download and install Terraform from the official site.

- Open terminal and run `terraform -version` to verify installation.

2. Step 2: Configure Provider and Resources

- Create a working directory and inside it, create a file named `main.tf`.

- Define the AWS provider and EC2 instance resource as shown below:

- Terraform code:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "windows_vm" {
  ami           = "ami-0b2f6494ff0b07a0e" # Example AMI for
Windows Server
  instance_type = "t2.micro"
  key_name      = "your-key-name"

  tags = {
    Name = "Terraform-Windows-VM"
  }
}
```

3. Step 3: Initialize and Plan Deployment

- Navigate to your Terraform directory in the terminal.

- Run the following commands:

```
terraform init
terraform plan
```

4. Step 4: Apply Configuration

- Run `terraform apply` and confirm when prompted.

- Terraform will now provision the instance.

5. Step 5: Verify the Instance

- Open AWS Console > EC2 Dashboard.
- Check that your Windows instance is running.

Expected Output:

A Windows EC2 instance should be created and visible in the AWS Console. You can connect to the instance using RDP (Remote Desktop Protocol) if properly configured.

Attach screenshot of the running instance here.

Conclusion:

In this experiment, we successfully demonstrated the provisioning of a Windows EC2 instance using Terraform, reflecting the concept of Infrastructure as Code. Terraform automates cloud resource deployment, reduces manual error, and enhances scalability. This experiment provides practical knowledge on defining infrastructure declaratively and managing lifecycle changes efficiently.

EXPERIMENT NO. 4

AIM:

To understand and implement how hypervisors such as QEMU utilize the Linux Kernel-based Virtual Machine (KVM) API to provide efficient, hardware-assisted virtualization on modern Linux systems.

Prerequisites:

• A system with a CPU that supports virtualization (Intel VT-x or AMD-V enabled in BIOS/UEFI).

• A Linux OS (preferably Ubuntu/Debian).

• Terminal access with sudo privileges.

• Installed tools: qemu-kvm, virt-manager, libvirt-daemon, bridge-utils.

• ISO image of an operating system (like Ubuntu) for testing VM creation.

Theory:

Virtualization is a technique where a single physical machine is partitioned into multiple virtual environments, known as Virtual

Machines (VMs). This allows better utilization of hardware resources and isolation between applications.

Hypervisors:

Hypervisors are software or firmware that create and run VMs. They act as an interface between virtual machines and the physical hardware.

Type	Description	Examples
Type 1	Bare-metal; runs directly on hardware	VMware ESXi, Xen
Type 2	Hosted; runs on top of a host OS	VirtualBox, QEMU with KVM

What is KVM?

KVM (Kernel-based Virtual Machine) is a full virtualization solution built into the Linux kernel. It converts the Linux kernel into a Type 1 hypervisor by exposing virtualization extensions (Intel VT-x or AMD-V) to user-space applications like QEMU.

What is QEMU?

QEMU is a generic, open-source machine emulator and virtualizer. When run in emulation mode (without KVM), it simulates all hardware which is slower. With KVM enabled, QEMU becomes a fast virtualizer by utilizing CPU instructions and hardware features via the KVM API.

Relationship Between QEMU and KVM:
QEMU acts as the frontend (user-space application) while KVM acts as the backend (kernel module). QEMU uses the KVM API to access privileged CPU instructions, page tables, and I/O access via the /dev/kvm device file.

Steps to Implement:

• Step 1: Check Virtualization Support in CPU:

```
egrep -c '(vmx|svm)' /proc/cpuinfo
```

• Step 2: Install KVM and QEMU-related Packages:

```
sudo apt update
sudo apt install qemu-kvm libvirt-daemon-system libvirt-clients
bridge-utils virt-manager –y
```

• Step 3: Verify KVM is Enabled:

```
lsmod | grep kvm
```

• Step 4: Check Permissions and Add User:

```
groups $USER
sudo usermod -aG libvirt $USER
```

```
sudo usermod -aG kvm $USER
reboot
```

• Step 5: Launch GUI-based Virtual Machine:

