Introduction to AWS (Amazon Web Services)

Amazon Web Services (AWS) is a **cloud computing platform** provided by **Amazon**, launched in **2006**. It offers a wide range of **on-demand IT services** such as computing power, storage, databases, networking, artificial intelligence, and more — all available via the internet on a **pay-as-you-go** basis.

Key AWS Services

AWS provides more than **200 fully featured services**, but the core categories include:

Compute

- **Amazon EC2 (Elastic Compute Cloud):** Virtual servers in the cloud.
- **AWS Lambda:** Run code without provisioning or managing servers (serverless computing).

Storage

- **Amazon S3 (Simple Storage Service):** Scalable object storage for data backup and archiving.
- **Amazon EBS (Elastic Block Store):** Persistent block storage for EC2 instances.

Database

- **Amazon RDS (Relational Database Service):** Managed SQL databases like MySQL, PostgreSQL, and Oracle.

Networking

- **Amazon VPC (Virtual Private Cloud):** Isolated networks within the AWS cloud.
- **Route 53:** Scalable Domain Name System (DNS) service.

AI and Machine Learning

- **Amazon SageMaker:** Build, train, and deploy ML models.
- **Amazon Rekognition:** Image and video analysis.

Security & Identity

- **AWS IAM (Identity and Access Management):** Manage user access and permissions securely.

Amazon EKS (Elastic Kubernetes Service) in AWS

Amazon EKS (Elastic Kubernetes Service) is a managed Kubernetes service provided by Amazon Web Services (AWS). It allows you to run, scale, and manage Kubernetes applications easily without having to install, operate, or maintain your own Kubernetes control plane or nodes.

What Is Kubernetes?

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications (such as those running in Docker containers)

EKS Cluster consists of two main components:

1. **Control Plane (Managed by AWS)**
    - AWS automatically manages the **Kubernetes master nodes** (API server, etcd, controller manager, scheduler).
    - The control plane is **highly available** and **scalable** across multiple Availability Zones.
2. **Worker Nodes (Managed by You or AWS)**
    - These are **EC2 instances** (or **Fargate serverless nodes**) where your containerized applications actually run.
    - The worker nodes connect to the control plane securely using the Kubernetes API.

GitHub

GitHub is a **web-based platform** for version control and collaboration that uses **Git**, a distributed version control system. It allows multiple people to work on a project at the same time without interfering with each other's work. GitHub is mainly used to host code, track changes, manage projects, and collaborate with others.

It's an essential tool for **open-source projects**, **team-based software development**, and anyone looking to store their code in the cloud.

Jenkins

Jenkins is an open-source automation server used primarily for Continuous Integration (CI) and Continuous Delivery (CD). It helps automate the process of building, testing, and deploying applications, which can streamline software development and improve the overall quality of software.

Key Jenkins Concepts:

- Pipeline: A series of automated steps (jobs) that are executed to build, test, and deploy applications.
- Job: A single task or set of tasks, such as building a project or deploying an application.
- Build: The process of compiling or packaging your code.
- Plugin: Jenkins is highly extensible through plugins, allowing integrations with other tools, including GitHub, AWS, Docker, etc.
- Slave/Agent: A machine that connects to the Jenkins master and runs builds.

Experiment 1

Creation of EKS Server using AWS step by step procedure.

Creating an Amazon EKS (Elastic Kubernetes Service) cluster on AWS involves several steps. Below is a step-by-step guide to set up an EKS cluster using the AWS Management Console, AWS CLI, and eksctl (a simple CLI tool for EKS). You can choose your preferred method depending on your needs.

Prerequisites

1. **AWS Account**: You should have an active AWS account.
2. **AWS CLI Installed**: If not, install AWS CLI.
3. **kubectl Installed**: Kubernetes command-line tool. Install kubectl.
4. **eksctl Installed**: eksctl is a simple CLI tool for EKS. Install eksctl.

Step-by-Step Procedure

1. Set up AWS CLI

- Open a terminal and configure AWS CLI:
- aws configure

Enter your **AWS Access Key ID**, **Secret Access Key**, **Region**, and **Output format**.

2. Create an EKS Cluster using eksctl

The easiest way to create an EKS cluster is by using eksctl. It abstracts much of the complexity and automates tasks like creating the EKS control plane, node groups, etc.

- **Create a Cluster**:
  To create a basic EKS cluster, run:
- eksctl create cluster \
- --name my-eks-cluster \
- --version 1.21 \
- --region us-west-2 \
- --nodegroup-name standard-workers \
- --node-type t3.medium \
- --nodes 3 \
- --nodes-min 1 \
- --nodes-max 4 \
- --managed
    - --name: Name of the cluster.
    - --version: Kubernetes version.
    - --region: AWS region.

- --nodegroup-name: Name of the node group.
- --node-type: EC2 instance type for the worker nodes.
- --nodes: Number of worker nodes.
- --nodes-min & --nodes-max: Minimum and maximum number of worker nodes.
- --managed: Tells eksctl to create a managed node group.

- **Verify Cluster Creation**:
  After running the command, it will take a few minutes for the cluster to be provisioned. Once done, verify the creation:

- kubectl get nodes

This will list your worker nodes in the EKS cluster.

3. Set up kubectl to Use Your EKS Cluster

- After the cluster is created, use the AWS CLI to update the kubeconfig:

- aws eks --region us-west-2 update-kubeconfig --name my-eks-cluster

This will configure kubectl to use the newly created EKS cluster.

4. Verify the Cluster Connection

- Verify that kubectl is connected to your cluster:

- kubectl get svc

This should return a list of services in your cluster (likely empty if you haven't deployed anything yet).

5. Create Kubernetes Deployments and Services (Optional)

- Now that the cluster is up and running, you can deploy applications to it. For example, to deploy an Nginx pod and expose it as a service:

1. **Create Deployment**:
2. kubectl create deployment nginx --image=nginx
3. **Expose the Deployment**:
4. kubectl expose deployment nginx --port=80 --type=LoadBalancer
5. **Check the Service**:
6. kubectl get svc

The service will have an external IP that you can use to access the Nginx application.

6. Scale the Cluster (Optional)

You can scale your EKS cluster by adding or removing nodes to the node group:

eksctl scale nodegroup \

--cluster my-eks-cluster \

--name standard-workers \

--nodes 5 \

--region us-west-2

7. Clean Up (Optional)

Once you're done with the cluster, you can delete it to avoid ongoing costs:

eksctl delete cluster --name my-eks-cluster --region us-west-2

Experiment 2

How to install docker in aws linux EC2 instances.

Step 1: Connect to your EC2 instance

First, SSH into your EC2 instance using the following command (replace your-ec2-public-ip with your EC2's public IP address):

ssh -i /path/to/your-key-pair.pem ec2-user@your-ec2-public-ip

Step 2: Update the System

It's a good idea to update the package index before installing any new software. Run the following command to update the system:

sudo yum update -y

Step 3: Install Docker

Amazon Linux 2 provides Docker in its default repositories, so you can install it using yum:

sudo yum install -y docker

This command will download and install Docker and its dependencies.

Step 4: Start the Docker Service

After Docker is installed, start the Docker service with the following command:

sudo service docker start

Step 5: Enable Docker to Start on Boot

To ensure that Docker starts automatically every time your EC2 instance reboots, run:

sudo systemctl enable docker

Step 6: Add Your User to the Docker Group (Optional but Recommended)

By default, Docker requires root privileges, but it's better to add your EC2 user (e.g., ec2-user) to the Docker group so you can run Docker commands without sudo.

Run the following command to add your user to the Docker group:

sudo usermod -aG docker ec2-user

After adding your user to the Docker group, you need to log out and log back in for the changes to take effect. Simply disconnect from the SSH session and then reconnect:

exit

Then log back in again:

ssh -i /path/to/your-key-pair.pem ec2-user@your-ec2-public-ip

Step 7: Verify Docker Installation

To verify that Docker is installed and running properly, run:

docker --version

You should see the Docker version installed on your system.

Next, test Docker by running the "hello-world" image:

docker run hello-world

If everything is set up correctly, Docker will download the "hello-world" image (if not already downloaded) and print a success message.