```
virt-manager
```

• Step 6: Launch VM via QEMU Command Line:

```
qemu-system-x86_64 -enable-kvm -m 2048 -hda ubuntu.img -cdrom ubuntu.iso -boot d
```

Expected Output:

• Virtualization modules (kvm, kvm_intel or kvm_amd) loaded successfully.
• QEMU creates a VM with near-native performance using KVM.
• Virtual machine boots and installs guest OS successfully.
• Verification via virt-manager shows VMs using KVM backend.

Conclusion:

In this experiment, we explored how QEMU utilizes the Linux KVM API to enable fast, hardware-assisted virtualization. We understood the interaction between user-space QEMU and the kernel-level KVM module. By using both GUI and CLI approaches, we successfully created and ran a virtual machine,

reinforcing the concept of efficient virtualization in modern cloud and edge computing platforms.

EXPERIMENT NO. 5

AIM:
To configure Amazon Athena to access data stored in Amazon S3 and execute SQL queries directly on that data using the Athena query engine.

Prerequisites:

• An AWS account with access to S3 and Athena services.

• An existing S3 bucket with structured data (e.g., CSV, JSON, Parquet).

• IAM user with AmazonAthenaFullAccess and AmazonS3ReadOnlyAccess policies.

• Region set where Athena and the S3 bucket are both available.

• Optional: A simple dataset in CSV format uploaded to an S3 path.

Theory:
Amazon Athena is a serverless, interactive query service that enables users to analyze data directly in Amazon S3 using standard SQL. It reads files directly from S3, supports multiple data formats, and does not require provisioning infrastructure. Athena uses the Presto query engine under the hood.

How Athena Works:
• Athena reads files directly from S3.
• Supports CSV, JSON, ORC, Avro, and Parquet formats.
• Schema and tables are defined using the Data Catalog.
• Athena stores query results in an S3 output location.
• No infrastructure provisioning required.

Steps to Perform:

- Step 1: Prepare Data in S3

Upload a sample CSV (e.g., students.csv) to an S3 bucket:

s3://your-bucket-name/data/students.csv

Example content:
id,name,dept,marks
1,Alice,CS,85
2,Bob,IT,78

- Step 2: Create an S3 Output Folder for Athena

In the same or another S3 bucket, create a folder like:
s3://your-bucket-name/athena-results/

- Step 3: Open Athena Console

Go to AWS Console > Athena.
In Settings (top right), configure the result location to:
s3://your-bucket-name/athena-results/

- Step 4: Create Database and Table

Run the following SQL commands in the Athena query editor:

```
CREATE DATABASE IF NOT EXISTS student_data;