Step 8: (Optional) Configure Docker to Use the AWS ECR (Elastic Container Registry)

If you plan to pull Docker images from AWS ECR (Elastic Container Registry), you'll need to configure your instance to authenticate with ECR. This can be done by running the following AWS CLI command:

aws ecr get-login-password --region <your-region> | sudo docker login --username AWS --password-stdin <aws-account-id>.dkr.ecr.<your-region>.amazonaws.com

Replace <your-region> and <aws-account-id> with your specific AWS region and account ID.

Step 9: Install Docker Compose (Optional)

If you need Docker Compose (for managing multi-container Docker applications), you can install it with the following commands:

1. Download the Docker Compose binary:

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

2. Apply executable permissions to the binary:

sudo chmod +x /usr/local/bin/docker-compose

3. Verify the installation:

docker-compose --version

Step 10: Done!

You have successfully installed Docker on your Amazon Linux EC2 instance. You can now begin running containers on your EC2 instance.

Experiment 3

How to deploy a 2048 game app on AWS EKS cluster.

Deploying a **2048 game app** on an **AWS EKS (Elastic Kubernetes Service)** cluster involves several steps. You will need to containerize the application, set up an EKS cluster, and deploy the containerized application on Kubernetes.

Here's a step-by-step guide to deploy the **2048 game app** on AWS EKS:

Prerequisites:

1. **AWS CLI** installed and configured.
2. **kubectl** installed.
3. **eksctl** installed (a command-line tool for managing AWS EKS clusters).
4. A **Dockerized** version of the 2048 game app (you can either build your own or use an existing one).

Steps to deploy 2048 game app on EKS:

Step 1: Prepare the 2048 Game App Docker Image

If you don't already have a Docker image for the 2048 game app, you can use an existing one from Docker Hub or create your own.

Example Dockerfile (if creating your own):

Here is a simple Dockerfile if you were to build the 2048 app in a web environment (HTML, CSS, JS).

```
# Use a basic HTTP server to serve the app

FROM python:3.8-slim


# Install dependencies

RUN pip install http.server


# Copy the 2048 game app to the container (assuming you have it locally)

COPY . /app


# Set the working directory

WORKDIR /app


# Expose port 8000

EXPOSE 8000


# Start the HTTP server to serve the app
```

CMD ["python3", "-m", "http.server", "8000"]

1. Build your Docker image:
2. docker build -t 2048-game-app .
3. Test locally (optional):
4. docker run -p 8000:8000 2048-game-app
5. Push your Docker image to Amazon ECR (Elastic Container Registry):

First, create an ECR repository in your AWS account:

aws ecr create-repository --repository-name 2048-game-app --region us-east-1

Then log in to ECR and push the Docker image:

aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com

docker tag 2048-game-app:latest <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/2048-game-app:latest

docker push <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/2048-game-app:latest

Replace <aws_account_id> with your AWS account ID.

Step 2: Set Up an EKS Cluster

You can use **eksctl** to create an EKS cluster.

1. Create the EKS cluster:
2. eksctl create cluster \
3. --name 2048-cluster \
4. --region us-east-1 \
5. --nodegroup-name standard-workers \
6. --node-type t3.medium \
7. --nodes 3 \
8. --nodes-min 1 \
9. --nodes-max 4 \
10. --managed
11. Configure kubectl to access the cluster:
12. aws eks --region us-east-1 update-kubeconfig --name 2048-cluster

Step 3: Create Kubernetes Deployment and Service Manifests

Now, let's create Kubernetes manifests for the deployment and service.

1. **Create a deployment.yaml file**:
2. apiVersion: apps/v1
3. kind: Deployment
4. metadata:
5. name: 2048-game-app
6. labels:
7. app: 2048-game-app
8. spec:
9. replicas: 2
10. selector:
11. matchLabels:
12. app: 2048-game-app
13. template:
14. metadata:
15. labels:
16. app: 2048-game-app
17. spec:
18. containers:
19. - name: 2048-game-app
20. image: <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/2048-game-app:latest
21. ports:
22. - containerPort: 8000

Replace <aws_account_id> with your AWS account ID.