CREATE EXTERNAL TABLE IF NOT EXISTS
student_data.students (
  id INT,
  name STRING,
  dept STRING,
  marks INT
)
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
  "quoteChar" = "\""
```

)
LOCATION 's3://your-bucket-name/data/';

- Step 5: Run Sample Queries

SELECT * FROM students;

SELECT * FROM students WHERE marks > 80;

SELECT dept, AVG(marks) AS avg_marks FROM students
GROUP BY dept;

Expected Output:
• Athena successfully queries the S3 file and returns tabular results.
• The query result files are stored in the specified S3 output folder.
• SQL-based queries on large datasets can be performed without provisioning a database engine.

Conclusion:
In this experiment, we configured Amazon Athena to run serverless SQL queries directly on data stored in Amazon S3. We created a schema using Athena's query editor, defined external tables, and successfully performed queries on structured CSV data. This demonstrates how AWS Athena simplifies data analytics workflows with minimal setup and cost-effective execution.

EXPERIMENT NO. 6

AIM:
To study various migration tools and services offered by cloud platforms such as AWS, Azure, or GCP, which facilitate smooth migration of workloads (data, applications, VMs, databases) to the cloud.

Prerequisites:

• Basic understanding of cloud computing and virtualization.

• Internet-connected system with access to at least one cloud provider's console (e.g., AWS, Azure, GCP).

• Optional: Admin-level access or free-tier account on the selected platform.

• Sample workload (VM, DB, file server) to understand migration flow.

Theory:

Cloud Migration refers to the process of moving digital business operations — such as applications, workloads, data, and services — to a cloud computing environment. Cloud platforms provide dedicated Migration Services to simplify, accelerate, and secure this process.

Key Migration Scenarios:

Workload Type	Target	Example Tool/Service
VMs/Servers	Cloud VM	AWS MGN, Azure Migrate
Databases	Managed DB	AWS DMS, Azure Database Migration Service
Files/Storage	Cloud Object Store	AWS Snowball, Azure Data Box
Apps	Cloud App Service	Google Migrate for Compute Engine

Common Migration Tools by Cloud Providers:

- AWS (Amazon Web Services):

• AWS Application Migration Service (MGN): Automates lift-and-shift migrations.

• AWS Database Migration Service (DMS): Migrates databases to AWS with minimal downtime.

• AWS Snowball: Physical device for offline data transfer.

• Cloud Endure (now part of MGN): Continuous replication with minimal downtime.

- Microsoft Azure:

• Azure Migrate: Unified migration platform for servers, databases, and apps.

• Azure Database Migration Service: Moves on-premises DBs to Azure SQL, Cosmos DB.

• Azure Data Box: Offline data transfer device similar to AWS Snowball.

- Google Cloud Platform (GCP):

• Migrate to Virtual Machines: Migrates VMs to Google Compute Engine.

• Database Migration Service (DMS): Cloud-native database migration tool.

• Transfer Appliance: Offline bulk data transfer device.

Steps to Study a Migration Tool (e.g., AWS MGN):

• Step 1: Sign in to AWS Console

- Go to AWS MGN → Enable the service.

• Step 2: Install Replication Agent

- On the source server, download and install AWS Replication Agent.
- Configure the agent with your AWS credentials.

• Step 3: Configure Source Server

- AWS detects the server and starts real-time block-level replication.

• Step 4: Launch Test Instances

- Launch test instances to verify application functionality.

• Step 5: Cut Over

- Launch the migrated server as a live instance and redirect traffic.

Expected Output:
• Migration tool is successfully configured.
• Workload (e.g., a VM or DB) is replicated or transferred to the cloud.
• Migration is verified via test or cut-over instance.
• Dashboard shows success status and minimal downtime.

Conclusion:

In this experiment, we explored the migration process and studied the tools offered by major cloud providers like AWS, Azure, and GCP. By examining services such as AWS MGN, Azure Migrate, and Google DMS, we understand how organizations move workloads efficiently to the cloud with minimal downtime and high reliability.

EXPERIMENT NO. 7

AIM:

Study and Implement CloudFormation templates to provide the requested resources in your AWS Account.

Prerequisites:

* An active AWS account with access to AWS Management Console.

* Basic knowledge of JSON or YAML syntax.

* Familiarity with AWS services like EC2, S3, IAM, etc.

* Internet connectivity and a computer.

* AWS CloudFormation service enabled in your region.

Theory:

AWS CloudFormation is an Infrastructure as Code (IaC) service that helps you model and set up your

AWS resources so that you can spend less time managing infrastructure and more time focusing on your

applications. With CloudFormation, you define a template using either YAML or JSON to provision AWS

resources such as EC2 instances, S3 buckets, security groups, etc.

The key concepts include:

Template: A configuration file written in JSON or YAML.

Stack: A collection of AWS resources created and managed as a single unit.

Resources: The AWS services you want to create.

Parameters, Mappings, Outputs: Optional elements used to customize templates.

CloudFormation ensures automation, version control, and repeatability of infrastructure deployments.

Steps to Create and Launch a CloudFormation Stack:

Step 1: Log in to AWS Console