23. **Create a service.yaml file**:
24. apiVersion: v1
25. kind: Service
26. metadata:
27. name: 2048-game-app-service
28. spec:
29. selector:
30. app: 2048-game-app
31. ports:

```
    - protocol: TCP
      port: 80
      targetPort: 8000
  type: LoadBalancer
```

The LoadBalancer type service will expose the application to the internet via an external load balancer.

Step 4: Deploy the Application on EKS

Now that the Kubernetes manifests are ready, deploy them to EKS.

1. Apply the deployment and service files:
2. kubectl apply -f deployment.yaml
3. kubectl apply -f service.yaml
4. Check the status of the pods:
5. kubectl get pods

You should see pods being created and running.

6. Check the service to get the external IP address of the Load Balancer:
7. kubectl get svc

Look for the external IP in the EXTERNAL-IP column. It may take a few minutes to provision.

Step 5: Access the Game App

Once the Load Balancer is set up, you can access the 2048 game app using the external IP address from the kubectl get svc output.

Step 6: Clean Up Resources (Optional)

When you're done, you can clean up the resources by deleting the EKS cluster and other AWS resources.

1. Delete the EKS cluster:
2. eksctl delete cluster --name 2048-cluster --region us-east-1
3. Delete the ECR repository (optional, if you no longer need it):

aws ecr delete-repository --repository-name 2048-game-app --region us-east-1 --force

Experiment 4

How to integrate jenkins in GIT Hub

Step 1: Install Jenkins and Required Plugins

1. **Install Jenkins** (if not already installed):
    - You can follow the installation instructions for Jenkins based on your OS from Jenkins official website.
2. **Install Git and GitHub Plugins** in Jenkins:
    - Go to **Jenkins Dashboard** > **Manage Jenkins** > **Manage Plugins**.
    - In the **Available** tab, search for and install:
        - **Git Plugin**
        - **GitHub Plugin**

Step 2: Generate GitHub Personal Access Token (PAT)

1. **Go to GitHub** > **Settings** > **Developer settings** > **Personal access tokens**.
2. **Generate a new token**:
    - Select the following scopes:
        - repo (for private repositories)
        - admin:repo_hook (for webhook creation and management)
    - Click **Generate Token** and **copy the token** (you'll need it for Jenkins).

Step 3: Configure GitHub in Jenkins

1. **Go to Jenkins Dashboard** > **Manage Jenkins** > **Configure System**.
2. Scroll down to the **GitHub** section, and click **Add GitHub Server**.
3. Provide a name for the GitHub server (e.g., GitHub).
4. Under **Credentials**, click **Add** and select **Username with password**.
    - **Username**: Your GitHub username.
    - **Password**: Paste the GitHub Personal Access Token (PAT) you generated earlier.
5. Click **Test Connection** to ensure the connection works, then click **Save**.

Step 4: Create a New Jenkins Job

1. **Go to Jenkins Dashboard** > **New Item**.
2. Choose **Freestyle Project** and give your project a name (e.g., GitHub-Integration).
3. Click **OK**.

Step 5: Configure GitHub Repository in Jenkins Job

1. In your new Jenkins job, go to the **Source Code Management** section.
2. Choose **Git**.
3. In the **Repository URL**, enter the URL of your GitHub repository (e.g., https://github.com/username/repo.git).
4. Under **Credentials**, choose the GitHub credentials you added earlier.
5. Save the configuration.

Step 6: Set Up Build Triggers (GitHub Webhook)

1. Scroll down to the **Build Triggers** section in the Jenkins job configuration.
2. Check **GitHub hook trigger for GITScm polling**. This will allow Jenkins to trigger builds based on GitHub events like pushes or pull requests.

Step 7: Configure GitHub Webhook for Jenkins

1. Go to your **GitHub Repository**.
2. Go to **Settings** > **Webhooks** > **Add webhook**.
3. In the **Payload URL**, enter the URL of your Jenkins server's GitHub webhook endpoint:
    - Example: http://<your-jenkins-server>/github-webhook/
4. **Content type**: Select application/json.
5. Under **Which events would you like to trigger this webhook?**:
    - Select **Just the push event** (or **Let me select individual events** if you want more options like pull requests).
6. Click **Add webhook**.