- Go to https://aws.amazon.com

- Sign in with your credentials.

Step 2: Open CloudFormation

- Navigate to Services > CloudFormation.

- Click on “Create stack” > “With new resources (standard)”.

Step 3: Upload or Create the Template

- Choose “Upload a template file”.

- Click “Choose file” and upload a pre-written template (e.g., ec2-sample.yaml).

- Alternatively, you can select “Create template in Designer” to visually design one.

Step 4: Configure Stack Details

- Provide a stack name (e.g., MyFirstStack).

- Specify parameters if required (for dynamic templates).

Step 5: Configure Stack Options

- Add optional tags (e.g., Name=DevStack).

- Leave other settings as default for now.

Step 6: Review and Create

- Review the summary and acknowledge that AWS might create IAM resources.

- Click Create stack.

Step 7: Monitor Stack Creation

- You'll be redirected to the Stack events tab.

- Wait until the status changes to "CREATE_COMPLETE".

Step 8: Verify the Created Resources

- Go to EC2 Dashboard (or the relevant service) and verify the resource(s).

- To delete the resources, go to the stack and click "Delete".

Conclusion:

In this experiment, we explored AWS CloudFormation as an automation tool for provisioning cloud

resources. We learned how to define infrastructure using a YAML template, launch it as a stack, and

manage it through the AWS Management Console. This approach ensures consistent and scalable infrastructure deployment while minimizing manual effort and errors.

EXPERIMENT NO. 8

AIM: Implement container management with Kubernetes.

Prerequisites:

* Basic knowledge of containers and Docker.

* Kubernetes installed via Minikube, Docker Desktop, or using a cloud provider (GKE, EKS, or AKS).

* kubectl command-line tool installed and configured.

* A working container image (e.g., Nginx or custom Docker image).

* Internet connectivity.

Theory:

Kubernetes (also known as K8s) is an open-source container orchestration platform that automates

deployment, scaling, and management of containerized applications.

Key concepts:

* Pod: The smallest deployable unit in Kubernetes, typically wraps one or more containers.

* Node: A physical or virtual machine that runs the containers.

* Cluster: A set of nodes managed by the Kubernetes control plane.

* Deployment: A controller that manages replicas of pods for high availability.

* Service: An abstraction that defines a set of pods and a policy by which to access them (e.g.,

LoadBalancer, ClusterIP).

Kubernetes simplifies container management by handling scheduling, updates, networking, and scaling.

Steps to Deploy and Manage a Container Using Kubernetes:

Step 1: Start Minikube (or any K8s cluster)

- Open terminal and start the local Kubernetes cluster:

minikube start

Step 2: Create a Deployment

- Create a deployment using a simple container image (e.g., Nginx):

kubectl create deployment my-nginx --image=nginx

Step 3: Verify Deployment

- Check if the pod is running:

kubectl get pods

kubectl get deployments

Step 4: Expose the Deployment as a Service

- To access the Nginx app, expose it via a NodePort:

kubectl expose deployment my-nginx --type=NodePort --port=80

Step 5: Get the Access URL

- Retrieve the URL to access the application:

minikube service my-nginx --url

- Open the URL in your browser to see the default Nginx page.

Step 6: Scale the Deployment

- Increase the number of running pods:

kubectl scale deployment my-nginx --replicas=3

- Check if scaling was successful:

kubectl get pods

Step 7: Update the Deployment (Optional)

- Roll out a new image version:

kubectl set image deployment/my-nginx nginx=nginx:1.19

-Watch the rollout status:

kubectl rollout status deployment/my-nginx

Step 8: Clean Up

-Delete service and deployment after testing:

kubectl delete service my-nginx

kubectl delete deployment my-nginx

minikube stop

Conclusion:

In this experiment, we successfully implemented container management using Kubernetes. We created a

deployment, exposed it via a service, scaled it, and even updated it — all with simple commands.

Kubernetes provides robust tools for orchestrating containers, offering high availability, scalability, and

maintainability of cloud-native applications.

EXPERIMENT NO. 9

AIM:
To implement common AWS operations using the AWS SDK for Kotlin and IAM (Identity and Access Management) for secure, programmatic interaction with AWS services.

Prerequisites:
• Kotlin development environment (IntelliJ IDEA or CLI).

• AWS CLI configured (aws configure).

• AWS SDK for Kotlin installed via Gradle or Maven.

• IAM user with programmatic access (Access Key & Secret).

• Basic understanding of Kotlin, AWS IAM, and AWS services like S3 or EC2.

Theory:
The AWS SDK for Kotlin enables developers to interact with AWS services directly from Kotlin applications. It provides Kotlin-native APIs and allows operations such as listing S3

buckets, uploading files, starting/stopping EC2 instances, and more.

IAM (Identity and Access Management) is used to manage access securely by creating users, assigning permissions, and using roles.

IAM + AWS SDK: How It Works

• Create an IAM user with necessary permissions (e.g., S3 read/write).
• Use the user's access key and secret key for authentication.
• The AWS SDK uses these credentials to securely call AWS APIs.

Common Use Cases to Implement:

- Example 1: List All S3 Buckets

```
import aws.sdk.kotlin.services.s3.S3Client
import aws.sdk.kotlin.services.s3.model.ListBucketsRequest

suspend fun listBuckets() {
    S3Client { region = "us-east-1" }.use { s3 ->
        val response = s3.listBuckets(ListBucketsRequest {})
        response.buckets?.forEach { println(it.name) }
    }
}
```

- Example 2: Upload a File to S3

```
import aws.sdk.kotlin.services.s3.model.PutObjectRequest
import java.io.File
import aws.smithy.kotlin.runtime.content.ByteStream
```

```
suspend fun uploadFile(bucket: String, key: String, filePath:
String) {
    val request = PutObjectRequest {
        this.bucket = bucket
        this.key = key
        this.body = ByteStream.fromFile(File(filePath))
    }

    S3Client { region = "us-east-1" }.use { s3 ->
        s3.putObject(request)
        println("Uploaded $key to $bucket.")
    }
}
```

- Example 3: Describe EC2 Instances

```
import aws.sdk.kotlin.services.ec2.Ec2Client
import
aws.sdk.kotlin.services.ec2.model.DescribeInstancesRequest

suspend fun describeInstances() {
    Ec2Client { region = "us-east-1" }.use { ec2 ->
        val response =
ec2.describeInstances(DescribeInstancesRequest {})
        response.reservations?.forEach { res ->
            res.instances?.forEach { println("Instance:
${it.instanceId}") }
        }
    }
```

}

Expected Output:

• Kotlin program lists all S3 buckets.
• Files are uploaded to the desired S3 bucket programmatically.
• EC2 instances are queried and listed from the application.
• Program interacts securely with AWS using IAM credentials.

Conclusion:

In this experiment, we developed Kotlin programs using the AWS SDK for Kotlin to interact with services like S3 and EC2, authenticated using IAM credentials. This enables secure, serverless automation and integration of AWS services directly into Kotlin applications — a powerful practice in modern cloud-native development.

EXPERIMENT NO. 10

AIM:
To demonstrate the use of cloud computing security tools, focusing on notable open-source solutions for protecting cloud infrastructure, data, and services.

Prerequisites:
• Linux-based or cloud-hosted environment (local or on AWS/GCP/Azure).

• Internet access and basic command-line skills.

• Basic knowledge of cloud security concepts: IAM, encryption, firewalls.

• Admin privileges on test environment.

Theory:
Cloud security is a critical aspect of cloud computing, ensuring the confidentiality, integrity, and availability of cloud-hosted resources and data. Open-source tools are widely adopted for threat detection, access control, vulnerability scanning, and configuration auditing. These tools help identify misconfigurations, monitor traffic, and secure APIs.

Categories of Open Source Cloud Security Tools:

Category	Tools / Description

Identity & Access	Keycloak, OPA (Open Policy Agent): IAM and policy-based access control
Vulnerability Scanning	OpenVAS, Clair, Trivy: Detects security flaws in OS, containers, etc.
Configuration Auditing	ScoutSuite, Prowler: Audits AWS/Azure/GCP account configurations
Container Security	Falco, Sysdig OSS: Detect suspicious activity in containers/K8s
SIEM / Monitoring	Wazuh, OSSEC: Host and log monitoring, file integrity, threat response

Demonstration: Using Prowler for AWS Security Auditing

Step 1: Install Prowler

git clone https://github.com/prowler-cloud/prowler
cd prowler

Step 2: Configure AWS Credentials

aws configure