Step 8: Test the Integration

1. Push a change or commit to your GitHub repository.
2. Go to Jenkins and check if the job triggers a build automatically.
3. If everything is set up correctly, Jenkins will start building the project whenever a new commit is pushed to GitHub.

Step 9: (Optional) View Build Logs

You can check the build logs in Jenkins to verify if the build was successful or if there were any errors. This can be done by: Clicking on the **job name** in Jenkins > **Build History** > **Build number** > **Console Output**.

Experiments 5

How to Build a Custom Docker Image and Run a Container

Objective: Build a custom Docker image (simple Python app) and run it on AWS EC2.

Steps:

1. **Create a Python app on your EC2 instance**:

SSH into your EC2 instance.

Create a simple Python app (e.g., app.py):

```
# app.py
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    return "Hello, Docker on AWS!"

if __name__ == "__main__":
app.run(host='0.0.0.0', port=5000)
```

step 2 : **Create a Dockerfile to containerize the Python app**:
In the same directory, create a Dockerfile:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install dependencies
RUN pip install -r requirements.txt
```

Expose port 5000

EXPOSE 5000

Run app.py when the container launches

CMD ["python", "app.py"]

Create a requirements.txt file to include the necessary dependencies:

Flask

Build the Docker image:
Run the following command to build your custom Docker image:

sudo docker build -t python-flask-app .

Run the Docker container:
After building the image, run the container:

sudo docker run -d -p 5000:5000 python-flask-app

Access your Python app:
Open your browser and go to http://<your-ec2-public-ip>:5000. You should see Hello, Docker on AWS!

Experiment 6

How to Run Docker Containers with AWS ECS (Elastic Container Service)

Objective: Deploy and manage Docker containers using AWS ECS.

Steps:

1. **Create an ECS Cluster**:
   - Go to the **ECS Console** in AWS.
   - Click **Create Cluster** and choose **Networking only** (Fargate).
   - Follow the steps to create the cluster.
2. **Create a Task Definition**:
   - In the ECS console, go to **Task Definitions** and create a new **Fargate** task definition.
   - Specify the container (e.g., your Docker image in ECR), memory, CPU, and networking configurations.
3. **Run a Task on ECS**:
   - Create a new service and run the task in your ECS cluster.
   - Specify the desired number of tasks and other settings like load balancing.
4. **Access the Application**:
   If you've set up a load balancer, access your web application using the load balancer's URL. If not, use the ECS task IP to access your application.

Experiment 7

How to Automatic Deployment of Code to EC2 Using GitHub Actions

Objective: Deploy code to an EC2 instance automatically when code is pushed to GitHub.

Steps:

1. **Launch an EC2 Instance** (Similar to previous experiments).
2. **Create a GitHub Action Workflow**:
    - Go to the **GitHub repository**.
    - In .github/workflows, create a file called deploy-to-ec2.yml to define your deployment process.

Example workflow for EC2:

```
name: Deploy to EC2


on:
  push:
    branches:
      - main  # Trigger on push to the main branch


jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2


    - name: Deploy to EC2
      env:
        PRIVATE_KEY: ${{ secrets.EC2_PRIVATE_KEY }}
        EC2_HOST: ${{ secrets.EC2_HOST }}
      run: |
        echo "$PRIVATE_KEY" > private_key.pem
        chmod 600 private_key.pem
        ssh -o StrictHostKeyChecking=no -i private_key.pem ec2-user@$EC2_HOST 'cd /home/ec2-user/my-app && git pull origin main && npm install && npm start'
```

3. **Set up SSH Keys**:

- Generate an SSH key pair and add the public key to the ~/.ssh/authorized_keys file of your EC2 instance.
- Store the private key in GitHub secrets (EC2_PRIVATE_KEY).

4. **Push Changes**:
   - Push changes to the main branch of your repository.
   - The GitHub Actions workflow will automatically SSH into your EC2 instance and deploy the code (pull from GitHub, install dependencies, and run the app).

NOTE : Take Two screenshots from each experiments and paste it in record.