Step 3: Run Basic Audit

./prowler

Step 4: Run a Specific Check (e.g., S3 Bucket Permissions)

./prowler -c check31

Alternative Tool: Trivy – Container Vulnerability Scanner

Step 1: Install Trivy (Linux)

sudo apt install trivy

Step 2: Scan a Docker image

trivy image nginx:latest

Expected Output:

• Tools like Prowler generate HTML/JSON/terminal reports showing misconfigurations and compliance scores.
• Trivy lists CVEs found in container images.
• Insight into cloud security posture and weaknesses is gained.

Conclusion:

This experiment highlights how open-source tools like Prowler and Trivy can be used to assess and improve the security posture of

cloud environments. They help organizations enforce security best practices, identify misconfigurations, and prevent potential breaches — all crucial in a shared-responsibility cloud model.

Content Beyond Syllabus

EXPERIMENT NO. 11

AIM:
To study and demonstrate DevOps Tooling by AWS that helps organizations adopt a DevOps model effectively using automation, continuous integration (CI), and continuous delivery (CD) services.

Prerequisites:
• AWS account with access to developer tools.

• IAM user with appropriate permissions for CodePipeline, CodeBuild, CodeDeploy.

• Source code repository (e.g., GitHub or AWS CodeCommit).

• Basic understanding of CI/CD and DevOps principles.

• Web app (e.g., a sample Node.js/React app) for deployment.

Theory:
DevOps is the combination of development and operations practices aimed at shortening the system development life cycle while delivering features, fixes, and updates frequently in close alignment with business goals.

AWS DevOps Services provide fully managed CI/CD tooling that supports rapid and reliable software delivery.

AWS DevOps Tools Overview:

Service	Purpose
CodeCommit	Managed Git-based version control
CodeBuild	Compiles code, runs tests, creates deployment artifacts
CodeDeploy	Automates application deployment
CodePipeline	Automates entire release process (CI/CD pipeline)
CloudWatch	Monitoring and observability
CloudFormation	IaC for automating and provisioning infrastructure

Demonstration Scenario: Automating a Web App Deployment

• **Step 1:** Create a Git Repo (CodeCommit or GitHub)
Push your sample app (e.g., index.html, app.js, etc.) to the repo.

• **Step 2:** Create a CodeBuild Project
Set source provider (CodeCommit/GitHub).
Add buildspec.yml in the repo for build instructions:

```
version: 0.2
phases:
  install:
    runtime-versions:
      nodejs: 14
  build:
    commands:
      - echo "Build step"
artifacts:
  files:
    - '**/*'
```

• **Step 3:** Create a CodeDeploy Application
Set deployment group with EC2 or ECS as target.
Create and attach IAM roles.

• **Step 4:** Set Up CodePipeline
Source: GitHub/CodeCommit
Build: CodeBuild
Deploy: CodeDeploy
Save and release the pipeline.

Expected Output:

- On every code push, pipeline is triggered.
- CodeBuild compiles the project.
- CodeDeploy deploys the updated code to EC2/ECS.
- Logs and deployment status are visible in AWS Console

Conclusion:

In this experiment, we studied how AWS Developer Tools such as CodePipeline, CodeBuild, CodeCommit, and CodeDeploy help implement a DevOps lifecycle in the cloud. This experiment demonstrates how DevOps practices can be streamlined through automation, improving reliability, speed, and collaboration in modern software delivery pipelines.